Doctoral Thesis

Manycore Architectures and SIMD Optimizations for High Performance Computing

Marcos Horro Varela

2022

UNIVERSIDADE DA CORUÑA

Manycore Architectures and SIMD Optimizations for High Performance Computing

---

Marcos Horro Varela

Doctoral Thesis

March 2022

PhD Advisors:

Gabriel Rodríguez Álvarez

Juan Touriño Domínguez

PhD Program in Information Technology Research

UNIVERSIDADE DA CORUÑA

Dr. Gabriel Rodríguez Álvarez
Profesor Titular de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

Dr. Juan Touriño Domínguez
Catedrático de Universidad
Dpto. de Ingeniería de Computadores
Universidade da Coruña

CERTIFICAN

Que la memoria titulada "*Manycore Architectures and SIMD Optimizations for High Performance Computing*" constituye un trabajo original de investigación realizado por D. Marcos Horro Varela bajo nuestra dirección en el marco del Programa de Doctorado en Investigación en Tecnologías de la Información de la Universidade da Coruña, y concluye la Tesis Doctoral que presenta para optar al grado de Doctor con la Mención de Doctor Internacional.

En A Coruña, a 1 de marzo de 2022

Fdo.: Gabriel Rodríguez Álvarez
Director de la Tesis Doctoral

Fdo.: Juan Touriño Domínguez
Director de la Tesis Doctoral

Fdo.: Marcos Horro Varela
Autor de la Tesis Doctoral

A meus pais e a Marta.

Agradecementos

Non é doado condensar nun par de liñas todas as sensacións e sentimentos atopados despois dun arduo traballo de máis de catro anos, cunha pandemia mundial de por medio. A investigación é, en moitas ocasións, un traballo individual, solitario e frustrante, pero tamén curioso, interesante, satisfactorio, enriquecedor e que pode ser compartido con milleiros de persoas en todo o mundo, non só no ámbito científico. Experimentei todas as facianas emocionais nesta travesía, pero agora só podo ter palabras de agradecemento a todas as persoas e institucións que apoiaron este traballo e me permitiron completar esta Tese de forma satisfactoria.

Tiven a sorte de que Gabriel e Juan contactaran comigo en setembro do 2014. Primeiro para completar o traballo de fin de grao, despois o do mestrado, e así continuar ca Tese Doutoral. Agradecerlles infinitamente a súa confianza durante estes anos, polos momentos nos que os resultados non acompañaban, e por todas as facilidades laborais e académicas que me permitiron investigar aquelas materias do meu interese. En especial a Gabriel, por ser o meu salvavidas tanto no persoal como no laboral, gracias de verdade. Do mesmo xeito, agradecer a todos os meus compañeiros dos laboratorios 0.2 e 1.2, con especial mención por estes anos a Toño, Christian, Diego, Sergio, Santa Comba, Róber, Deibe, Ana, Pérez, Andión, Jorge, e Nuria; sen vós o camiño sería aínda máis pesado. Tamén aos meus compañeiros e amigos de grao e mestrado na Coruña, con mención a Feal, Rubén, Fran, Avelino e Eloy. Estender o meu agradecemento e apoio institucional a todos os profesores

e membros do Grupo de Arquitectura de Computadores, ao Departamento de Enxeñaría de Computadores, á Facultade de Informática da Coruña e á Universidade da Coruña. Simplemente grazas.

Gustaríame agradecer ao meu ex-profesor J. B. Búa por despertar a miña curiosidade en canto ás matemáticas e á investigación cando aínda cursaba 3º da ESO.

The same way, I gratefully thank Dr. Louis-Noël Pouchet for hosting me and advising me in one of my best experiences working abroad at the Colorado State University, Fort Collins. It has been an honor working those three months with you, and extending our collaborations further, *merci beaucoup*. Also thanks to my *mecedoras* Lucía and Hassna for making my stay great in Fort Collins.

Agradecementos tamén ao resto de entidades financeiras deste traballo: ás redes estatais de investigación, ás redes europeas, á Xunta de Galicia, e ao Goberno de España. Tamén agradecementos á empresa Inditex S.A. polo financiamento da estadía nos EUA.

Se cheguei ata aquí é por meus pais, os meus piares vitais: nunca terei palabras suficientes para agradecer o voso sacrificio por darme a mellor educación e por darme todos os medios posibles para chegar a ser a persoa que son hoxe en día, gracias infinitas. Gracias tamén ao meu irmán Jorge polo seu apoio incondicional, e a toda a miña familia por crer en min. Gracias tamén á miña outra familia: gracias aos meus amigos de Cambados por estar sempre aí, por facer que calquera sitio sexa como estar na casa, con mención a Me, Clara, Emm, Xabi, Martin, Deivid, Johnny, Agarunde, Piñe, Duri, Angelote e Vítor. Gracias aos meus amigos de Coruña, con especial mención a Domingo, aos meus Javis e a Miwel.

A Marta, porque sen ti o que vén despois disto deixa de ter sentido, a ti por ser o meu remanso nesta viaxe tan movida.

Non caben nestas liñas todas as palabras de agradecemento. Moitas gracias de todo corazón.

A Coruña
Markos Horro

"La ciencia es un mito, sólo que es el mito más hermoso,
el único generalizable a toda la especie y
quizás el más digno de respetarse."
–Antonio Escohotado

"En tanto en cuanto nos 'dean' lo que es nuestro,
discutiremos ese 'conceto' con el fin de discutirlo."
–M. Manquiña

Resumo

Nos últimos cincuenta anos a arquitectura de computadores estivo marcada pola capacidade de aumentar o número de transistores nos microchips segundo a Lei de Moore. Esta corrente cambiou drasticamente na última década. O paralelismo xoga un factor clave nos deseños modernos: dende un punto de vista *hardware* aumentando o número de CPUs, e dende unha perspectiva *software* explotando as capacidades arquitectónicas con especial énfase nas unidades vectoriais. Nesta Tese poñemos o foco en dúas dimensións ortogonais: a análise e optimización do tráfico de coherencia en arquitecturas *manycore*, e o desenvolvemento de optimizacións vectoriais. Na primeira dimensión desenvolvemos técnicas estáticas e dinámicas para mellorar a afinidade entre as CPUs e os datos naquelas arquitecturas que implementan redes de interconexión en malla. A nosa proposta reduce a contención nesas mallas mellorando a localidade dos datos de acordo á distribución física dos compoñentes. Na segunda dimensión desenvolvemos un compilador fonte-a-fonte para vectorizar códigos con patróns de acceso a memoria irregulares. Presentamos dúas contribucións: estratexias para recuperar de forma vectorial direccións de memoria non contiguas, e a fusión de reducións independentes. Desenvolvemos un sistema SMT para a xeración de alternativas de empaquetado de operandos aleatorios baseadas no conxunto de instrucións da arquitectura subxacente, e unha ferramenta para a caracterización de arquitecturas. A nosa avaliación mostra potenciais beneficios nestes códigos irregulares aplicando as nosas propostas.

Resumen

En los últimos cincuenta años la arquitectura de computadores estuvo condicionada por la capacidad de aumentar el número de transistores en los microchips siguiendo la Ley de More. Esta tendencia cambió drásticamente en la última década. El paralelismo es ahora un factor clave en los diseños modernos: desde una perspectiva *hardware* aumentando el número de CPUs, y desde un punto de vista *software* explotando las capacidades arquitecturales con especial énfasis en las unidades vectoriales. En esta Tesis ponemos el foco en dos dimensiones ortogonales: el análisis y optimización del tráfico de coherencia caché en arquitecturas *manycore*, y el desarrollo de optimizaciones vectoriales. Para la primera dimensión hemos desarrollado técnicas estáticas y dinámicas para mejorar la afinidad entre las CPUs y los datos en arquitecturas con redes de interconexión en malla. Nuestra propuesta reduce la contención en las mallas mejorando la localidad de los datos siguiendo la distribución física de los componentes. En la segunda dimensión hemos desarrollado un compilador fuente-a-fuente para vectorizar códigos con patrones de acceso a memoria irregulares. Presentamos dos contribuciones: estrategias para recuperar de forma vectorial direcciones de memoria no contiguas, y la fusión de reducciones independientes. Hemos desarrollado un sistema SMT para la generación de alternativas de empaquetamiento de operandos aleatorios basadas en el conjunto de instrucciones de la arquitectura subyacente, y una herramienta para la caracterización de arquitecturas. Nuestra evaluación muestra potenciales beneficios en estos códigos irregulares aplicando nuestras propuestas.

Abstract

For the past fifty years computer architecture has been driven by the ability to etch more transistors onto a single die following the Moore's Law. This trend changed in the past decade. Parallelism is now a key factor in modern designs: from the hardware side by scaling the number of cores, and from the software side by exploiting the capabilities available in the architecture, emphasizing on the SIMD units. In this Thesis we focus on two orthogonal dimensions: the analysis and optimization of coherence traffic in modern manycores, and the development of SIMD optimizations. For the first dimension we develop static and dynamic techniques for enhancing core-to-data affinity for manycores featuring mesh interconnection networks. Our approach reduces the contention on these meshes by improving data locality according to the physical layout. For the second dimension we develop a source-to-source compiler for vectorizing codes presenting irregular access patterns. We present two main contributions: strategies for gathering non-contiguous memory addresses, and fusing independent reductions. We have developed an SMT-based system to generate alternatives for packing random operands from memory based on the host ISA, and a profiling framework for characterizing platforms. Our evaluation shows promising speedups when applying these SIMD optimizations to those codes.

Preface

Parallelism has become more important in hardware development as the frequency increase has reached its physical limit due to the power wall. Nowadays, parallelism is present at different levels in high performance computing: from instruction decoding and execution (Instruction-Level parallelism, ILP) to the number of interconnected nodes in a cluster or worldwide. From an architectural point of view, parallelism is exhibited in the number of cores etched on a single die. However, increasing the degree of components interconnected brings forward scalability complications. In addition, cores are, nowadays, extremely sophisticated by implementing complex pipelines featuring wide vector length capabilities. In this way, the present Thesis, "*Manycore Architectures and SIMD Optimizations for High Performance Computing*", addresses these two orthogonal dimensions by analyzing modern manycore architectures and focusing on discovering potential design improvements at two different levels of the architectural stack, and providing techniques for synthesizing efficient platform-aware SIMD code.

Objectives and Work Methodology

The main objectives of this Thesis are listed below, including some key sub-goals.

1. Analysis and modeling of the Intel Xeon Phi x200 (Knights Landing, KNL).

- Analysis and characterization of the core architecture, distributed cache coherence directory and interconnection network.
- Implementation of the Knights Landing architectural model by extending an architectural simulator (Tejas Simulator).
- Experimental assessment of the simulator accuracy by running different workloads.

2. Optimizing coherence traffic in manycore architectures.

   - Approach to discover the physical layout and location of components.
   - Reverse engineering of hash address functions on KNL.
   - Runtime- and compile-time-based approaches to optimize the coherence traffic in the interconnection network.
   - Evaluation of the static and dynamic approaches proposed.

3. Development of a profiling and performance analysis toolkit specifically designed for experiments requiring the configuration of many parameters.

   - Automatic compilation, execution and analysis given any set of programs or benchmarks and parameters of interest, e.g., the size of a matrix, the step in a loop, etc.
   - Data mining techniques for extracting knowledge from experiments based on the target dimensions, i.e., to quantify the influence of variables.
   - Toolkit compatible with any type of application, designed for increasing the productivity, quality, and repeatability of the experiments.

4. Synthesis of efficient x86 SIMD code for random vector packing and fusion of reductions.

   - Generation of random vector packing combinations using the instructions available given a concrete ISA, based on an SMT model.
   - Building of a cost model driven by the empirical performance based on the micro-benchmarking of those combinations for each concrete platform.
   - Development of a source-to-source compiler for synthesizing efficient vector code for each platform based on the individual characterization.
   - Evaluation using different target applications.

Funding and Technical Means

The means that were used to carry out this Thesis have been the following:

- Working material, human and financial support primarily by the Computer Architecture Group of the University of A Coruña, along with the Research Fellowship funded by the Ministry of Education of Spain (FPU program, ref. FPU16/00816).
- Access to bibliographical material through the library of the University of A Coruña.
- Additional funding through the following research projects:
    - Regional funding by the Galician Government (Xunta de Galicia) under the Consolidation Program of Competitive Research Groups (Computer Architecture Group, refs. ED431C 2017/04 and ED431C 2021/30) and Network of Cloud and Big Data Technologies for HPC (ref. ED431D R2016/045).
    - State funding by the Ministry of Science and Innovation of Spain through the projects "New Challenges in High Performance Computing: from Architectures to Applications (II)" (ref. TIN2016-75845-P) and "Current Challenges in HPC: Architectures, Software and Applications" (ref. PID2019-104184RB-I00).
    - European funding: project "Network for Sustainable Ultrascale Computing" (NESUS COST Action ref. IC1305).
    - European Network on High Performance and Embedded Architecture and Compilation (HiPEAC) competitive grant for attending the 13th International Summer School on Advanced Computer Architecture and Compilation for High Performance and Embedded Systems (ACACES 2017).
- Access to clusters, supercomputers and computing platforms:
    - *Pluton* cluster (Computer Architecture Group, University of A Coruña) featuring 30 nodes with 688 physical cores (1376 threads), including 3

nodes with Cascade Lake architecture, 4.1 TiB of memory, and 2 many-core accelerators Intel Xeon Phi (Knights Corner).

- Intel Xeon Phi x200 standalone machine (Computer Architecture Group, University of A Coruña).
- Access to a private cluster within the Department of Computer Science at Colorado State University, USA.

- A three-month research visit to Colorado State University. This visit was funded by Inditex-University of A Coruña through a competitive grant under the "inMotion Program".

Structure of the Thesis

The Thesis is organized as follows:

1. Chapter 1 introduces the main challenges in current high performance computing, presenting the main topics that will be investigated in the Thesis.

2. Chapter 2 explores modern manycore architectures and the effect of coherence traffic in the interconnection network. We also perform reverse engineering for obtaining the hash address functions used for distributing data across tiles. Equipped with this information, we propose static and dynamic approaches to leverage this data distribution by improving spatial locality in programs. We conclude this chapter by assessing the solutions proposed, and describing the limitations of our approaches using different sets of benchmarks and applications.

3. Chapter 3 continues the topic of the previous chapter by analyzing and modeling a modern manycore architecture (Intel Xeon Phi x200 Knights Landing) and its interconnection network. In this chapter we also describe the development of an extension for the Tejas simulator (a cycle-accurate architectural simulator) based on this model, assessing its accuracy against real hardware.

4. Chapter 4 presents MARTA, which stands for Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. This is a toolkit developed

for improving productivity and automating error-prone tasks when profiling micro-benchmarks and/or regular applications based on tested-and-true good practices. It also implements data mining and machine learning techniques for analyzing and extracting knowledge from profiling data. We assess its performance with five valuable case studies, and using it for running and profiling benchmarks and applications.

5. Chapter 5 proposes novel data packing techniques for random memory addresses into the same vector register, as well as for the fusion of different reductions. For this purpose, we have developed a novel SMT-based system for generating random packing templates based on a given ISA. Leveraging this information, we have built a cost model driven by the empirical performance of those templates on each platform. This model is then used by MACVETH (Multi-Architectural C-VEcTorizer for HPC applications), a novel source-to-source compiler that is able to pack any random set of memory addresses in order to vectorize a set of operations, including reductions. This compiler is also able to pack and fuse independent reductions together. We conclude this chapter assessing the quality and the use cases of our approach.

6. Chapter 6 concludes the Thesis discussing some final remarks, and proposing additional research lines that are worth considering as future work.

Main Contributions

The main original and novel contributions of this Thesis are stated below:

- Development of an extension for the Tejas Simulator to explore the KNL architecture or any other similar one featuring a distributed directory system. This extension allows to analyze the coherence traffic over the interconnection network [44, 46].

- Reverse engineering of the Intel Knights Landing architecture to discover its physical layout. Based on this, we developed ways to optimize the coherence traffic, the thread-to-core-affinity, and the scheduling of a set of tasks on

the mesh, leveraging the unique characteristics of a particular processor unit stemming from process variations [45].

- Uncovering the pseudo-random mapping function of physical memory blocks across the pieces of the distributed directory in KNL. Leveraging this knowledge, candidate optimizations to improve memory latency through the optimization of coherence traffic are studied. Although these optimizations do improve memory throughput, ultimately this does not translate into performance gains due to inherent overheads stemming from the computational complexity of the mapping functions [68].

- Development of MARTA, a toolkit for profiling and performance analysis meant to increase productivity [50, 52]. This toolkit is not a substitute for any other well-known toolkits available, but its main and novel contribution is the emphasis on the automation, improving productivity and quality of results. In an orthogonal dimension, the toolkit also includes a module for performance analysis, using data mining and machine learning techniques.

- Development of MRKVS (Mega-Random Kernel Vector SMT), an SMT-based system for generating any random packing template given an ISA. From these templates, we build a cost model for enabling random vector packing.

- Development of MACVETH (Multi-Architectural C-VEcTorizer for HPC applications) [51], a source-to-source compiler for synthesizing efficient SIMD code on specific regions of code featuring patterns with irregular memory accesses. This compiler includes the cost model built with MRKVS, and heuristics for vectorizing the fusion of independent reductions.

Developed Software

The software libraries and tools developed in this Thesis are publicly available:

- Tejas KNL. Custom implementation of the Intel Knights Landing architecture on the Tejas Simulator. Available at `https://github.com/UDC-GAC/tejas_knl`.

- papi_wrapper: C macro-based library for simplifying the use of the PAPI library. Available at https://github.com/UDC-GAC/papi_wrapper.
- MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. Framework built for increasing productivity and quality of the experiments requiring (micro-)benchmarking and post hoc performance analysis. Available at https://github.com/UDC-GAC/MARTA.
- MRKVS: Mega-Random Kernel Vector SMT. Z3-based [23] system for generating combinations of instructions for packing random data on the same vector register. Available at https://github.com/UDC-GAC/MRKVS.
- MACVETH: Multi-Architectural C-VEcTorizer for HPC applications. Source-to-source C compiler for vectorizing irregular random memory accesses and reductions. Available at https://github.com/UDC-GAC/MACVETH.

Publications from the Thesis

Journal publications

- S. Kommrusch, M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Optimizing Coherence Traffic in Manycore Processors Using Closed-Form Caching/Home Agent Mappings. *IEEE Access*, 9:28930–28945, 2021. doi: 10.1109/ACCESS.2021.3058280. JCR Q2 [68].
- M. Horro, G. Rodríguez, and J. Touriño. Simulating the Network Activity of Modern Manycores. *IEEE Access*, 7:81195–81210, 2019. doi: 10.1109/ACCESS.2019.2923855. JCR Q1 [46].

International conferences

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. Submitted for publication. 2022 [52].

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MACVETH: Multi-Architectural C-VEcTorizer for HPC applications. Submitted for publication. 2022 [51].

- M. Horro, M. T. Kandemir, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Effect of Distributed Directories in Mesh Interconnects. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 51:1–6, Las Vegas, NV, USA, 2019. doi: 10.1145/3316781.3317808. Core A. GII-GRIN-SCIE Class 1 [45].

National conferences

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Toolkit para (Micro-) Benchmarking y Análisis de Características de Rendimiento en Kernels. In *Actas XXXI Jornadas de Paralelismo (SARTECO)*, pages 303–312, Málaga, Spain, 2021 [50].

Other minor publications

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Exploring SIMD Instructions for Packing Random Vector Operands in Modern x86 CPUs. In *Proceedings of the 17th International Summer School on Advanced Computer Architecture and Compilation for High-Performance Embedded Systems (ACACES)*, pages 143–146, Fiuggi, Italy, 2021 [49].

- M. Horro, G. Rodríguez, J. Touriño, and M. T. Kandemir. Study of the Intel Knights Landing (KNL) Memory System Tradeoffs. In *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, pages 1–4, Fiuggi, Italy, 2017 [44].

Contents

List of Tables

List of Figures

Listings

List of Algorithms

"But what... is it good for?"

–Engineer at the Advanced Computing Systems Division of IBM commenting on the nascent microchip in 1968

1 Introduction: Challenges in High Performance Computing

"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (...). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."

– Gordon E. Moore [94]

It has been more than fifty five years, in April 1965, since Gordon Moore was first asked to predict the evolution of semiconductors in the following years. He is one of the "traitorous eight" who left Shockley Semiconductor Laboratory to found Fairchild Semiconductors in 1957. There, just a couple of years later, Robert Noyce, also co-founder, invented the planar integrated circuit. Moore, by observing electronics trends and the pace followed by semiconductor industry in the previous years, predicted that the number of components on those integrated circuits would double every year for the next decade. This "simple" forecast became prophecy. Just three years later, in 1968, he co-founded with, again, Robert Noyce a new company

named NM Electronics, which was later renamed as Intel Corporation. Another spin-off from Fairchild, Advanced Micro Devices (AMD), was also founded a year later. That decade passed, and it was in 1975 when he revised again his prognosis adjusting the increase rate to be doubling every two years, establishing what we have studied and known as "Moore's Law", due to its accuracy. Nevertheless, this prediction is commonly quoted to be doubling every eighteen months; but that was never stated by Moore [54].

The extraordinary linearity in the prediction led to an incredible growth in industry, as manufacturing costs dropped rapidly. In such way, technology started to reach the masses. From those 65,000 components in the seventies, we have reached more than 2.6 trillion or 2.6×10^{12} ($> 3 \times 10^7 \times 65,000$) transistors on a single die by 2021 [123]. Nevertheless, *cramming more components onto integrated circuits* was not the only cause for this incredible growth.

Moore's Law was stated from an economic perspective first, although soon was correlated with performance as smaller transistors could switch at higher speeds, i.e., clock frequencies could be increased without incurring timing errors. Nonetheless, frequencies have not increased at the same rate as transistors have shrunk. Dennard's scaling states, in a nutshell, that power density remains constant as transistors shrink. This statement was based on MOSFET design power equations, as described and simplified in Equation 1.1, which establishes a relation between power dissipated (P), switching gate frequency (F), drain voltage (V_{dd}) and dynamic capacitance (C), a factor depending on manufacturing process; being α the switching activity factor.

$$P = \alpha C F V_{dd}^2 \tag{1.1}$$

An important aspect of Dennard's scaling is that as transistors got smaller the voltage could be reduced, enabling chips to operate at higher frequencies, remaining that power density constant in all chip area. But only these equations ignored two certainly important and limiting factors: the minimal threshold voltage and the leakage current. The first one establishes the minimum amount of potential needed to have a transistor turned on without misfunction. The second one poses difficulties for safe thermal dissipation, narrowing the budget of active transistors supported the

same interval of time. These inabilities or limitations establish what is commonly referred to as the "power wall".

It is debatable, and out of the scope of this Thesis, whether Moore's Law has withered or not, or how long it will hold, but it is clear that with Dennard's scaling breakdown trends in micro-architecture have changed. Frequency is no longer the key factor in terms of performance improvements as recent advances in GPU performance have demonstrated [28], motivating the development of new approaches for reaching Exascale milestone, such as dark silicon or energy-aware solutions, heterogeneous, reconfigurable, and manycore architectures. These last ones are the most promising candidates as they pack within the same processor an interconnection network with a high number of independent processing units, enabling a high degree of parallelism. Exascale computing [9] refers to the ability of executing 10^{18} floating-point operations within a second (FLOP/s). This is one of the most ambitious short-term challenges in high performance computing. Even though it has been reached under certain circumstances using distributed computing, by 2021 there are no listed supercomputers in the Top500 [125] reaching that performance. This order of FLOP/s would enable better accuracy in complex scientific applications and tasks such as weather forecasting [9], which relies on a vast amount of mutually dependent parameters; neural activity (The Human Brain Project [81]), which requires to simulate billions of interconnected neurons; personalized medicine, in order to classify pathologies based on medical history, current vital signs, etc.; some applications in fluid dynamics that would need more accurate solutions than current approximations for partial differential Navier-Stokes equations; and many others. As such, it is a critical milestone for computer engineering to reach.

For these reasons, the first objective of this Thesis is the exploration of the modern manycore architectures mentioned above. We target the Intel Xeon Phi x200 Knights Landing architecture, nowadays discontinued even though its high-level architectural legacy lives on the newer Intel Xeon Scalable generations. We focused on the performance of its mesh interconnection network, as this architecture was the first attempt by Intel to reach Exascale computing. The key idea beneath its design was to provide an efficient mesh of interconnected AVX-512-capable cores, organized in tiles, and equipped with a 3D-stacked high-bandwidth memory on-package. In theory, these features should allow highly parallel applications to fully

exploit memory bandwidth, and take advantage of large vector widths to improve the overall performance in terms of FLOPS/s. This architecture also presented a distributed directory for keeping coherence within the tiles and cores in the mesh in the form of a Cache/Home Agent (CHA). Memory addresses are distributed among slices of this distributed directory according to a non-disclosed hash function, i.e., each tile holds a set of the memory addresses in the system. Accesses to memory are supposed to present a UMA behavior, but, as we will detail later, after some experiments we discovered significant deviations on these memory latencies. With this knowledge we first built a model based on reverse engineering of the physical layout for developing an extension for the Tejas Simulator in order to model the behavior of the traffic in the interconnection network. Leveraging this information, we also developed static and dynamic techniques for improving the core-to-CHA affinities in the applications for reducing traffic contention and memory latencies in the network. Our findings and methodology employed are described in detail in Chapters 2 and 3.

With Knights Landing, Intel also introduced new instructions using up to 512-bit width vectors with the AVX-512 instruction set. This new extension clearly stated the intention to exploit Instruction- and Memory-Level Parallelism (ILP and MLP, respectively) in high performance computing. Notwithstanding the potential benefits for memory bandwidth, AVX-512 presented some limitations in Knights Landing (and some subsequent generations) since it reduced drastically the CPU frequency [24, 26] when running certain instructions of this SIMD ISA. In addition, there are many variants of AVX-512, increasing the fragmentation in x86 architectures. Specifically, this fragmentation in the SIMD ISAs manifests the need of synthesizing ad hoc code for each platform. But these are not the only limitations: compilers according to their cost model could also decide not to use these vector capabilities, or even the very nature of the codes might not be suitable for those large vector widths without additional tuning. There are codes, such as Sparse Matrix-Vector Multiplication (SpMV), which are important kernels in machine and deep learning applications, presenting sparse and irregular accesses to memory (i.e., non-contiguous memory accesses). These access patterns prevent compilers, in most cases, to perform any type of vectorization in the code. Accordingly, the second objective of this Thesis is the development of SIMD optimizations by 1) proposing a novel approach for packing those random vector operands, and 2) fusing operations

such as independent reductions, also present in irregular codes such as SpMV. We propose an SMT-based system for automatically generate these packing candidates based on the instructions available in the ISA. With these candidates, we can generate a cost model for each platform according to the best candidate based on its performance. For automating the generation of these cost models we also developed a profiling and performance analysis toolkit, which is described in Chapter 4. This toolkit was conceived for building these costs models empirically, but it can be used for profiling any other kernel or application. All this toolchain leads to the source-to-source compiler we have implemented for synthesizing efficient platform-aware SIMD code. This compiler is fully detailed in Chapter 5, including the SIMD optimizations described for the packing and fusion of operands and operations in a vector fashion. Finally, conclusions and future work of the Thesis are described in Chapter 6.

"When in doubt, use brute force."

–Ken Thompson

2

Effect of Distributed Directories and Optimization of Coherence Traffic in Manycores

Chapter's contents

In this chapter we will review and study the most important aspects of modern manycore architectures, focusing on the Intel Knights Landing architecture and its distributed cache coherence directory, which play a key role in the optimization and

scaling of modern manycore architectures. We also propose and assess optimizations for improving data locality by leveraging architectural details of the processor.

2.1 Introduction

Computer architecture has evolved quickly in the past century, from very simple monolithic cores to thousands of interconnected processors in large clusters. Parallelism plays an important role in this evolution, as manycore architectures are one of the most promising candidates to reach the Exascale computing era. Manycore architectures are an evolution of classic multicore architectures which increase the number of interconnected cores. Coherence traffic was a limiting factor in classic multicore architectures, as scaling the number of cores on a cache coherent system caused a growth in the overhead associated with the cache coherence protocol, limiting the ability of a program to extract increased performance from the system. This was known as the "coherence wall" [70, 82]. Manycore architectures typically implement distributed cache directories to alleviate this limitation. This directory must be accessed each time a core requests access to a memory block which is not already locally available in the appropriate state (i.e., not invalid). This distributed design increases the scalability of the coherence system by removing the bottlenecks that a centralized directory would impose: the roundtrip from each core to the directory and the volume of requests that the directory would process. Intel implemented this distributed mechanism on its first manycore architecture Knights Landing [8]. This architecture organizes the processing units in a mesh of tiles, where each tile features two cores, a shared L2 cache, and a portion of the distributed directory named Caching/Home Agent (CHA). Each CHA holds a set of cache line addresses, which are distributed among all CHAs using a built-in address hash. As such, cache miss requests are bypassed directly out to the mesh to be serviced by a specific CHA as determined by the address hash. However, this CHA has also a cost, and quantifying the impact on performance is not trivial, as non-disclosed hash functions are used for distributing data, hardening any programmatically optimization. For these reasons, in this chapter we propose a reverse engineering process for obtaining those hash functions, an analysis of the effect of core affinity on memory latency, and different techniques for improving data locality on these architectures. More

specifically, in this chapter we describe the following contributions:

- We propose a mechanism to discover the physical layout of the logical components (cores and CHAs) of a mesh interconnect-based processor, as well as the mapping of memory blocks across CHAs and memory interfaces (Section 2.3).
- Leveraging the previous contribution, we analyze the impact of coherence traffic in the memory latency of distributed directory architectures. Mechanisms to optimize coherence traffic are proposed, improving core-to-CHA and thread-to-core affinity (Section 2.4).
- We gather and analyze data generated by a large number of executions on a KNL unit, and develop optimized strategies for scheduling a set of tasks across the tiles in the mesh. We perform experiments to quantify the efectiveness of our optimizations. Our results reveal that exploiting the multiple opportunities for locality in a mesh interconnect is essential to increase the potential performance of future manycores (Section 2.5).
- With the information extracted in Section 2.3, the mapping of memory blocks to CHAs is reverse engineered. Binary functions which compute a target CHA from a physical memory address are exposed and shown to be pseudo-random in nature (Section 2.6).
- Different optimization strategies to improve memory latency by leveraging the mappings between memory and CHAs are designed. Approaches are proposed based on both dynamic and static work scheduling (Sections 2.7 and 2.8).
- Experiments are performed to quantify the effectiveness of the proposed optimizations. It is shown how the proposed schedulings improve memory latency by exploiting CHA proximity. However, due to the pseudo-random nature of the block-mapping functions the implementation of these schedulings affects other performance-impacting factors, which may ultimately lead to performance degradation (Sections 2.7.1 and 2.8.2).

The rest of the chapter is structured as follows: Section 2.2 delves deeper into manycore architectures and the Intel KNL. Section 2.3 presents the reverse engineering process to map logical components of an MI-based architecture to its physical

layout. In Section 2.4 we describe the approach followed to optimize the coherence traffic. Section 2.5 evaluates the potential advantages of the proposed approach, and develops ways to exploit the architectural characteristics of a particular KNL unit. Section 2.6 details the reverse engineering process that leads to the discovery of the memory-to-CHA mapping. Sections 2.7 and 2.8 detail runtime- and compile-time-based approaches, respectively, to optimize coherence traffic, and summarize the results of the experimental evaluation phase. Finally, Section 2.9 discusses the results obtained, the viability of the approaches presented, and related work.

2.2 Intel Knights Landing (KNL): Xeon Phi x200

The increasing demand of computational resources over the last decade has shifted architectural paradigms. The relationship between energy consumption and frequency, known as Dennard's scaling, is not linear anymore [27]. The "multicore crisis" in the past decade was partially a response to this problem. The idea behind manycore processors is to comply with thermal limitations by integrating a higher number of simpler, slower processors, capable of taking advantage of embarrassingly parallel applications or to execute many smaller workloads at the same time. Even though cores are simpler, they can communicate more efficiently since they are integrated inside a single processor die and connected to a network-on-chip (NoC).

Manycore organizations present a challenge for the memory system. Since more data-hungry cores coexist now inside a single die, the memory wall grows higher. Modern architectures propose to use heterogeneous memory hierarchies, which combine different memory technologies with their own characteristics and trade-offs. Traditional memory hierarchies are augmented with these new technologies as shown in Figure 2.1, contributing to reduce the gap between processor and memory speeds.

This work focuses on the Intel Xeon Phi x200 architecture, codenamed Knights Landing (KNL), released in 2016 and discontinued in mid-2018. However, the distinguishing characteristics of its NoC live on the newer Intel developments for HPC, namely the Xeon Scalable processors, whose third generation codenamed Ice Lake was announced in 2020 [96]. Intel KNL is presented as a standalone x86 processor. In contrast, its predecessor, Knights Corner, was a co-processor which required

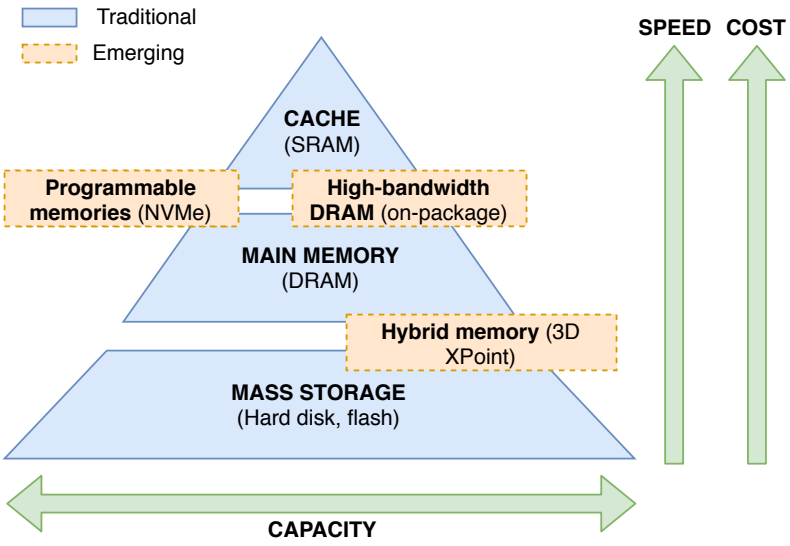


Figure 2.1: Traditional memory hierarchy updated with new emerging technologies.

a general purpose host processor. KNL is therefore not limited by the on-board memory size or the PCIe bus bandwidth. Moreover, the use of the x86 ISA enables KNL to execute different operating systems, legacy libraries, and general purpose applications. As such, KNL is more versatile than its predecessor and than current GPUs, which need applications to be rewritten following specific paradigms such as CUDA [65].

The rest of this section discusses the main characteristics of the Intel KNL: its internal organization, core architecture, on-die interconnect, memory system, and cluster and memory modes.

2.2.1 Internal organization

KNL integrates up to 72 cores organized in a 2D mesh of 38 tiles. Each tile comprises two cores, two vector processing units (VPUs), an L2 cache (shared between cores within the same tile, but private to the rest), and a portion of the distributed directory named Caching/Home Agent (CHA), as shown in Figure 2.2. Each CHA holds a set of cache line addresses, which are distributed among all tiles and their respective CHAs using a built-in address hash. In this way, cache miss requests are

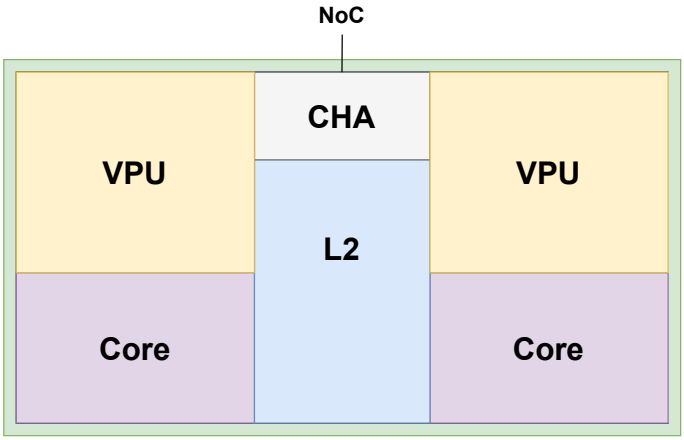


Figure 2.2: High-level tile organization in KNL.

bypassed directly out to the mesh to be serviced by a specific CHA as determined by the address hash. Depending on the particular processor model, the number of tiles enabled varies between 32, 34 and 36, featuring 64, 68 and 72 enabled cores, respectively. When a tile is disabled, the cores and caches within are also deactivated. However, the CHA and the logic for routing within the interconnection network remain active for all tiles, even those disabled. The CHA acts as the connection point between a tile and the mesh . While the CHA box physically resides in the tile, it is logically part of the on-die interconnect mesh.

Intel KNL cores are two-wide out-of-order derived from low-power Silvermont cores, designed for Intel Atom processors but modified to make them suitable for HPC [62]. They have been enhanced with AVX-512-capable VPUs. Furthermore, each core has a private 32-KiB L1 data cache, backed up by a 1-MiB L2 shared with the other core in the same tile but private to it. The CHA manages a portion of the distributed cache directory, which stores the status and location of the most up-to-date copy of a memory line, queried when an L2 miss occurs.

KNL features a 2D mesh NoC, replacing the ring topology used in Knights Corner, as depicted in Figure 2.3, which corresponds to the Intel Mesh Interconnect architecture. Messages traverse the mesh using a simple YX routing protocol: a transaction always travels vertically first, until it hits its target row. Then, it begins traveling horizontally until it reaches its destination. Each vertical hop takes 1 clock cycle, while horizontal hops take 2 cycles. The mesh features 4 parallel networks or

rings, each customized for carrying different types of packets [58]:

- AD ring (address ring): carries tile read/write requests and memory controller snoops to the CHA.
- BL ring (block ring): carries data transfers (two transfers for one cache line).
- AK ring (acknowledge ring): carries acknowledgements from memory controller to CHA and from CHA to tile. It also carries snoop responses from core to CHA.
- IV ring (invalidate ring): carries CHA snoop requests of tile caches (i.e., L2 cache misses).

2.2.2 Memory system

KNL integrates two different types of DRAM memories (see Figure 2.3). Up to 16 GiB of on-package 3D-stacked Multi-Channel DRAM (MCDRAM) provide high-bandwidth accesses through eight independent interfaces. Besides, there are two more DDR interfaces controlling three DRAM channels each, adding up to 192 GiB of memory. A distributed cache coherence mechanism using Intel MESIF [40] is employed. Each time a core requests a memory block that does not reside in the local tile caches, the distributed directory is queried. A message is sent to the appropriate CHA (message (1) in Figure 2.3). If the block already resides in one of the L2 caches in the mesh in Forward state[1], the CHA will forward the request to the owner, which will send the data to the requestor in turn (messages (2) and (3) in the figure). In other cases, the data must be fetched from the appropriate memory interface. The data flow shown in the figure exemplifies one of the performance hazards inherent to the KNL architecture: although the data for the requested block lies in the forwarder tile F, just above the requestor R, the coherence data is stored far away in tile C. As such, 18 cycles are required to transfer the data (10 vertical and 4 horizontal hops). But, if the directory information were stored either in the requestor or in the forwarder, the round trip time of data packets would be of only 2 cycles (2 vertical hops on the mesh).

[1] A cache containing a block in Forward state is in charge of serving said block upon a request. The requestor acquires the block in Forward state, while the sender changes it to Shared.

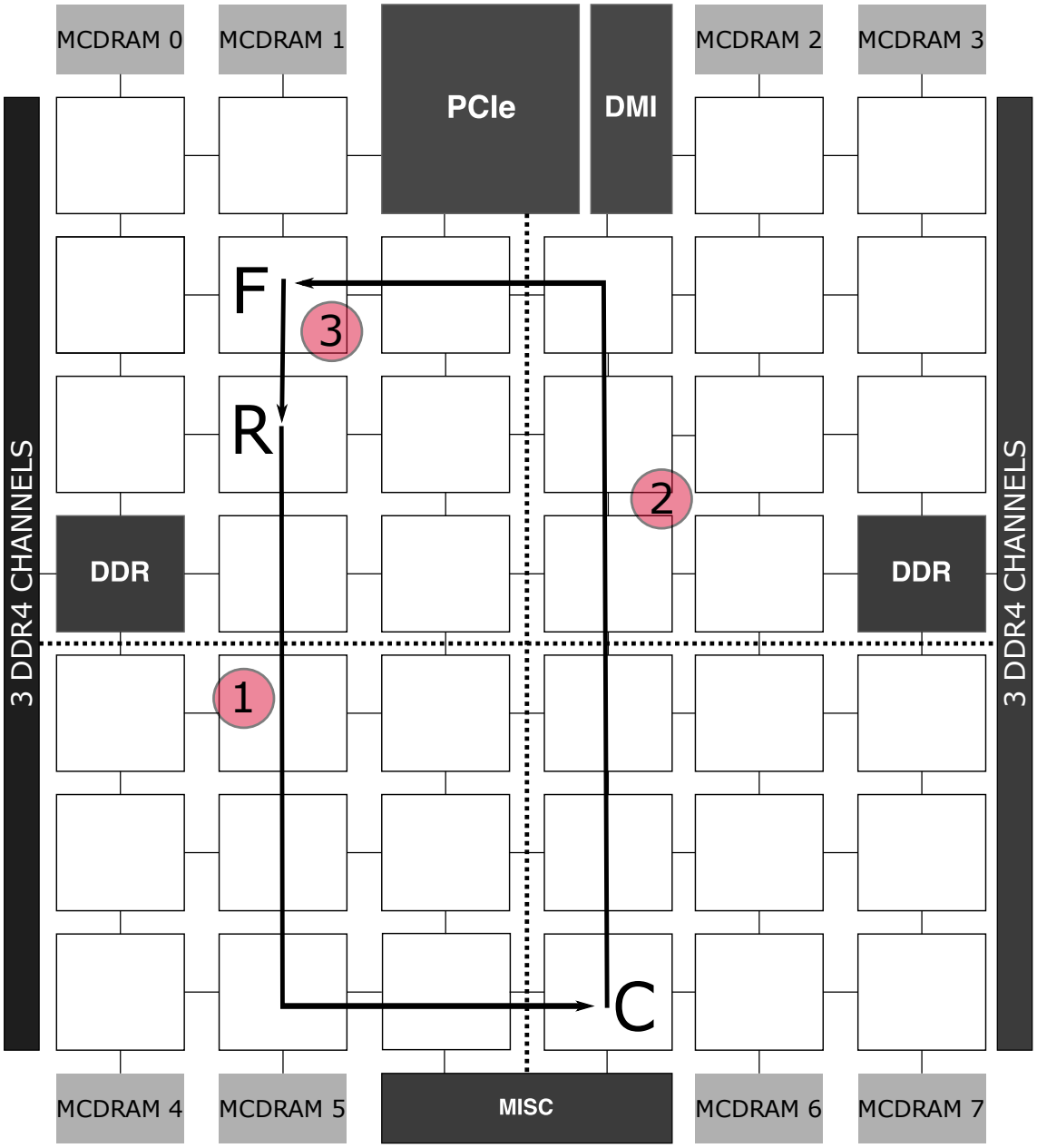


Figure 2.3: Intel KNL floorplan. Each box represents a tile. Cache miss flow for Quadrant cluster mode. R is the requestor, C is the holder, and F is the forwarder.

2.2.3 Cluster modes

The affinity between memory interfaces, CHAs and cores can be configured in KNL through the so-called cluster modes:

- Quadrant: the mesh is virtually divided into four clusters. The memory blocks managed by a CHA are guaranteed to be accessed through a memory interface within the same quadrant. The hash functions which assign memory blocks to CHAs and memory interfaces are not publicly disclosed. This mode is the de-facto standard for KNL operation.
- Sub-NUMA cluster (SNC): all memory is divided into two or four contiguous memory blocks, and each block is assigned to a cluster interleaving cache lines among the memory channels in that cluster. The idea is to create different NUMA nodes isolating traffic within them. It is recommended for MPI and NUMA-aware applications only.
- All-to-all (A2A): data has no affinity at all. This is the most inefficient mode. It should only be used when memory modules are unevenly distributed across memory interfaces.

2.2.4 Memory modes

The MCDRAM memory may be configured into one of two modes: "Flat" memory, in which the address space is explicitly exposed as an independent NUMA domain and is available to the programmer; and "Cache" mode, in which MCDRAM serves as a transparent memory-side cache. In this work we focus exclusively on the MCDRAM subsystem, although all the proposed optimizations are directly extensible to the DRAM subsystem, and on the Quadrant/Flat mode.

2.3 Mapping the Knights Landing Processor

When working in the Sub-NUMA cluster mode the correspondence between logical and physical cores is explicit. Considering 64 cores, four different NUMA mem-

ory domains are created, and each core is associated to one of them: cores {0–15}, {16–31}, {32–47} and {48–63} belong to the four different logical clusters in the mesh. This allows to carefully select the affinity for a team of processes executing an MPI application, knowing that the memory allocated to a processor will be guaranteed to lie in the local interfaces to each cluster, as will the associated directory information. It is not possible to exploit this paradigm using a multithreaded code (e.g., OpenMP) without simulating the distributed memory nature of multiprocess parallelism.

In the default Quadrant mode there is no indication regarding to the neighborhood relationships between different logical core IDs. This makes it impossible to reason about core affinities. Furthermore, even if we discovered core location and bound a team of threads to neighboring cores, each time a cache block is not locally available the requestor will need to query the associated CHA to discover the status and location of the block. This coherence data may reside in any part of the mesh. For this reason, it is not sufficient to know where each core is located in the physical mesh; we would also need to know where each CHA is located in order to carefully plan the memory accesses for each thread, as well as the block-to-CHA mapping.

We reverse engineered the physical layout of an Intel x200 7210 processor in Quadrant mode, with 64 enabled cores, by profiling memory access latencies, building potential layout candidates, and iteratively discarding the ones which present a larger squared error with respect to the observed behavior. For this purpose, we systematically measured the access latency from each logical core ID to cache blocks located in each of the 8 MCDRAM interfaces and each of the 38 CHAs in the mesh. Note that, in Quadrant mode, blocks stored in a given MCDRAM interface can only be indexed by CHAs located in its same quadrant. We created a routine which, given a tuple (C, CH, MC) containing core, CHA and MCDRAM IDs, locates a cache block stored in MC and indexed by CH, and measures the latency of accessing it from C. This routine is detailed in Algorithm 2.1. We first initialize a sufficiently large region of memory (buffer B) to ensure that it will contain instances of all possible (CH, MC) associations. Given that the hash function assigning blocks to CHAs and MCDRAM interfaces is reasonably uniform, this memory does not need to be extremely large (a few 4-KiB memory pages are enough). Then, we test each cache block looking for one which is indexed by CH and stored in MC. To do so,

we access each block and flush it from the cache N times in a loop. After the loop ends, we check which MCDRAM and CHA pair has at least N accesses by using a custom kernel module which leverages the uncore Model Specific Registers (MSR)[2]. After we find a block associated to (CH, MC), we repeatedly access it again, but this time we measure access latencies and compute the average.

In the manner described above, we find the average access latency for all the valid (C, CH, MC) tuples in the mesh. Note that we only need to obtain the latency for one out of each 2 cores, since cores in the same tile share the same CHA, and it is inferrable from `/proc/cpuinfo` that cores $(2x)$ and $(2x+1)$ lie in the same tile. By analyzing the missing (CH, MC) pairs, we discover the association of CHAs and MCDRAMs to quadrants. In particular, we find that data in MCDRAM interfaces $(2y)$ and $(2y+1)$ are indexed by CHAs z such that $(z \mod 4 = y)$, e.g., MCDRAMs 0 and 1 are associated to CHAs 0, 4 ... 36; MCDRAMs 2 and 3 to CHAs 1, 5 ... 37; and so on.

Once these data are collected, we analyze them to determine where each pair (C, CH) of core and CHA is located on the physical mesh, taking into account the public KNL specifications. The floorplan includes 38 physical tiles, some of which have their cores disabled depending on the processor model.[3] Remember that, despite having disabled cores, all tiles have fully functional CHAs and mesh interconnects. The actual location of the tiles with disabled cores changes for each processor unit, depending on process variations [89]. However, the CPUID instruction can be used to discover the actual (C, CH) associations between cores and CHAs. It also provides the list of CHAs which do not have enabled cores. Armed with this information, and with our measured core-to-CHA-to-MCDRAM latencies, we build a squared error model for each candidate assignment of (C, CH) pairs to the physical mesh. In our Intel x200 7210, only 32 tiles have active cores. As such, we have to discover the actual location of these 32 tiles, plus the 6 disabled tiles. Taking into account that we know the associations of (C, CH) pairs and quadrants, as detailed above, there are $10! \times 10! \times 9! \times 9!$ different combinations, as two quadrants have 10 tiles while the remaining two quadrants have only 9 tiles each. This information

[2]We employ the `PERF_EVT_SEL_X_Y` and `ECLK_PMON_CTRX_LOW/HIGH` registers to monitor CHAs and MCDRAMs, respectively [57]. We measure events `RxR_INSERTS.IRQ` and `RPQ.Inserts` [58].

[3]The exact count is 6 tiles with disabled cores in Intel x200 7210 and 7230 series, 4 in the 7250 series, and 2 in the 7290 series.

Algorithm 2.1: Measures latencies for a (C, CH, MC) tuple

```
   Input: Core C, CHA CH and MCDRAM MC
   Output: Average access latency from C to MC via CH
 1 bind_to_core(C);
 2 allocate_buffer(B);
 3 for each cache block b in B do
 4 |   n = 0;
 5 |   start_cha_counters();
 6 |   start_mcdram_counters();
 7 |   while n < N do
 8 |   |   access(b);
 9 |   |   flush_cache_block(b);
10 |   |   n = n + 1;
11 |   end
12 |   stop_mcdram_counters();
13 |   stop_cha_counters();
14 |   mc_b = read_mcdram_counters();
15 |   if mc_b == MC then
16 |   |   cha_b = read_cha_counters();
17 |   |   if cha_b == CH then
18 |   |   |   n = 0;
19 |   |   |   start = get_time();
20 |   |   |   while n < N do
21 |   |   |   |   access(b);
22 |   |   |   |   flush_cache_block(b);
23 |   |   |   |   n = n + 1;
24 |   |   |   end
25 |   |   |   stop = get_time();
26 |   |   |   L = start - stop;
27 |   |   |   return L/N;
28 end
29 return error;
```

allows us to reduce the possible combinations by a factor of 10^{21} with respect to the original 38! possible candidates. To reduce even further the number of possibilities we employ heuristics. First, we locate feasible candidates for the corner tiles, i.e., those contiguous to each MCDRAM interface. For this purpose, we identify the minimum experimental memory latency L (117 cycles in our tests), and search for

(C, CH, MC) tuples with an access latency of at most L plus a configurable error margin. In this way, we reduce the possible combinations for the 8 corners to under 200. Next, for each of these candidates, we build mean squared error models for placing the remaining tiles, and finally accept the one which shows the least squared error.

The obtained results present a clear pattern in the location of both CHAs and cores, as shown in Figure 2.4. The CHAs in each quadrant are sequentially arranged in a vertical fashion. Cores are assigned sequentially to CHAs, skipping those tiles with disabled cores. This technique allows to obtain the physical layout of any individual KNL unit immediately, by just checking which CHAs have disabled cores through CPUID instructions.

2.4 Processor Affinity and Data Layout

Once the mapping of the logical components of the processor onto the physical floorplan is exposed, the next step is to take advantage of this information. There are at least two orthogonal ways in which an application might exploit locality across the mesh:

- Each thread should access data with coherence information stored in a nearby CHA as much as possible. In this way, memory access latency will be improved due to the shorter message trips across the network. Furthermore, restricting coherence data to subsets of the tile will improve the network contention when a large number of cores is active. We refer to this optimization as exploiting core-to-CHA affinity.
- A core requesting data in a nearby L2 cache may not take advantage of this proximity due to the coherence data being assigned to a distant CHA, as illustrated in Figure. 2.3. However, once core-to-CHA affinity is improved, applications will benefit from co-locating cooperating threads. We refer to this optimization as thread-to-core affinity.

Mapping memory blocks and their associated CHAs and using them accordingly requires important changes to the compilation chain and/or the source code of an

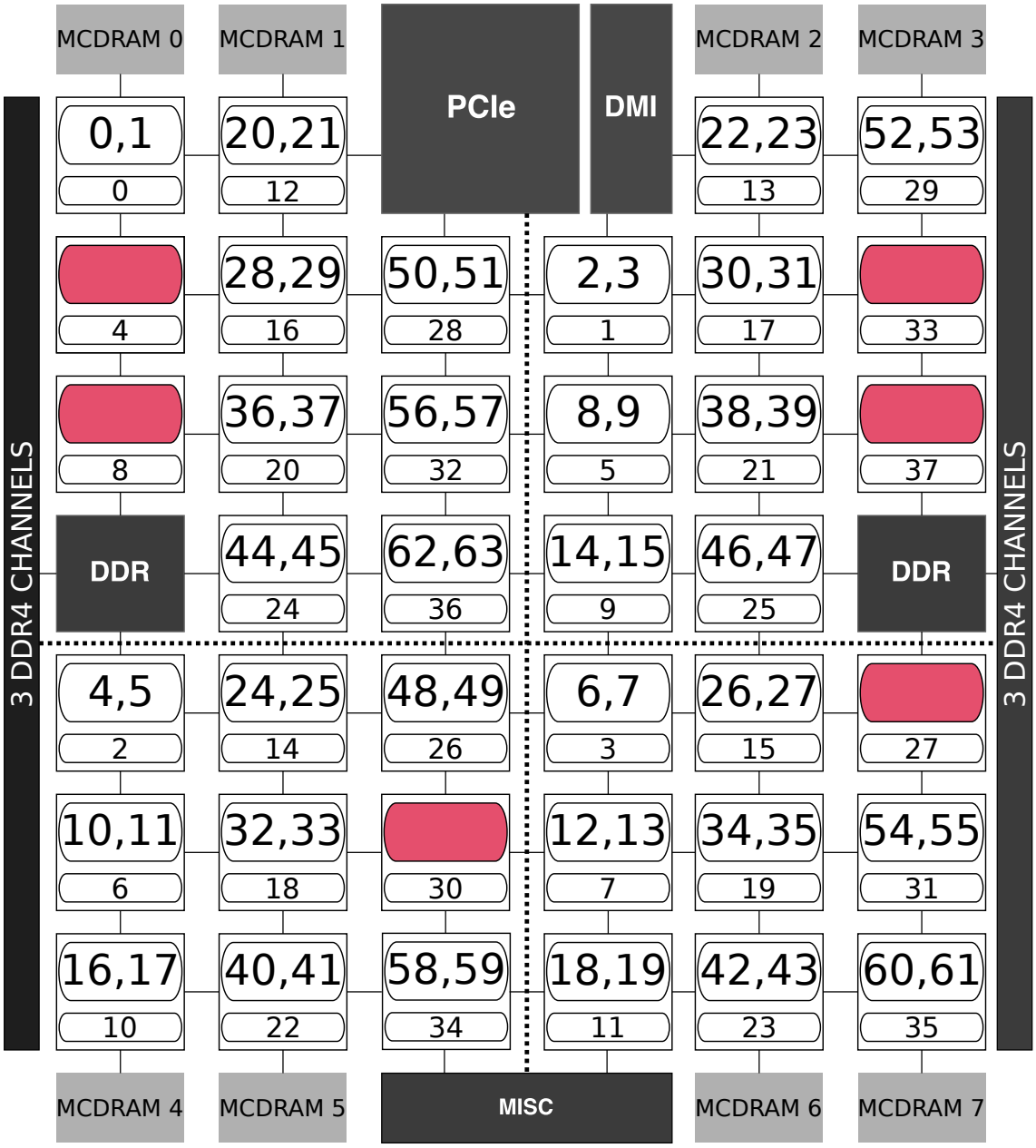


Figure 2.4: Result of our model. Each tile is formed by two cores (their IDs are enclosed in a large box) and a CHA (its ID enclosed in a small box). Tiles with blank boxes indicate that their cores are not active.

application. For any array in a computational kernel, we need to obtain a mapping of the correspondence between the memory blocks in the array and the CHAs in the mesh. Once that is done, work has to be scheduled across the available threads

according to the affinity between the core executing each thread and the CHAs. There are two ways in which this can be accomplished: 1) dynamically, by running an inspector/executor which finds the mapping between memory blocks and CHAs, and schedules tasks accordingly; and 2) statically, by exposing information about the memory system to the compiler.

On a first approach, we are interested only in showing that CHA proximity plays an important role in the performance of multithreaded codes, and showcasing the potential of optimizing mesh locality. For this purpose, we map the CHA locations of all the memory blocks in the 16 GiB MCDRAM memory subsystem. For each block, we note the associated CHA and MCDRAM interface, and store both in one byte. This information takes up 256 MiB for the entire memory, and is incorporated into the runtime of each application. Upon execution, an inspector-executor copies the data to be accessed by each core to memory locations indexed by CHAs with high affinity to that core. This requires using arrays of indirections, and therefore this technique will likely bring performance advantages to irregular codes only. Nevertheless, the aim of this work is to: 1) provide evidence of the impact of the distributed directory; 2) highlight the importance of disclosing architectural features for code optimization; and 3) serve as a basis to develop a compiler-based optimization model. All our experimental codes will use arrays of indirections, even when accessing memory sequentially, in order to fairly assess the impact in memory performance of core-to-CHA and thread-to-core affinity.

2.5 Experimental Results Varying Processor Affinities

We applied the proposed approach to several commonplace computational kernels to analyze how the location of coherence data affects system behavior. The experiments were run on the Intel Xeon Phi x200 7210 mapped in Section 2.3, with 64 cores, 192 GiB of DDR, and 16 GiB of MCDRAM. Codes are compiled using ICC 18.0.3 with `-O2 -xKNL`. The system is configured in Quadrant/Flat mode, dividing the address space into two different regions, one for DDR and one for MCDRAM. The frequency is fixed to the base of 1.30 GHz to avoid thermal variations or CPU throttling (e.g., DVFS). Our applications were configured to use MCDRAM exclusively for data allocation through `numactl`. All our data arrays are allocated into 1

GiB hugepages. We ran three different sets of experiments. First, we analyze the impact of the core-to-CHA affinity optimization using different affinity strengths (Section 2.5.1). Next, we analyze the effect of the thread-to-core affinity optimization in coherence traffic, using a modified 1D stencil (Section 2.5.2). Finally, we broaden our scope to analyze the impact of optimized thread-to-core scheduling of several different workloads (Section 2.5.3).

2.5.1 Effect of core-to-CHA affinity on memory latency

We first measure the potential of optimizing core-to-CHA affinity to reduce memory latency. For this purpose, we employ a scalar vector product kernel, due to its high memory bandwidth requirements. We work with a total dataset of 128 MiB to ensure that no reuse takes place through caches, and repeat the computation 100 times to average out performance differences across the full experiment.

Figure 2.5a illustrates how performance metrics evolve for all possible affinities between cores and CHAs. The core-to-CHA affinity (X axis) indicates the maximum distance in CPU cycles allowed from a core to the CHAs indexing the data it accesses. The figure shows a clear performance improvement from limiting the spread of coherence traffic. The reduction in outstanding weighted cycles is of 7.2%, close to the expected theoretical optimum. However, the derived speedup is only 3.1%. The culprit is the increase in the number of μTLB misses, due to the pseudo-random nature of the access to the data arrays enforced by the search for blocks associated to local CHAs, and to the lack of hugepages support on the L1 TLB. Increasing the neighborhood size reduces μTLB misses, as more memory blocks are usable by each core. This also increases access latency, as more distant MCDRAM interfaces are accessed causing an increase in the travel times of data and coherence packets. The proportion of distant accesses eventually converges to approximately one fourth, as 2 out of the 8 MCDRAM interfaces are considered close to each core.

2.5.2 Effect of thread-to-core affinity on coherence traffic

A second optimization enabled by our architectural analysis is the exploitation of thread-to-core affinity. We aim to improve the locality of the data across the

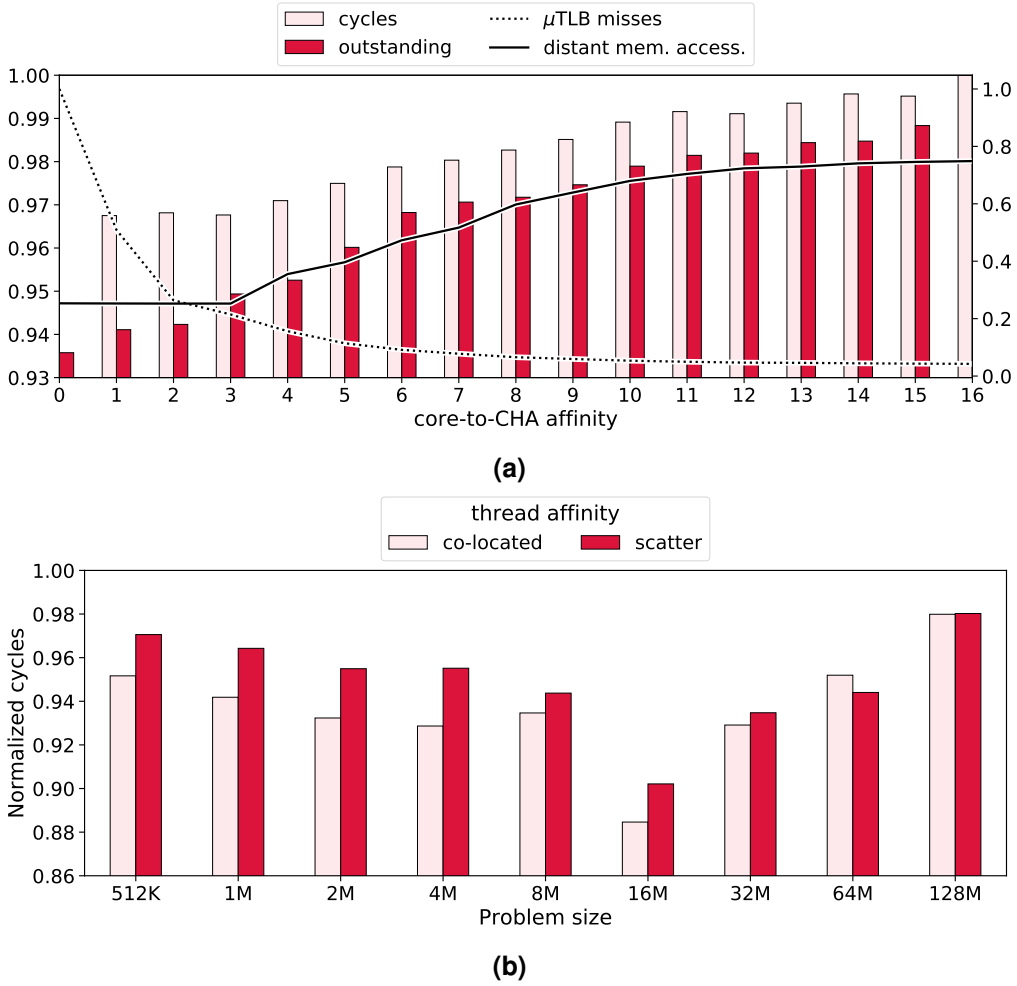


Figure 2.5: Effects of core-to-CHA and thread-to-core affinities: 2.5a) execution cycles, outstanding weighted cycles, μTLB misses, and accesses to distant memory interfaces for different core-to-CHA affinities. Results are normalized to the maximum value for each series, except for accesses to distant memory interfaces, which are normalized to the total number of memory accesses. μTLB misses and distant accesses are referenced to the right axis; and 2.5b) execution cycles of the best-performing core-to-CHA affinity for two different thread-to-core affinities: "scatter" (thread i is assigned to OS core i), and "co-located" (adjacent threads are placed in adjacent physical cores). Results are normalized to the execution cycles of the non-optimized code with scatter thread placement. The left Y axes are truncated to better reflect the differences in values.

L2 caches in the mesh to reduce coherence traffic. For this purpose, we modify a `jacobi-1d` stencil so that neighboring cores swap their data at the end of each timestep. Note that it is futile to try to exploit thread-to-core affinity without enforcing the core-to-CHA affinity first, as a thread sharing a block of data will need to traverse the mesh to query the appropriate CHA before finding out that the block lay in a neighboring core.

The 1D stencil was run on 64 cores using 108 different configurations, including several problem sizes, five different core-to-CHA affinities, and two thread-to-core affinities: scatter and an ad hoc "co-located" affinity in which adjacent threads are assigned to adjacent cores whenever possible. In total, more than 3,000 executions of the stencil were run. Fig. 2.5b shows the normalized median execution cycles after discarding outliers. As can be observed, the two optimizations target different types of workloads. For applications with small datasets, in which reuse comes from other cores in the mesh, adjusting thread-to-core affinity yields important benefits. The figure clearly shows how this optimization loses effectivity when the memory footprint reaches the total combined cache size (32 MiB in our Intel x200 7210 processor). On the other hand, applications with large datasets, which consume a large volume of data directly from memory, benefit more from the memory latency reduction provided by core-to-CHA affinity. This optimization also loses effectivity as the footprint increases. In this case, the reason is the exponentially increasing number of μTLB misses, as covered in Section 2.5.1.

2.5.3 Optimized thread-to-core scheduling

In the previous section we studied how changing the thread-to-core affinity impacted performance for a particular workload. The data was always shared among consecutive threads, and therefore a simple affinity could be devised ensuring that threads sharing data were never more than two hops apart on the mesh. However, designing balanced affinities for more complex data sharing relationships in 64-threaded applications is a non-trivial problem, particularly given the irregular structure of the mesh. Instead, we focus on how to optimally schedule smaller workloads on the available cores.

In order to characterize the mesh behavior we executed different applications

Table 2.1: Benchmarks used in the experiments, characterized by the weighted averages of cache accesses and misses, memory accesses, and floating-point operations. Values are reported in millions per thread per second.

Benchmark	Description	D1 acc.	D1 misses	L2 misses	MCDRAM	FLOPS
rvec	Vector reduction	14.88	10.24	8.42	8.34	190.25
rvv	Vector-vector add & reduction	19.68	12.59	11.08	10.95	257.05
vecsearch	Search for value in vector	11.42	8.96	8.39	8.32	284.88
jac-2d	2D Jacobi stencil	249.98	12.26	9.58	9.38	334.62
avv	Vector-vector addition	124.45	9.66	11.40	11.29	635.02
jac-1d	1D Jacobi stencil	52.98	11.02	15.22	15.10	702.72
jac-1d-swap	1D Jacobi data swap after tstep	59.79	11.50	14.77	7.37	717.60

with footprints ranging from 512 KiB to 2 MiB per thread (4 times the allotted cache space per core), running on 4, 8, 16 and 32 threads. We tested a total of 113 thread-to-core schedules, including 4-, 8-, 16- and 32-thread groups. For instance, in the case of 4 threads, we tested the full set of 46 different contiguous 2-tile allocations, plus the default "scatter" affinity in ICC, plus several random ones to act as control groups. Our benchmarks are the stencils and array kernels detailed in Table 2.1. In total, more than 45,000 executions were performed. The results were analyzed using k-means clustering to discover the factors that impact performance. We gather information about architectural trends, particularized for our processor unit: which cores are faster or slower, how the distance to the MCDRAM interfaces affects benchmarks depending on bandwidth requirements, which types of benchmarks benefit from core-to-CHA and thread-to-core affinity optimizations, etc.

We validate the collected historical data by generating random mixes of applications and executing them using a schedule which exploits the architectural characteristics discovered during the analysis phase. We randomly generate 6 different workload mixes, each including 8 benchmarks of varying sizes, as detailed in Table 2.2. We then execute each mix using three different configurations: 1) "fair": cores are assigned to each task proportionally to the size of their dataset, no core-to-CHA-affinity is enforced, and the binding of threads to cores is managed by the OS; 2) "load": cores are distributed across the tasks by using the historical data to estimate expected execution times, no core-to-CHA affinity is enforced, and the binding of threads to cores is managed by the OS; and 3) "optimized": the number of cores per task is the same as in 2), but strong core-to-CHA affinity is enforced,

Table 2.2: Applications in each mix of workloads.

mix	apps
#1	jac-2d-512KiB, rvv-512KiB, jac-1d-8MiB, jac-2d-8MiB, rvec-8MiB, rvv-32MiB, vecsearch-32MiB, avv-128MiB
#2	avv-512KiB, rvv-1MiB, vecsearch-1MiB, avv-2MiB, jac-1d-swap-4MiB, jac-1d-swap-32MiB, jac-2d-128MiB
#3	vecsearch-512KiB, jac-2d-1MiB, vecsearch-1MiB, jac-2d-2MiB, avv-2MiB, vecsearch-4MiB, rvec-8MiB, jac-2d-16MiB
#4	rvec-512KiB, rvv-512KiB, vecsearch-512KiB, avv-32MiB, jac-1d-4MiB, vecsearch-4MiB, avv-32MiB, jac-2d-32MiB
#5	jac-1d-swap-1MiB, jac-2D-1MiB, rvec-2MiB, vecsearch-4MiB, avv-16MiB, jac-1d-swap-16MiB, jac-2d-16MiB, jac-2d-32MiB
#6	jac-1d-swap-512KiB, avv-1MiB, avv-2MiB, rvv-2MiB, rvv-8MiB, avv-32MiB, rvv-32MiB, jac-2d-64MiB

and an optimal thread-to-core binding is computed by consulting historical data.

Figure 2.6a shows the performance of our scheduling strategies. The improvement obtained by the "load" scheduling depends on how well the computational load of each mix is predicted by the footprint of its applications. For example, in mix #1 the initial "load" scheduling does not improve the execution time, as our resource allocation binds threads of very small benchmarks to different hyperthreads of the same core, to better exploit the available slack in the mix. The "optimized" scheduling takes into account the characteristics of the mesh to achieve further improvements. This effect is most noticeable in mix #5, in which the longest computation corresponds to `jac-1d-swap-16MiB`, a benchmark which is particularly sensible to the co-location of its computing threads. Aggregating all mixes, "load" scheduling improves total execution times by 20.8%, and "optimized" increases that gain to 61.3%.

Figure 2.6b provides a more detailed view of different performance metrics for the "optimized" schedules. The plot aggregates the sum of all metrics for all tasks in each mix. Note that sometimes the total number of execution cycles increases with respect to the original execution cycles. Yet, as shown in Fig. 2.6a, the total execution time always improves. The reason is that the "optimized" scheduling exploits the slacks of non-critical path tasks to better balance resource allocation. Another interesting effect is the total increase in the number of MCDRAM accesses

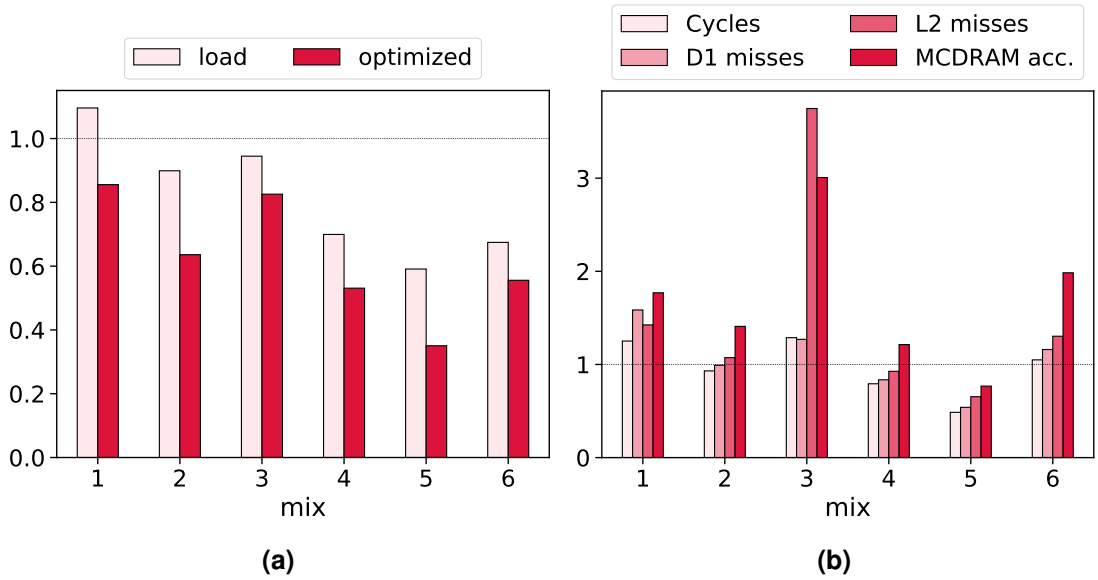


Figure 2.6: Results of scheduling strategies: 2.6a) execution cycles of "load" and "optimized" schedules normalized to "fair" values; and 2.6b) performance counters of "optimized" schedules aggregated over the sum of each mix and normalized to the values in "fair" schedules.

for almost all mixes. This is caused by a benchmark with large dataset but short comparative execution time, being allocated a reduced set of resources. This causes the benchmark to become memory-bound, and its execution time to increase, but keeping it out of the critical path of the mix. To avoid interference, these tasks are allocated a set of MCDRAM interfaces which are not used by other high-bandwidth demanding tasks. On the other side of the spectrum is mix #5, where the number of MCDRAM accesses is greatly reduced by the co-location of the executing threads and the increase in allocated cache resources.

2.6 Reverse Engineering the CHA Mapping

As we described in Section 2.2, KNL employs a built-in address hash to distribute all cache lines among CHAs in the mesh. In Section 2.3 we have already described how to obtain empirically these correspondences block-to-CHA. Leveraging this model, in this section we focus on building a closed-form function of the

mapping of cache lines or memory blocks to CHAs in order to enable new optimization strategies for these mesh interconnect-based processors.

In hardware designs, pseudo-random mappings often make use of XOR gates, such as with Cyclic Redundancy Codes (CRCs), Linear Feedback Shift Registers, and other XOR hashes [69]. XOR mappings can be efficiently implemented in gates relative to other forms of pseudo-random mapping binary addresses, such as modulo arithmetic of the form $x = (n_1\text{addr} + n_2) \bmod n_3$. Using the data obtained in Section 2.3, Table 2.3 shows the CHA mapping for the first 128 cache lines out of the 256 million mapped locations, i.e., the entire MCDRAM address space.

This section describes the analysis of this mapping data in order to generate the closed forms of the mapping functions. Since full 64-byte cache lines are stored when a CHA location is determined for the data, the address-to-CHA mapping does not make use of address bits 5:0. In a first, coarse-grained analysis of the data, we find that the CHA mapping depends only on address bits $A_{34:6}$, which allows for $2^{536,870,912}$ distinct binary functions for each of the 6 CHA bits.

Table 2.3: Address-to-CHA mapping for the first 128 CHA values out of 256 million. To aid in visualizing, CHA_0 and CHA_1 are shown **in a box** and CHA_{37} is shown in white over black.

	Address bits 9:6															
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Address	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
bits	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
12:10	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
000	26	9	24	11	21	6	23	4	31	12	29	14	16	3	18	**1**
001	27	8	25	10	20	7	22	5	14	**37**	28	15	33	34	19	**0**
010	10	25	8	27	5	22	7	20	15	36	13	30	32	35	2	17
011	11	24	9	26	4	23	6	21	30	**37**	12	31	33	34	3	16
100	12	31	14	29	3	16	**1**	18	9	26	11	24	6	21	4	23
101	13	30	15	28	2	17	**0**	19	8	27	34	33	7	20	**37**	6
110	28	15	30	13	19	**0**	17	2	25	10	35	32	22	5	36	7
111	29	14	31	12	18	**1**	16	3	24	11	34	33	23	4	**37**	22

Given values for CHA from 0 to 37, 6 bits are needed to represent this number, but given that 38 is not a power of 2, we did not expect to see a straightforward XOR equation of address bits for each CHA bit. However, given the ease of computing binary functions in hardware, we did expect and found that each bit for

the CHA value can be computed independently (again, as opposed to a scheme like *addr* mod 38).

The process we follow to determine the equations for CHA bits is shown in Figure 2.7. Our final process finds equations of the form $\text{CHA}_n = f\overline{g}|h$ where most of the bits are correctly predicted by the f function alone, $\overline{g}$ masks some bits to 0, and h is OR'ed in to correct some bits to 1. Following the idea that the function should be easily implementable in hardware, we first attempt to find $f = a_1 \oplus a_2 \oplus ... \oplus a_{n-1} \oplus z(a_n, a_{n+1}, ...)$; that is, f is a function which XORs certain address bits together with a binary function z which uses a small number of identifiable address bits.

For instance, consider the toggle frequency for CHA_0 when different bits of the

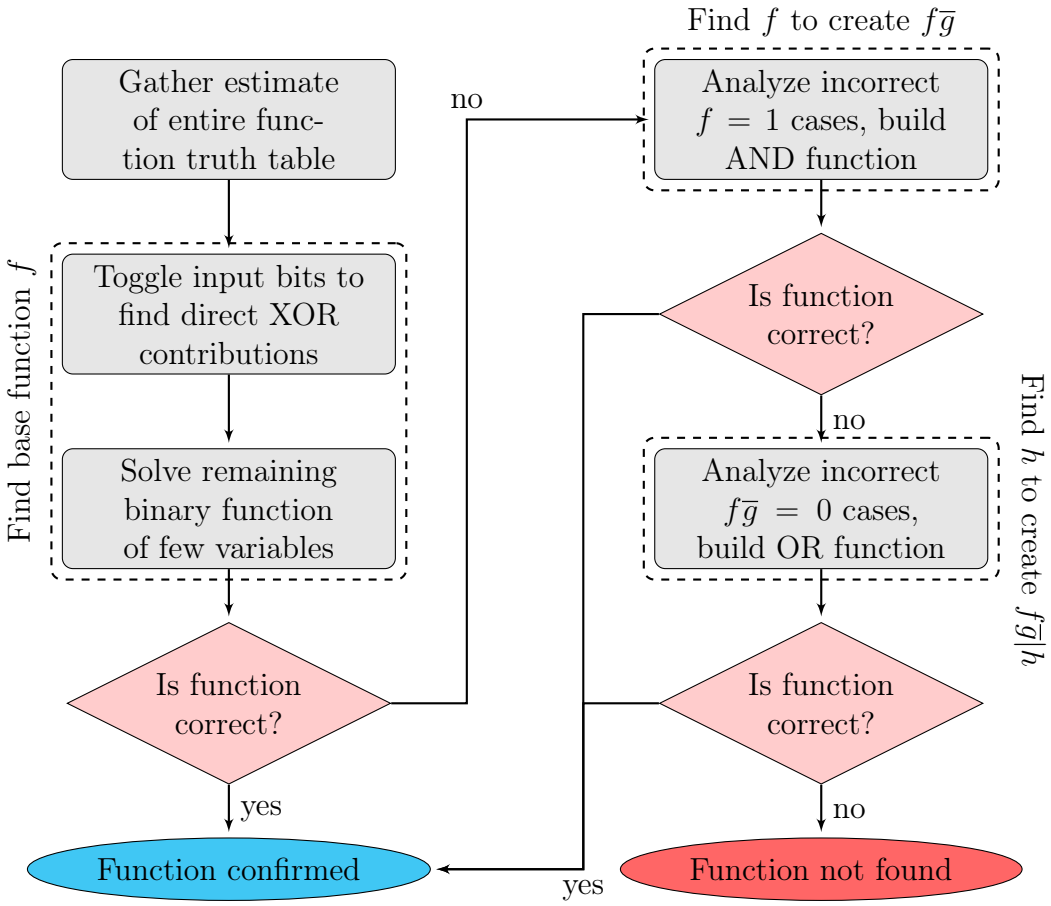


Figure 2.7: Reverse engineering hardware-friendly hash functions.

address are toggled shown in Table 2.4. As can be seen, 99.93% of the time toggling a_6 or a_8 changes the result of CHA_0, whereas toggling a_7 almost never affects its value. It can be concluded that the mapping function for bit 0 must be of the form $\text{CHA}_0 = a_6 \oplus a_8 \oplus ...$, where "..." is yet to be determined. The fact that the data is not 100% precise is attributable to measurement errors in the performance counter-based mapping process.

The analysis of the toggle frequency finds that some of the bits $A_{29:6}$ are directly XOR'ed into CHA_0, while some others do not appear at all. However, the study also shows that bits $A_{33:30}$ affect the function, but not in the same categorical way. The toggle frequency is somewhere between 5% and 95%. In order to reverse engineer the role of these bits in the function, the limited input binary function of $A_{33:30}$ is analyzed to detect which combinations of these bits toggle the result of the partial XOR function built from $A_{29:6}$, as shown in Table 2.5. This reverse engineering process yields the functions CHA_0 and CHA_1 in Figure 2.8 for the 2 least significant bits of the CHA.

Although the number of CHA locations (38) is not divisible by 4, we found that CHA_{0-1} are each on for 50% of the addresses, and as seen in Figure 2.4 this distributes data evenly among the 4 quadrants of the die. $\text{CHA}_1 = 1$ indicates the data is in the lower half of the die; $\text{CHA}_0 = 1$ indicates the data is on the right side of the die. Given that the lower quadrants have one fewer CHA as compared to upper quadrants, this will cause an imbalance of up to 20% in the number of memory blocks mapped to different CHAs, as described in more detail in Section 2.7.

The functions for CHA_{2-5} are more complex than those for CHA_{0-1}, in order to reasonably distribute data among the 38 CHA values. As shown in Figure 2.7, finding the base function f, which matches most binary function values, sometimes does not fully match the measured function values. In such cases, we search for $\overline{g}$ functions to logically AND with f to set certain values to 0, and h functions to logically OR with $f\overline{g}$ to set certain values to 1. As an example, we will describe the process of determining the g function for CHA_2 with reference to Table 2.6. The f function includes $a_8 \oplus a_9 \oplus a_{12}$ which results in regular blocks of 1's and 0's when the address bits 13:6 are varied with a total of 128 1's and 128 0's in the set of 256 cache lines. However, the performance counter data implies that 6 of those 1's are actually 0's. Note that in the binary representation of 0 through 37, bit 2 is high

Table 2.4: CHA_0 toggle frequency when toggling address bits a_6 to a_{34}. In this case, values greater than 0.98 or less than 0.02 indicate errors in the CHA predicted based on performance counters, and are interpreted as 1 and 0, respectively.

Address Bit	CHA_0 Toggles	Equation Role	Address Bit	CHA_0 Toggles	Equation Role
a_6	0.9993	XOR	a_{21}	0.0009	Ignore
a_7	0.0001	Ignore	a_{22}	0.0011	Ignore
a_8	0.9993	XOR	a_{23}	0.9989	XOR
a_9	0.9994	XOR	a_{24}	0.0014	Ignore
a_{10}	0.9994	XOR	a_{25}	0.0014	Ignore
a_{11}	0.0002	Ignore	a_{26}	0.0013	Ignore
a_{12}	0.0002	Ignore	a_{27}	0.9954	XOR
a_{13}	0.0002	Ignore	a_{28}	0.0045	Ignore
a_{14}	0.9994	XOR	a_{29}	0.0120	Ignore
a_{15}	0.9994	XOR	a_{30}	0.9458	Function
a_{16}	0.0004	Ignore	a_{31}	0.9444	Function
a_{17}	0.9993	XOR	a_{32}	0.0555	Function
a_{18}	0.9993	XOR	a_{33}	0.0546	Function
a_{19}	0.0006	Ignore	a_{34}	0.0000	Ignore
a_{20}	0.9993	XOR			

Table 2.5: For CHA_0, bits 33 to 30 do not directly get XOR'ed with other bits, but are part of a function that itself is XOR'ed with those bits.

Address Bits 33:30	Avg CHA_0 when direct XOR low	Address Bits 33:30	Avg CHA_0 when direct XOR low
0000	0.0019	1000	1.0
0001	0.0	1001	0.0014
0010	0.0	1010	0.0007
0011	1.0	1011	1.0
0100	1.0	1100	1.0
0101	0.0	1101	0.0
0110	0.0	1110	0.0005
0111	1.0	1111	0.9991

$\text{CHA}_0 = a_6 \oplus a_8 \oplus a_9 \oplus a_{10} \oplus a_{14} \oplus a_{15} \oplus a_{17} \oplus a_{18} \oplus a_{20} \oplus a_{23} \oplus a_{27} \oplus ((a_{30}a_{31})|(\overline{a_{30}a_{31}}(a_{32}|a_{33})))$

$\text{CHA}_1 = a_6 \oplus a_7 \oplus a_8 \oplus a_{12} \oplus a_{16} \oplus a_{17} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus a_{30} \oplus a_{33}$

$\text{CHA}_2 = f\overline{g}$, where:

$f = a_8 \oplus a_9 \oplus a_{12} \oplus a_{15} \oplus a_{16} \oplus a_{18} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus \overline{a_{30}}(a_{31}|a_{32}|a_{33}))$

$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus$
$a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(a_6 \oplus a_{12} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{28} \oplus$
$a_{32} \oplus a_{34})(a_7 \oplus a_{12} \oplus a_{14} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus$
$a_{32} \oplus a_{34})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus$
$a_{29} \oplus a_{33} \oplus a_{34})(a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{24} \oplus a_{26} \oplus a_{28} \oplus a_{29} \oplus a_{31} \oplus a_{33} \oplus a_{34})$

$\text{CHA}_3 = f\overline{g}|h$, where:

$f = a_8 \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{28} \oplus (a_{30}|a_{31}|a_{32})(a_{32} \oplus \overline{a_{33}})$

$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus$
$a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus$
$a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus$
$a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$

$h = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus$
$a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_6} \oplus a_{13} \oplus a_{15} \oplus a_{16} \oplus a_{17} \oplus a_{18} \oplus$
$a_{20} \oplus a_{21} \oplus a_{26} \oplus a_{29} \oplus a_{34})(\overline{a_7} \oplus a_{13} \oplus a_{14} \oplus a_{15} \oplus a_{16} \oplus a_{19} \oplus a_{25} \oplus a_{27} \oplus a_{34})$
$(\overline{a_8} \oplus a_{15} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus \overline{a_{30}} \oplus a_{31} \oplus a_{32} \oplus a_{34})(a_9 \oplus$
$a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})(\overline{a_{12}} \oplus a_{14} \oplus a_{15} \oplus$
$a_{16} \oplus a_{17} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{31} \oplus a_{32} \oplus a_{33} \oplus a_{34})$

$\text{CHA}_4 = f\overline{g}|gh$, where:

$f = a_6 \oplus a_{11} \oplus a_{12} \oplus a_{16} \oplus a_{18} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{30} \oplus a_{31} \oplus a_{32}$

$g = ((a_{11} \oplus a_{16} \oplus a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31})|(a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus$
$a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus$
$a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus$
$a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$

$h = (\overline{a_{10}} \oplus a_{11} \oplus a_{13} \oplus a_{16} \oplus a_{17} \oplus a_{18} \oplus a_{19} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus$
$a_{31} \oplus a_{33} \oplus a_{34})(\overline{a_6} \oplus a_{12} \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus$
$a_{29} \oplus a_{31} \oplus a_{32} \oplus a_{33})(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus$
$a_{31} \oplus a_{32} \oplus a_{33})(\overline{a_8} \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus$
$\overline{a_{30}} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$

$\text{CHA}_5 = f\overline{g}$, where:

$f = ((a_{10} \oplus a_{15} \oplus a_{16} \oplus a_{20} \oplus a_{22} \oplus a_{25} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus \overline{a_{30}} \oplus a_{34})|(a_{11} \oplus a_{16} \oplus$
$a_{17} \oplus a_{21} \oplus a_{23} \oplus a_{26} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus a_{31}))(\overline{a_7} \oplus a_{12} \oplus a_{13} \oplus a_{17} \oplus a_{19} \oplus a_{22} \oplus$
$a_{23} \oplus a_{24} \oplus a_{25} \oplus a_{27} \oplus a_{31} \oplus a_{32} \oplus a_{33})(a_9 \oplus a_{14} \oplus a_{15} \oplus a_{19} \oplus a_{21} \oplus a_{24} \oplus a_{25} \oplus a_{26} \oplus$
$a_{27} \oplus a_{29} \oplus a_{33} \oplus a_{34})$

$g = (\overline{a_6} \oplus a_{12} \oplus a_{13} \oplus a_{14} \oplus a_{18} \oplus a_{20} \oplus a_{21} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{26} \oplus a_{29} \oplus a_{31} \oplus$
$a_{32} \oplus a_{33})(\overline{a_8} \oplus a_{12} \oplus a_{14} \oplus a_{16} \oplus a_{18} \oplus a_{19} \oplus a_{22} \oplus a_{23} \oplus a_{24} \oplus a_{27} \oplus a_{28} \oplus a_{29} \oplus$
$\overline{a_{30}} \oplus a_{33})$

Figure 2.8: Reverse-engineered mapping function between memory blocks and CHAs.

18/38 = 47% of the time, hence the simple 50/50 XOR equation from f needs to be masked to 0 in some pseudo-random locations resulting on $\text{CHA}_2 = f\overline{g}$. Given where the masking occurs in Table 2.6, we surmise the structure of the g function to be $((a_{11} \oplus ...)|(a_{10} \oplus ...))(a_6 \oplus ...)(a_7 \oplus ...)(a_9 \oplus ...)(...)$ where "..." represents unknown functions of higher order bits. By comparing $f\overline{g}$ to the known CHA locations with partially completed g functions, we build up the complete mask functions detailed in Figure 2.8.

Like CHA_2, CHA_5 uses a base f function and a mask-to-0 g function. Bits CHA_3 and CHA_4 had a base f function, mask-to-0 g function, and mask-to-1 h function. The h function was found by recognizing where the $f\overline{g}$ pattern itself was not producing correct predictions. The process of determining the f function by observing XOR toggle indications and solving the 4- or 5-bit binary function remaining can be automated. In theory, given a rough constraint on the types of

Table 2.6: CHA_2 for the first 256 cache lines. Given a base function which includes $a_8 \oplus a_9 \oplus a_{12}$, the 6 boxed **0** positions show where a masking function is used.

	Address bits 9:6															
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Address	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
bits	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
13:10	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0000	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0001	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0010	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0011	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0101	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0110	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
0111	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1000	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
1001	0	0	0	0	1	1	1	1	1	1	1	**0**	0	0	0	0
1010	0	0	0	0	1	1	1	1	1	1	1	**0**	0	0	0	0
1011	0	0	0	0	1	1	1	1	1	1	1	**0**	0	0	0	0
1100	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1
1101	1	1	1	1	0	0	0	0	0	0	0	0	**0**	1	1	1
1110	1	1	1	1	0	0	0	0	0	0	0	0	**0**	1	1	1
1111	1	1	1	1	0	0	0	0	0	0	0	0	**0**	1	1	1

functions to be considered, the process of finding the g and h functions could also be semi-automated by searching for mispredictions of the f function to the true result. Future architectures may vary the mapping structure, but the general approach used here to reverse engineer XOR trees would be applicable to them. However, involving a human to interpret binary results and build the final equations may remain common for this type of task. Note that this approach allows for discovery of partial equations without necessarily solving the full binary function.

After generating the closed forms, we find a small proportion of 0.03% discrepancies when comparing the original mapping data to the CHAs predicted by the generated closed-form functions. We validated the closed forms by re-running the mapping micro-benchmarks for these divergent blocks, this time finding 100% agreement with the generated closed forms. Consequently, we attribute the discrepancy in the original mapping to transient and infrequent measurement errors in the Model Specific Registers (MSRs).

2.7 Runtime Optimization

The inspector-executor scheduling presented in Section 2.5 is limited to irregular codes. Transforming the data layout so that the data to be accessed by each tile lie in memory blocks for which the coherence information was assigned to nearby CHAs has an important overhead during the inspection phase. First, the input data need to be physically copied to target memory blocks with the required coherence properties. Then, the associated indirection arrays need to be recomputed. Lastly, the resulting data are now spread across a much larger region of memory, in order to find suitable memory blocks, and therefore cache locality is degraded and the number of page faults increased. With the closed form of the mapping functions exposed in Section 2.6, it is possible to apply this approach to general codes, instead of being restricted to irregular computations. The basic idea is to encode the schedule of tasks not on the indirection arrays, but to exploit the properties of the mapping function.

Consider the general matrix-vector multiplication code depicted in Listing 2.1. This is an interesting problem because of its simplicity, its transversality, and be-

```
1 #pragma omp parallel for
2 for (int i = 0; i < N; ++i)
3     for (int j = 0; j < N; ++j)
4         y[i] += B[i * N + j] * x[j];
```

Listing 2.1: Scalar code for general matrix-vector multiplication parallelized using a static block schedule.

cause of the fact that it is memory-bound in modern processors. As such, it will benefit from increasing the memory throughput. The dominant part of the memory footprint of the computation is the access to matrix B, and therefore the following analysis will be centered on trying to optimize its access.

Given the complexity of the mapping functions, it is implausible to dynamically perform a very fine-grained scheduling of iterations to tiles that will actually have the required coherence information in its local CHA. Besides, this would imbalance the computation, as our mapping data shows that some CHAs manage up to 20% more memory blocks than others. This is a consequence of two different factors. First, the upper quadrants have 10 CHAs each, whereas the lower quadrants have only 9 CHAs. That will create some imbalance, given that the memory distribution over quadrants is balanced, i.e., each quadrant manages exactly 4 GiB of memory. But furthermore, distributing a power of 2 number of memory blocks over a non-power of 2 number of CHAs creates an additional imbalance. The actual distribution of memory to CHAs is as follows:

- Upper quadrants have 10 CHAs, 8 of them manage 416 MiB each, while the remaining 2 (those with the highest IDs in each quadrant) manage 384 MiB each.
- Lower quadrants have 9 CHAs, 8 of them manage 464 MiB each, while the remaining one (that with the highest ID in each quadrant) manages 384 MiB.

Note that this does not vary across different Xeon Phi x200 models, as all units have 38 enabled CHAs, independently of the number of active cores.

In order to alleviate this imbalance we focus instead on the quadrant granularity, emulating the behavior of the sub-NUMA modes of the machine by ensuring that

each tile computes data with coherence information resident on its quadrant only. In this fashion, each quadrant manages exactly 4 GiB of memory. The approach followed for scheduling iterations in this fashion is described in the following.

The quadrant mapping benefits from a convenient feature of the address-to-CHA functions. As noted in Section 2.6, and due to the physical placement of logical CHAs on the NoC shown in Figure 2.4, bits $\text{CHA}_0 = c_0$ and $\text{CHA}_1 = c_1$ identify the quadrant c_1c_0 in which the CHA is located. Consider the kth memory block with address A^k aligned to a 256-byte boundary, i.e., k is a multiple of 4. Bits $A^k_{5:0}$ express an offset inside the memory block, and therefore are not used in the computation of the associated CHA. Because of the 256-byte alignment, $A^k_{7:6} = 00_b$. The address of the next memory block, $A^{k+1} = A^k + 64$, will share its most significant bits with A^k, i.e., $A^{k+1}_{63:8} = A^k_{63:8}$, and $A^{k+1}_{7:6} = 01_b$. Since A_6 participates in the XOR computation in the equations for CHA_1 and CHA_0 in Figure 2.8, it can be determined that the least significant bits of its associated CHA will be flipped, i.e., if the associated quadrant for A^k is c_1c_0 then the associated quadrant for A^{k+1} will be $\overline{c_1}\,\overline{c_0}$. Similarly, $A^{k+2}_{7:6} = 10_b$ and its associated quadrant will be $\overline{c_1}c_0$; and $A^{k+3}_{7:6} = 11_b$ and its associated quadrant is $c_0\overline{c_1}$. This results in the convenient organization that precisely 1 out of every 4 cache lines is in each physical quadrant, allowing parallel access routines to evenly divide up work among physical processors.

In the proposed sub-NUMA schedule a processor located in quadrant c_1c_0 will process only memory blocks with associated CHA in the same quadrant. After processing a block at address A, the next address in the same quadrant could be located at $A+100_b$, $A+101_b$, $A+110_b$, or $A+111_b$ depending on $A_{33:8}$. Determining which of the 4 addresses is next in our quadrant mathematically requires to compute the full CHA equations discovered in Section 2.6. However, these are complex so these computations should be performed as little as possible. The actual offset required to compute the next address in quadrant c_1c_0 has a fixed pattern for address bits $A_{12:8}$, which allows a 64-bit register to store the offsets for the next 32 cache lines. In this way, processors stepping through memory can thus avoid full computation of the mapping function 31 out of each 32 iterations.

2.7.1 Experimental results

In order to have full control over the executed instructions, the original code from Listing 2.1 is manually vectorized using AVX-512 Intrinsics as shown in Listing 2.2. In this way, opaque optimizations that may bias the comparison of different schedules are avoided. This section focuses on single-precision floating-point arithmetic only, but all obtained results are directly extrapolable to double precision.

Both the code in Listing 2.2 and the equivalent sub-NUMA schedule are executed on an Intel x200 7210 running at the base frequency of 1.30 GHz, to avoid turbo-related variations. The codes were compiled using ICC 19.1.1.217, with flags `-Ofast -xKNL -qopenmp`. They are executed on 64 threads using `KMP_AFFINITY=scatter`. Heap variables are stored into 1 GiB hugepages via `hugectl -heap`, and these hugepages are guaranteed to be allocated in the MCDRAM address space using `numatcl -m 1`. The experiments are run with $N = 16384$, which makes matrix B take up 1 GiB of memory, that is, an entire hugepage.

The roofline model generated by Intel Advisor [95] for these codes is shown in Figure 2.9. For these experiments, the hardware prefetcher was manually turned off using Model Specific Registers (MSR) [93] in order to observe the raw effect of the proposed coherence traffic optimizations without interference. As shown in the figure, the sequential schedule achieves 50.7 GFLOPS for an arithmetic intensity (AI) of 0.25, which is approximately 65% of the roofline for that AI, whereas the sub-NUMA schedule achieves 54.7 GFLOPS for an AI of 0.22, or 81% of the roofline. The GFLOPS have increased and the AI has decreased, due to the additional memory traffic required to compute the sub-NUMA schedule, resulting in a large net increase of the percentage of peak performance that is obtained. The figure shows how executions with double-precision arithmetic achieve the same approximate results, but dividing the number of raw GFLOPS by 2.

The improvement in raw performance measured by the roofline model, however, can be deceitful. Although the sub-NUMA schedule achieves a higher FLOP count, it also executes additional instructions on non-consecutive memory blocks, causing a degradation in cache behavior and ultimately execution time. In order to more closely investigate the effect of the proposed optimization, selected performance counters were measured for several different execution setups. The results

```
1  #pragma omp parallel for
2  for (int i = 0; i < N; ++i) {
3      __m512 bb, bx, accum;
4      accum0 = _mm512_setzero_ps();
5      for (int j = 0; j < N; j += 16) {
6          bb    = _mm512_load_ps(&B[i * N + j]);
7          bx    = _mm512_load_ps(&x[j]);
8          accum = _mm512_fmadd_ps(bb, bx, accum);
9      }
10     y[i] = _mm512_reduce_add_ps(accum);
11 }
```

Listing 2.2: Manually vectorized code for general matrix-vector multiplication parallelized using a static block schedule.

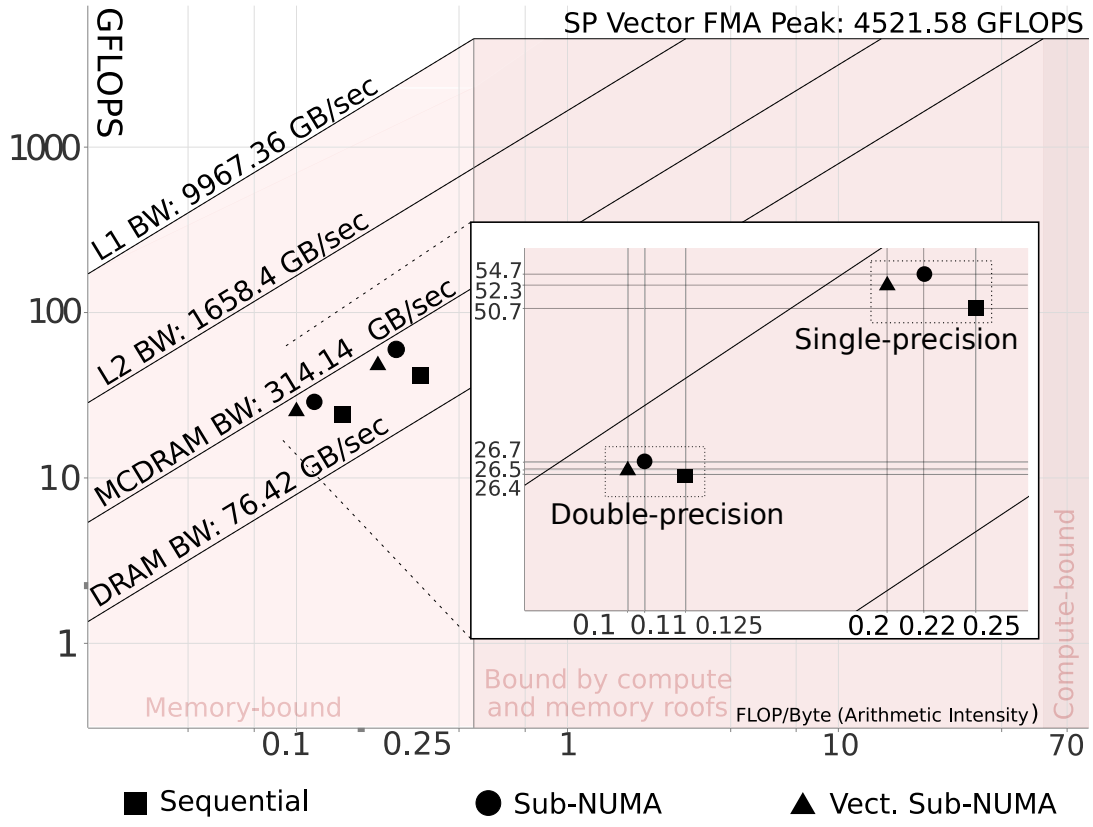


Figure 2.9: Roofline plot for the matrix-vector multiplication using both single- and double-precision arithmetic.

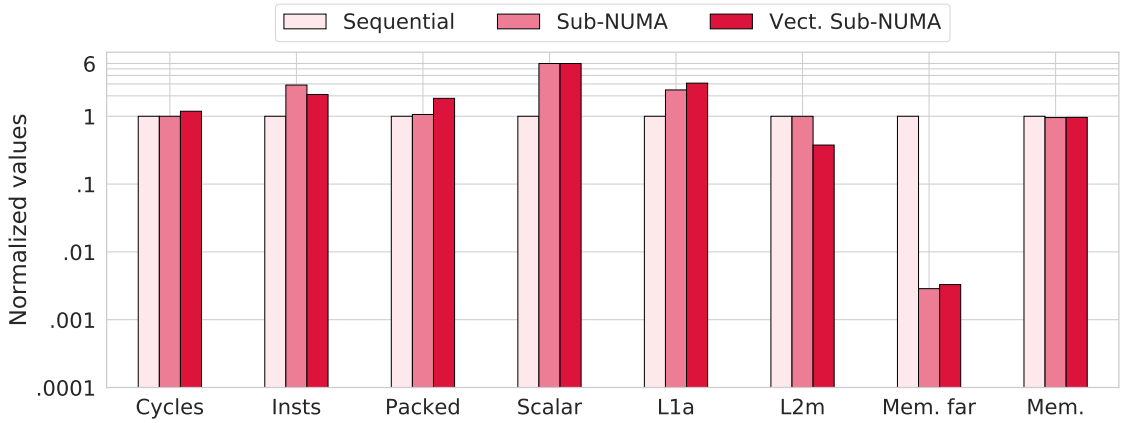


Figure 2.10: Sum of selected performance counters for all threads. Logarithmic scale is used for the Y axis. The figure shows the number of cycles, instructions issued, packed SIMD instructions, scalar SIMD instructions, L1 data accesses, L2 misses, MCDRAM "far" accesses to other quadrants in the NoC, and total number of MCDRAM accesses. Values are normalized to those of the sequential schedule.

are shown in Figure 2.10. In order to compute the sub-NUMA schedule, the number of instructions to be executed almost triples, increasing by 188%. The largest share of these are data L1 loads and stores, which grow by 145%. This increase, however, is absorbed by the L2 cache, and the L2 misses remain virtually identical. There is a very significant increase in the IPC of these codes, which goes from 18.6 in the original version to 53.15 in the sub-NUMA schedule. The memory latency, approximated by the `OFFCORE_RESPONSE_0:OUTSTANDING` performance counter, is slightly decreased by 1.8%. All these variables compound for an almost zero net effect on execution time: execution cycles are reduced by a modest 0.8%.

In order to try to decrease the schedule-related computations, a modified version which employs vectorization operations for offset computation was developed. In essence, the offsets for each 32 consecutive memory blocks are now computed using AVX-512 arithmetic. This version, labeled as "Vect. sub-NUMA" in Figures 2.9 and 2.10 reduces the number of instructions by 37.8% with respect to the regular sub-NUMA schedule. However, it worsens register pressure, increasing L1 accesses by a further 26%. As a result, the GFLOPS decrease to 52.3, and so does the AI to 0.20, for a grand total of 82.4% of the peak performance.

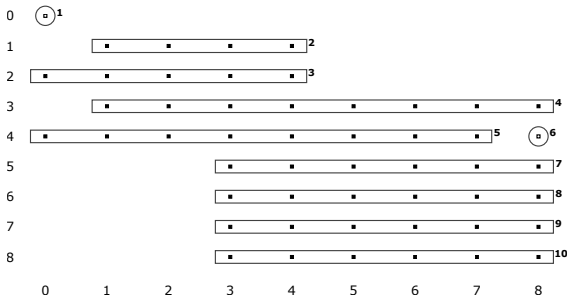
As previously mentioned, these results were executed after disabling hardware prefetching. The reason is that the sub-NUMA schedule does not access memory sequentially, and is at a tremendous disadvantage against the sequential schedule when the prefetcher is enabled, which would absorb and eliminate any potential advantage from the sub-NUMA schedule. In fact, when enabling the hardware prefetcher the performance of the sequential schedule is improved by 1.2x, whereas it is detrimental for sub-NUMA (i.e., its performance slightly decreases by approximately 5%) as it features a pseudo-random access pattern that mimics the memory-to-CHA mapping funtions.

2.8 Compile-time Optimization

As shown by the experiments in the previous section, improving the mesh locality during runtime has an important impact on other execution metrics due to the pseudo-random nature of the memory-to-CHA mapping functions and their computational complexity. A different way to exploit this knowledge is to optimize the scheduling of completely static codes during the compilation stage.

Augustine et al. [7] proposed a data-specific code generation technique for the optimization of sparse-immutable codes, including artificial neural network inference. In essence, this approach automatically builds sets of regular subcomputations by mining for regular subregions in the irregular data structure. The resulting code is specialized to the sparsity structure of the input matrix, but does not employ indirection arrays, improving predictability and SIMD vectorizability. This section focuses on the Sparse Matrix-Vector Multiplication (SpMV) as an immediate target of this class of data-specific optimizations.

A graphical depiction of a small subset of operations performed by the SpMV of matrix `FIDAP/ex7`, included in the SuiteSparse Matrix Collection [21] is offered in Figure 2.11. For many sparse matrices, this code generation approach delivers better performance than the generic, irregular alternative. Besides promoting vectorization, data-specific approaches encode the matrix structure implicitly in the program source. This does not only reduce the number of memory accesses, but collaterally stores the matrix structure in the first-level instruction cache, which is



(a) Nonzero points in the upper left corner of the matrix.

```
1  y[0] += A[0] * x[0];
2  y[1] += A[1:4] * x[1:4];
3  y[2] += A[5:9] * x[0:4];
4  y[3] += A[10:17] * x[1:8];
5  y[4] += A[18:25] * x[0:7];
6  y[4] += A[26] * x[8];
7  y[5] += A[27:32] * x[3:8];
8  y[6] += A[33:38] * x[3:8];
9  y[7] += A[39:44] * x[3:8];
10 y[8] += A[45:50] * x[3:8];
```

(b) Pseudo-code for a set of subcomputations in an SpMV kernel.

Figure 2.11: Sets of regular subcomputations built for the Sparse Matrix-Vector Multiplication of matrix `FIDAP/ex7` in the SuiteSparse repository. Each identified regular subcomputation is marked as a rectangle enclosing several nonzeros in (a), and captured as an AVX-512 operation, as shown in the pseudo-code in (b).

classically underutilized for small irregular codes such as SpMV. The effect is similar to extending the first-level data cache: matrix structure will be stored in the instruction cache (since it is embedded in the code), whereas actual matrix values will be stored in the data cache. The immediate disadvantage is that the code grows proportionally to the matrix size. Still, for sufficiently regular sparse matrices the combined size for structure and data values (the program footprint) will be small enough as to benefit from this trade-off.

As opposed to the dynamic approach of Section 2.7, the static optimization has no explicit execution overhead. As such, the schedule of each computation can be carefully analyzed and planned in order to improve coherence traffic. Note that, as opposed to the dynamic approach in which the mapping functions could be applied on already-allocated memory, in this case memory allocation must be statically known. The approach employed for this is detailed in Section 2.8.1. For the remainder of this section it is assumed that the physical address associated to each data block in the program is statically known.

Consider the generic SpMV statement s executed by the data-specific approach:

$$s : y_i = A_j \cdot x_k$$

Note that this statement does not include irregular indices, since the code has been generated for a specific input matrix with a fixed sparsity structure, as exemplified in Figure 2.11. Consequently, the compiler has static knowledge of all the memory movements that will be required for executing each specific part of the code. At a glance, the proposed compile-time approach computes an access cost for each statement in the data-specific SpMV code for each tile in the processor, and then schedules operations across the mesh following a greedy approach. Access costs are dynamically updated during the scheduling process to reflect the updated placement of each memory block in the private caches of each tile.

Consider a data block B with directory information associated to tile T_d and actual data accessed through tile T_B. The actual source of data can either be the private L2 cache of tile T_B, if the associated tile is the forwarder for B; or T_B can be one of the tiles with an associated memory interface, which will serve B after reading it from memory. Regardless of the actual coherence status of B, in order to access the data the requestor tile will send a message to T_d, which will forward the request to T_B, which in turn will send B back to the requestor. Figure 2.12 illustrates this situation. Note that T_d and T_B constitute the opposite corners of a rectangle on the NoC which contains the tiles that can access B with minimum latency. Tiles outside this rectangle incur extra latency, which can be computed as $2 \times (2 \times D_x + D_y)$, where D_x and D_y are the horizontal and vertical distances from the tile to the rectangle, respectively.

Based on these access times, a scheduling system is developed, conceptually described in Algorithm 2.2. Each tile in the NoC is visited in order, and for each of them the subset of operations to be executed on that tile is selected in a greedy, iterative fashion, choosing the one with the smallest data movement cost at each iteration, until that tile reaches its balanced load. The upper bound of its computational complexity is $O(\mathcal{S}^3)$: the algorithm essentially distributes all of the statements in the program to the tiles in the mesh, which would present linear complexity on $\mathcal{S}$. However, the cost of the set of remaining statements has to be recomputed frequently, due to the data movements derived from the assignment decisions in each iteration of the inner loop. The cost τ of executing each statement s in tile t is computed as:

$$\tau(s,t) = \tau(y_i,t) + \tau(A_j,t) + \tau(x_k,t)$$

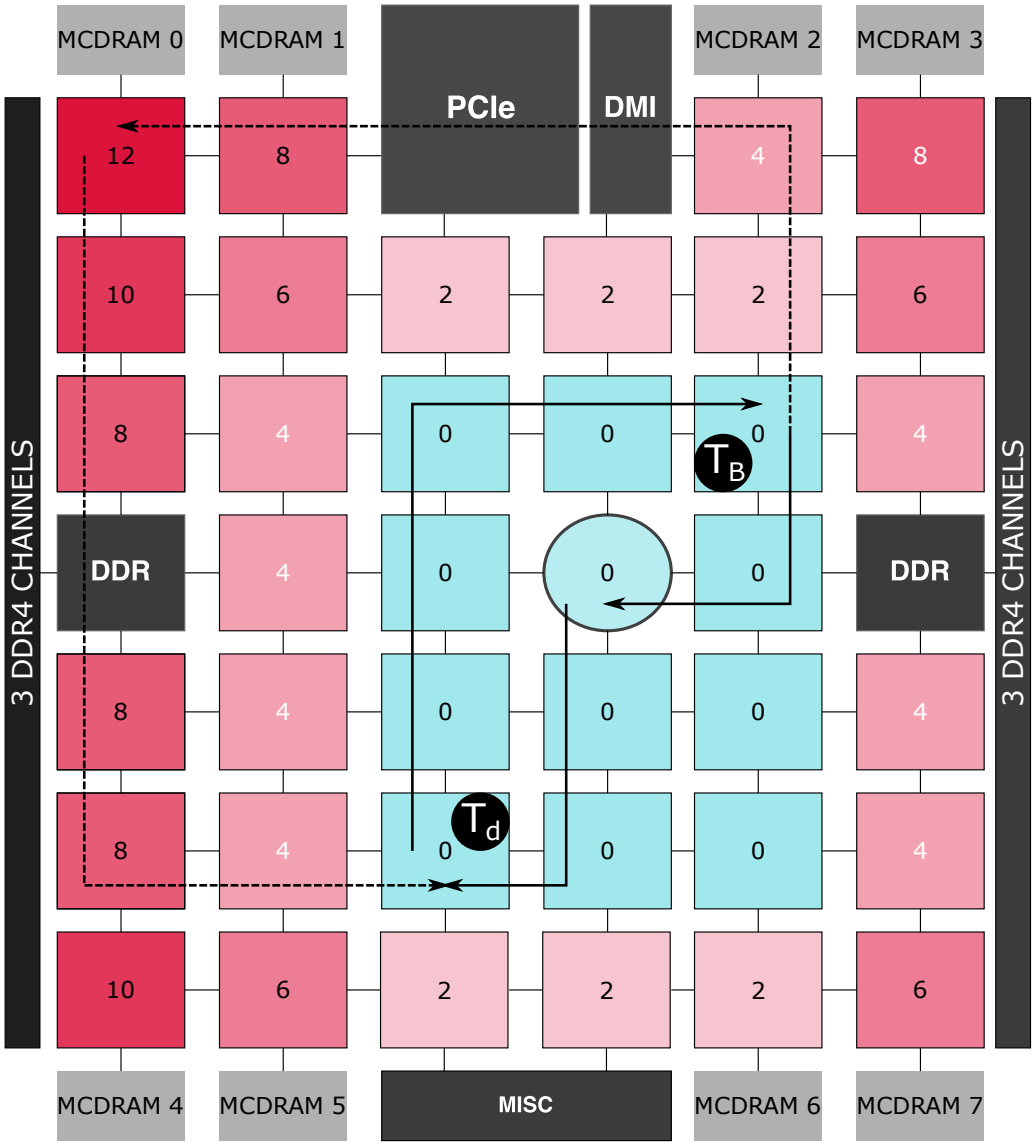


Figure 2.12: Overhead, in mesh cycles, of accessing a block of data resident in the L2 cache of tile T_B, and with coherence information resident in tile T_d. All the tiles inside the rectangle defined by T_d and T_B access data in 14 cycles with zero associated overhead. In any communication, first the CHA at T_d is queried. Then the CHA sends a forward request to the source L2 cache at T_B, which sends the data back to the requestor. For any tile inside the 0-overhead rectangle, being closer to the CHA means a shorter travel time for the query, and a larger one for the response. These compensate one another, yielding zero net effect. For tiles outside the rectangle, the overhead is compounded by the extra time that the query needs to enter the rectangle, plus the extra time that the response needs to arrive back at the requestor from inside a rectangle tile.

Algorithm 2.2: Static scheduling of SpMV operations

Input: Set of SpMV statements $\mathcal{S}$ to schedule, Set of tiles $\mathcal{T}$ in the NoC
Output: Schedule $\Theta(\mathcal{S}) \rightarrow \mathcal{T}$

```
Compute L_T = total number of FLOPS in S;
Compute L_b = L_T / #T FLOPS to be computed by each tile to balance load;
foreach tile t ∈ T do
    while Load(t) < L_b do
        Select s ∈ S : τ(s,t) ≤ τ(s',t), ∀s, s' ∈ S;
        Assign Θ(s) = t;
        Update S = S − {s};
    end
end
```

That is, the aggregated cost of accessing memory blocks y_i, A_j, and x_k from tile t. For each individual memory block, its access cost is computed as:

$$\tau(B,t) = \lambda_B + 2 \times (2 \times D_x(t, R_B) + D_y(t, R_B))$$

where:

- λ_B is a factor that depends on the latency to physically access B, including 12 cycles for accessing a private L2 in the NoC [62], and 117 cycles for accessing an MCDRAM interface (according to our measurements in Section 2.3).
- $D_{x/y}(t, R_B)$ is the horizontal/vertical distance from the requestor tile t to the rectangle defined by the tiles in its opposite corners T_B and T_d, containing the data and the coherence information, respectively, as described in Figure 2.12.

The order in which each of the tiles is visited is carefully selected: those with worst-case trip times are selected first. For instance, the upper-left tile in the NoC has a worst-case round trip time of 32 cycles when accessing data with T_d or T_B on the bottom-right tile. However, the round trip time from a central tile to any other tile in the mesh is of at most 18 cycles.

Note that the schedules generated by this static optimization process are no longer sub-NUMA, as opposed to the dynamic approach in Section 2.7. In this case, there is no runtime constraint enforcing quick computation of the schedule, so the

system can use the full fine-grained information about memory-to-CHA mapping to decide whether accessing data on a different quadrant will be the best option from a coherence traffic point of view.

Once all operations are scheduled, the code is generated specifically for each tile. In order to reduce code sizes, affine compression may be applied to group similar operations together on regular affine loops [108]. These do not employ indirection arrays, being still fully vectorizable, while reducing the pressure on the instruction cache.

2.8.1 Fixing physical addresses

One of the challenges of static scheduling with this class of pseudo-random functions is that it is not possible to compute the associated CHA of a virtual address, as the 34 least-significant bits of the address will be used. Even with 1 GiB hugepage sizes, the maximum supported by the architecture, only 30 bits remain unchanged during the virtual-to-physical address translation. This means that the code cannot rely simply on page alignment, as can be done for cache optimization, and must target specific physical pages.

In order to fix the physical pages that are assigned to a specific application, we employ 1 GiB hugepages. Since the MCDRAM address space has only 16 GiB in total, there will only be 16 possible pages that can be assigned to our application. The assignment order varies slightly depending on the machine state upon launch. To overcome this difficulty we employ a hybrid static/dynamic approach. During the static analysis, the code generated assumes that specific 1 GiB hugepages will be allocated to the different data structures in the program. These assumptions are registered in static constant variables in the source code. During runtime, an executor overallocates as many 1 GiB pages as possible. Then, it translates their virtual addresses to physical addresses by reading the process pagemap in `/proc`. Finally, it assigns the required hugepages to the data structures in the code by comparing the allocated physical addresses to the static constant variables assumed during the scheduling process, and frees the remaining, unused ones.

2.8.2 Experimental results

We generate data-specific codes for more than 20 sparse matrices selected from the range of matrices between 1 million and 10 million nonzeros in the SuiteSparse repository. The upper bound is used for tractability purposes. The lower bound to ensure sufficiently large operation. The selected matrices were the cluster centroids resulting from running k-means on SuiteSparse and using regularity and size as the target characteristics [7]. Each of the selected matrices was processed to extract the data-specific operations required by its SpMV. Three different implementations were generated for testing:

- The generic irregular version of Listing 2.3.
- A data-specific version with sequential schedule, as described by Augustine et al. [7].
- A data-specific version containing exactly the same set of operations, but scheduled in a coherence-aware fashion using Algorithm 2.2.

Codes are compiled using ICC 19.1.1.217 with `-Ofast -xKNL -qopenmp`. They are executed on an Intel x200 7210, running at the base frequency of 1.30 GHz, to avoid turbo-related variations, using 64 threads, one per core in the NoC. Ten repetitions were performed for each execution, and average values are reported for each thread after discarding outliers (identified as values x such that $|x - \bar{X}| > 3\sigma(X)$). For the generic irregular version and the sequentially-scheduled data-specific one the "scatter" thread placement is employed. For the coherence-aware version an ad hoc assignment is employed, ensuring that each thread is executed on the appropriate

```
1 #pragma omp parallel for private(j)
2 for(i = 0; i < n; ++i) {
3     y[i] = 0;
4     for( j = pos[i]; j < pos[i+1]; ++j)
5         y[i] += A[j] * x[cols[j]];
6 }
```

Listing 2.3: Classic irregular SpMV code.

statically scheduled tile. These codes are typically very large in size, explicitly containing the full set of operations to be performed for multiplying a sparse matrix by a given vector. Executable sizes vary between 39 and 206 MiB. As for dynamic scheduling, `hugectl -heap` and `numactl -m 1` are used to control the configuration of hugepages and memory domains. The hardware prefetcher is enabled for all the experiments in this section.

The data-specific versions were found to be 2.1x faster on average than the generic irregular version. This is a clear indication that a manycore architecture with light, vectorization-oriented processors is not well geared towards irregular codes, which feature many control flow-related instructions such as induction variable increments and branches. The data-specific versions perform, on aggregate, 4.7x less L1 accesses, but incur 1.2x more L1 data misses. The L1 instruction misses increase by 39.9x. This increase is mostly absorbed by the L2 cache and the hardware prefetcher, however, and overall the number of L2 misses is only 12.8% higher in the data-specific versions. Furthermore, these additional misses are resolved locally by the mesh, and the number of MCDRAM accesses decreases by 21.6%. In summary, the memory behavior, which is potentially the weakest runtime aspect of a data-specific version, is not significantly worsened. In exchange, the data-specific codes execute 5.4x less instructions, including 2.3x less scalar operations and 859x more vector operations. The biggest culprit in runtime difference is precisely the number of executed instructions, and the number of stalls due to missing reservation stations is 4.9x larger on irregular codes. Due to these intrinsic differences in the nature of each implementation, we drop the irregular version of SpMV in the following experiments, and focus on comparing only the sequential and coherence-aware schedules of the data-specific implementations. Figure 2.13 shows these detailed results.

From a performance point of view, on aggregate the coherence-aware schedule increases execution time by 3.2%. The detailed execution cycles obtained for the SpMV of each matrix are shown in Figure 2.14. None of the matrices achieves a performance improvement, the best one being 0.1% slower than the baseline. For some matrices the operation is noticeably slower, the extreme case offering 10.3% less performance.

In order to study in more detail the reasons for this performance degradation, three selected matrices are closely examined. Selected performance counters for

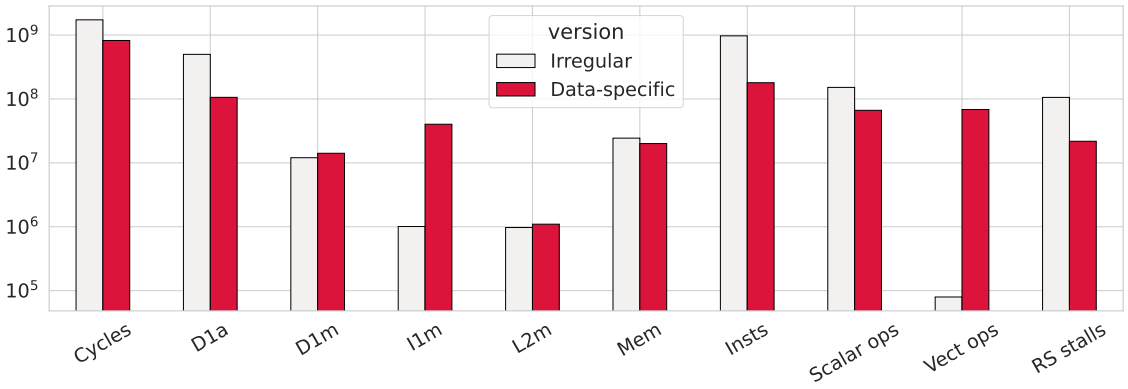


Figure 2.13: Execution cycles, data L1 accesses, data L1 misses, instruction L1 misses, L2 misses, MCDRAM accesses, executed instructions, scalar operations, vector operations, and stalls due to missing reservation stations (RS) for irregular and data-specific versions of the SpMV operation. Note that the Y axis is truncated for readability.

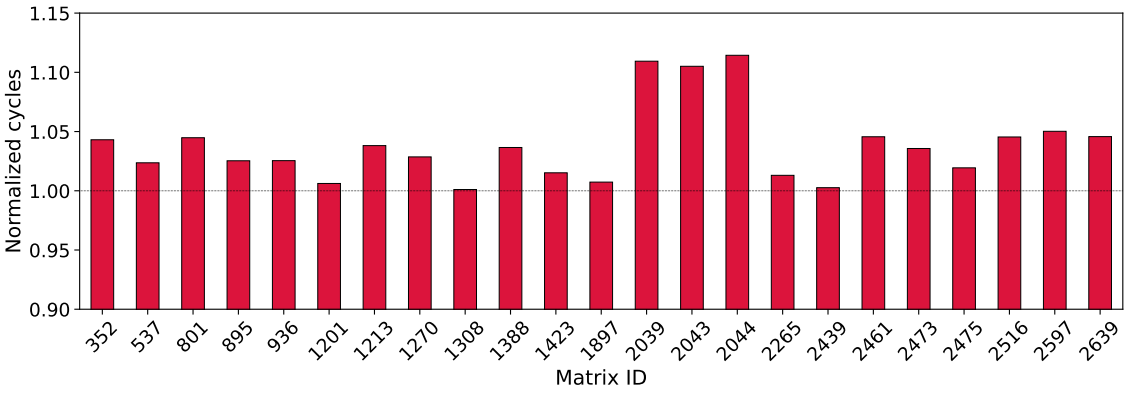


Figure 2.14: Execution cycles of the coherence-aware schedule, normalized to those of the sequential schedule. Note that the Y axis is truncated for readability.

these matrices are detailed in Figure 2.15. On careful inspection the performance is strongly correlated with the number of MCDRAM accesses incurred by each version of the code (Pearson correlation coefficient R=0.91).

The conclusion to be inferred from these experiments is that, even with fully static scheduling, CHA locality cannot be appropriately leveraged to improve performance of data-specific sparse codes. The reason is that, due to the pseudo-random nature of the assignment between memory blocks and CHAs, rescheduling the code to promote the access of nearby CHAs to improve the cache coherence traffic patterns necessarily impacts cache locality negatively for codes benefiting from sequential data access. Even though SpMV has a varying degree of randomness in the access to the x vector, the matrix data in A can be accessed sequentially, and this is a huge advantage of the sequential schedule, particularly taking into account the hardware prefetcher. Despite the performance degradation, a careful analysis of the performance counters evidences that the coherence-aware schedule broadly improves memory latency, as shown in Figure 2.16, by 10% on aggregate. Average latency goes from 0.77 cycles per access in the sequential schedule, to 0.70 cycles per access in the coherence-aware schedule. The IPC is very slightly increased, going from 12.37 to 12.42.

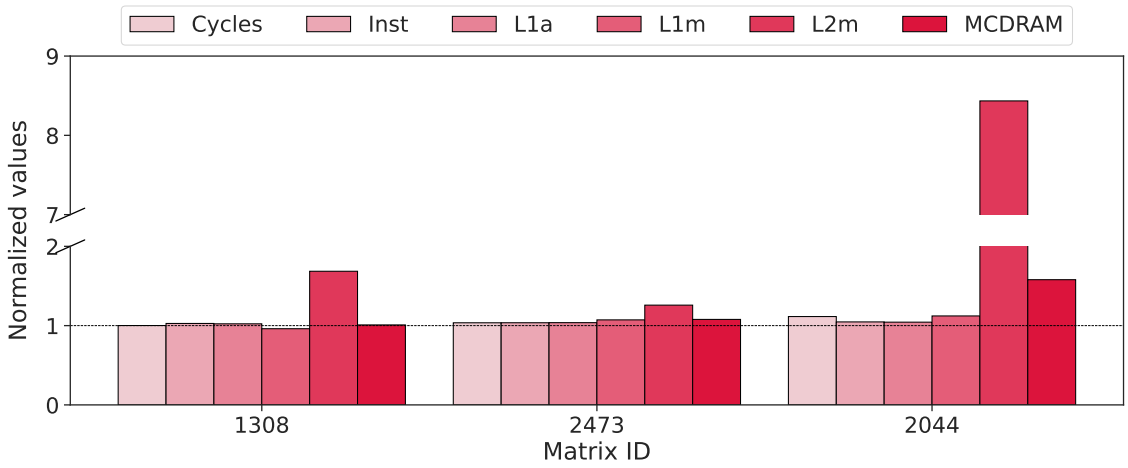


Figure 2.15: Execution cycles, executed instructions, data L1 accesses, data L1 misses, L2 misses, and MCDRAM accesses of the coherence-aware schedule for selected matrices in the experimental setup: the one with the best relative performance (#1308), the one with the worst one (#2044), and the middle case (#2473). Values are normalized to those of the sequential schedule.

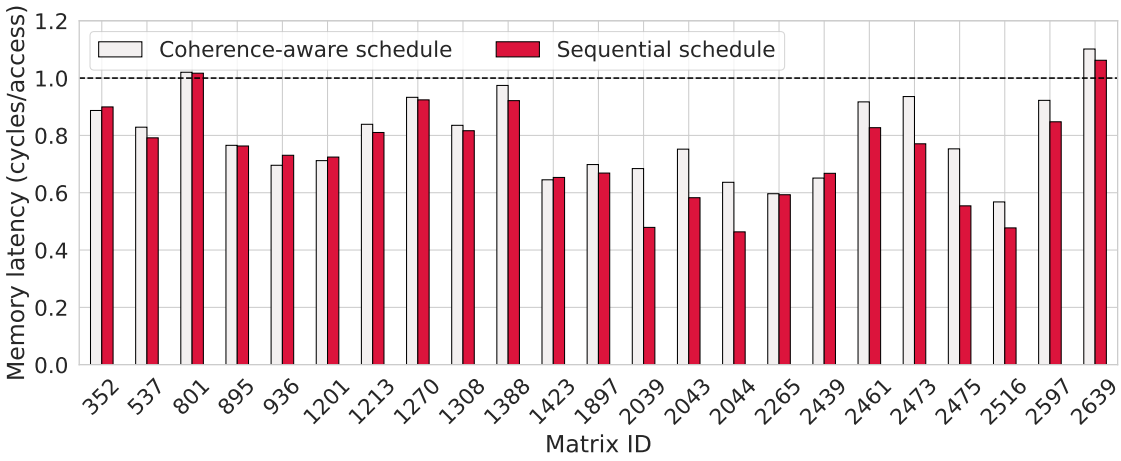


Figure 2.16: Memory latency of the coherence-aware schedule normalized to the sequential schedule baseline for all the matrices in the experimental setup.

2.9 Discussion and Related Work

When we initially explored the optimization of coherence traffic on the Knights Landing NoC, we observed a clear effect on the application performance due to affinity relationships between cores and CHAs. This work was based on a pre-computed assignment of memory blocks to CHAs, as described in Section 2.3. The optimized scheduling was performed dynamically in an inspector-executor fashion, which represents a very costly step that would negate any actual performance benefit in a real setting. Furthermore, the rescheduling could only be applied to irregular codes. Based on these promising results, we also extended this work by reverse engineering the functions behind this mapping. We expected these functions to be useful to alleviate the overhead of the inspector-executor approach, in addition to being usable by architecture-specific compilers that could perform low-level optimizations of coherence traffic. However, these expectations were toned down by the actual shape of the mapping functions. Although the XOR-based functions are cheap to implement in hardware and widely used for other non-regular mappings, such as the assignment between memory blocks and LLC slices in Intel Core processors [29, 60, 83], they are costly to compute in software. This cost can be overcome if the mapping presents some kind of regularity that can be exploited by carefully optimizing the code and schedules. For instance, Scolari et al. [116] propose to exploit the knowledge about

the hash functions that map data to LLC slices in an Intel Sandy Bridge processor to achieve performance isolation. This is possible since the hash functions which govern this mapping only employ a simple XOR of selected bits from 17 to 32 of an address, which means that blocks of 64 KiB of contiguous and aligned data will be mapped to the same LLC slice. This approach is limited to processors having a power of 2 number of cores, as otherwise the mapping functions become nonlinear. This is the reason for the contrasting complexity of the equations presented in Section 2.6, which require XOR and negations, dramatically broadening the search space and making brute force approaches essentially infeasible. This complexity is derived from the non-power of 2 number of tiles in the Xeon Phi x200, as shown by McCalpin [88]. His works on Intel Xeon Scalable and Knights Landing [86, 89] were developed completely independently of our research work.

When trying to exploit the mapping of coherence data to CHAs in the Xeon Phi x200 architecture, the software complexity of the XOR-based hash functions, together with their pseudo-random nature, in which sequences of four consecutive memory blocks are guaranteed, by design, to be mapped to CHAs in different quadrants, makes it impossible to apply similar, regular schemes. The approaches proposed in Sections 2.7 and 2.8 achieve to reduce contention on the processor network, but ultimately do not achieve to improve execution performance due to the computational cost involved in the optimized scheduling.

Note that the approach followed in this chapter has focused on a particular Intel Xeon Phi 7210 unit, but it is generalizable to any Xeon Phi 72xx. We have studied other units, including 7250 and 7290, and found that the memory-to-CHA mapping is identical, as is the physical placement of the CHAs on the network and the way to distribute the logical core IDs over the set of enabled physical cores. This set, however, is subject to fabrication process variations and changes for each specific unit of the processor. This is also assessed in [86] simply by running the CPUID instruction on each active logical processor to obtain that core's x2APIC ID (any cores that are disabled will result in missing x2APIC values in the list)[4]. The process for mapping the logical components of the processor to their physical locations, based on profiling the communication latency of different logical entities in the processor, followed by a discrimination process based on mean squared error models, is

[4]x2APIC is the evolution of xAPIC, Intel's family of interrupt controllers.

applicable to other processor designs. The process presented to reverse engineer the binary functions for the memory-to-CHA mapping could be applied when searching for hardware-friendly hash functions in general. The presented flow chart in Figure 2.7 may not work for a specific problem, but could provide useful information which indicate adjustments to explore or rule out certain function forms. In the same line, McCalpin [89] also presents a reverse engineering process for mapping the CHAs in KNL by only requiring measuring the data traffic counters. This approach is as accurate as ours, but it requires understanding and transforming the values of the low-level uncore hardware counters.

For all these inconveniences from the high performance computing point of view, the approach followed by Intel has many advantages in everyday computing. It is implausible to write a code that systematically accesses only a particular set of CHAs, making them into a bottleneck. Such a bottleneck can happen with regular mappings, such as a modulo-based mapping that can suffer from systematic conflicts for certain access patterns. Furthermore, it manages to distribute memory blocks across the quadrants in the NoC in a fair fashion, ensuring that all of them have to manage the same amount of information on aggregate. This is no simple task, given the irregularity of the NoC, which features a non-power of two number of tiles, unevenly distributed across quadrants. Still, the price to pay is an all-to-all coherence traffic pattern which requires dedicated communication rings to handle.

Going forward, it would be desirable to improve this design, coupling the directory distribution that avoids bottlenecks in the NoC with a more regular and predictable mapping of the memory blocks to enable programmers, particularly in the high performance computing domain, to have full control over coherence traffic. McCalpin [85, 86, 87, 88, 89] discusses address hashing in Intel Scalable and KNL processors, analyzing the binary permutations in address bits and their physical location in the mesh, i.e., their corresponding CHAs. His work focuses on cache line distribution, rather than on the actual hash functions.

In recent years, a number of papers have explored the design of scalable NoCs to support manycore architectures. Daya et al. [22] design a NoC based on an ordered network and a snoopy coherence protocol, and show how congestion increases heavily with the number of cores. Ferdman et al. [30] propose a scalable distributed directory system to alleviate the power and performance problems of sparse and duplicate-tag

directories, scaling up to 1,024 cores. Charles et al. [17] identify the importance of the coherence traffic in manycore performance, and show how the memory modes in the Intel KNL can be manipulated to achieve better performance. They neither explore software optimizations to coherence traffic, nor the actual layout of the KNL processor. Numerous other techniques to efficiently schedule applications on NoCs have been developed. Kim et al. [64] focus on categorizing the memory access behavior of threads and employing different policies for them. Xiao et al. [133] parallelize applications by implementing careful load balancing between cores and minimizing inter-core communications. Liu et al. [76] use a compiler-guided scheme to minimize on-chip network traffic by reducing the distances of cores to data, but without taking into account the effects of a distributed directory. Lu et al. [78] propose a polyhedral model and associated optimizations to achieve data locality in these topologies. In these works, no particular consideration is given to the effect of distributed cache home agents on memory access latency, and therefore deploying such approach on a KNL may be refined with placement and subsequent inter-core communications further improved using the results we presented earlier.

Several papers have explored the performance of the KNL architecture, mainly through the analysis of well-known benchmarks, machine learning applications, and parallel workloads [8, 15, 18, 61]. None of these works undertake the analysis of the locality characteristics of the KNL interconnect. Ramos and Hoefler [105] developed a capability model of the cache performance and memory bandwidth of the KNL, characterizing the impact of the different memory and cluster modes. However, this work does not consider the impact of the distributed directory.

"Talk is cheap. Show me the code."

–Linus Torvalds [126]

3

Simulating the Network Activity of Modern Manycore Architectures

Chapter's contents

In Chapter 2 we studied modern manycore architectures, focusing on the Intel Knights Landing, which features the Intel Mesh Interconnect (MI) architecture. This is the latest interconnect designed by Intel for its HPC product lines, also recently featured in Intel's Xeon Scalable lines. Processors are organized in a rectangular network-on-chip, connected to several different memory interfaces, and using a distributed directory to guarantee coherent memory accesses. As we studied in the

previous chapter, there is an impact on performance due to the coherence traffic traversing the mesh interconnection network. In this context, simulation can help to study this impact and make improvements accordingly.

3.1 Introduction

In this chapter we focus on the behavior of the MI in the KNL architecture, leveraging the knowledge of the physical layout discovered in the previous chapter. Since the traffic on the NoC is completely opaque to the programmer, simulation tools are needed to understand the performance trade-offs of code optimizations [82]. We have designed and developed an extension to the Tejas memory system simulator [113] to replicate and study the low-level data traffic of the processor network. The reliability and accuracy of the proposed simulator is assessed using several state-of-the-art sequential and parallel benchmarks, and a particular Intel MI-focused locality optimization is proposed and studied using the simulator and a real KNL system. The techniques and software developed in this work are potentially reusable for analyzing MI-based processors, such as the Intel Xeon Scalable. Specifically, in this chapter we present the following contributions:

- We incorporate a model of the memory accesses of the MI architecture into the Tejas architectural simulator. This extension has been made publicly available in [48] (Section 3.4).
- We validate the model by comparing the behavior of the simulated implementation against a real Intel Knights Landing processor using sequential and parallel benchmarks (Section 3.5).
- We leverage and evaluate the coherence traffic optimizations presented in Chapter 2 using our extension developed within the Tejas simulator (Section 3.6).

The rest of the chapter is structured as follows: Section 3.2 motivates the work. Section 3.3 introduces the Tejas simulator. Section 3.4 shows how to incorporate the developed model into the simulator. Section 3.5 presents the experimental validation

of the model. In Section 3.6 we present a case study on optimizing the coherence traffic of KNL taking into account the effect of the distributed cache directory. Related work is discussed in Section 3.7, and Section 3.8 concludes the chapter with our final remarks and future work.

3.2 Overview and Motivation

The design of the Intel Knights Landing [62] is carefully described in detail in Section 2.2. It features a 2D mesh for the interconnection network as depicted again and in Figure 3.1. As we have discussed in Chapter 2, the effect of coherence traffic on that network requires a deep understanding of the NoC architecture. Then, this knowledge can be translated into an architectural simulator in order to develop and evaluate potential optimizations from both software and hardware perspectives. Architectural simulators are useful and effective frameworks for analyzing different configurations of hardware with a lower cost and effort in comparison to other solutions such as FPGAs or ASICs (to the detriment of accuracy). Another advantage of these tools is their programmability, the ease to create new modules or modify existing parts of the simulator, in order to change or extend the functionality of the framework. There are many simulators in the market, some of them implementing more details in the pipeline, such as gem5 [11], meanwhile others are able to simulate up to thousands of cores, such as ZSim [111]. Finding a suitable solution for modeling the NoC is not an easy task, but for this work we have selected Tejas [113], a lightweight and easily extendable simulator. The key factor in our work to choose this simulator among other solutions was the ability to simulate a NoC of cache coherent interconnected cores.

For these reasons, in this chapter we extend the Tejas simulator by modeling the Intel KNL architecture focusing on its interconnection network and memory system. For assessing its performance, we employ the methodology presented and carefully described in Section 2.4 of Chapter 2, which demonstrates the potential of using an ad hoc data distribution that takes into account the distance between cores and CHAs, i.e., the core-to-CHA affinity. In our model, by applying these techniques we want to assess, in particular, the reduction in the network footprint, which is a critical parameter for NoC performance [17, 76, 134]. In this way, leveraging the

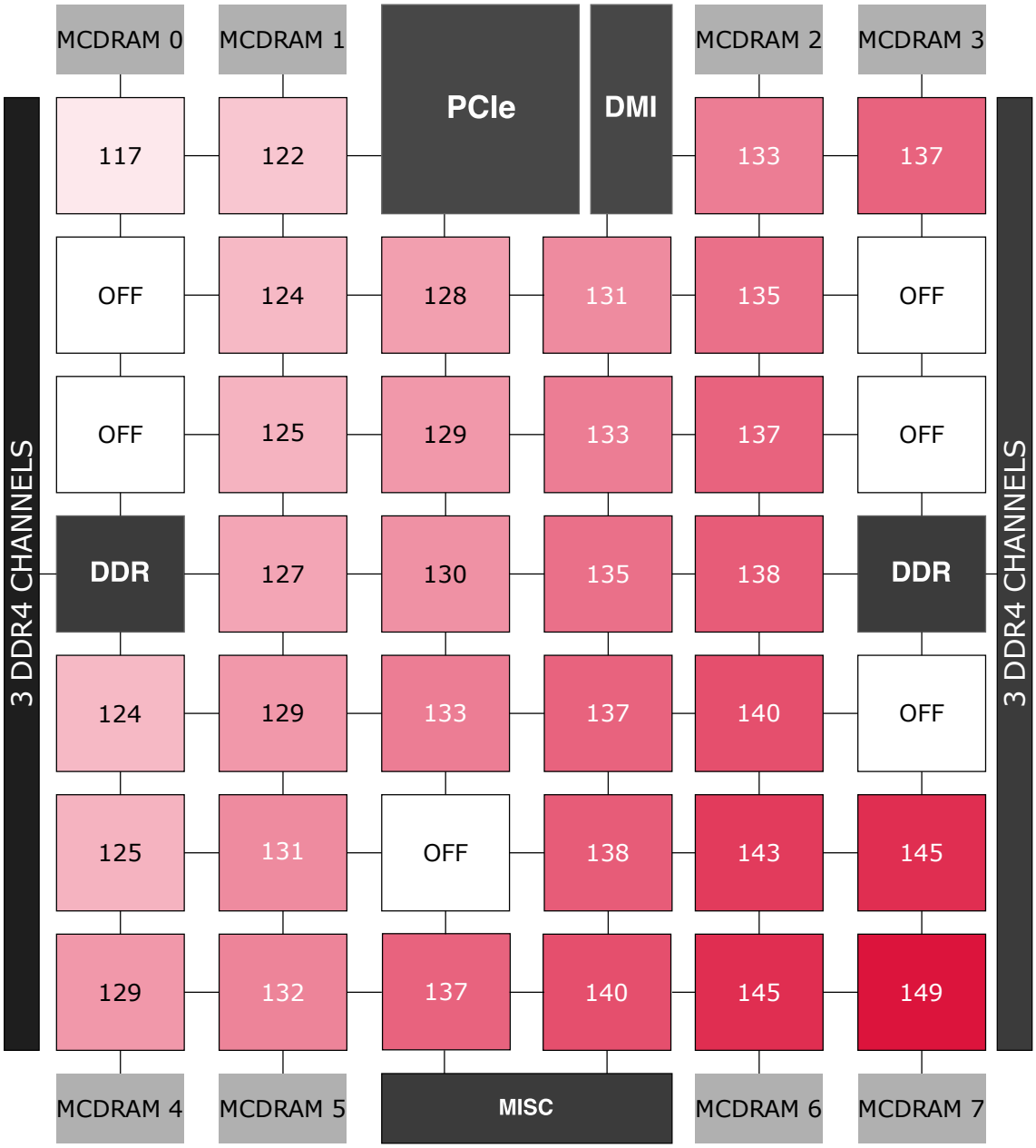


Figure 3.1: Floorplan of the Intel KNL architecture. Each tile (box in the figure) contains two cores and their local caches. In color, the heatmap of the measured access latency (in CPU cycles) from each tile in the mesh of an Intel Xeon Phi x200 7210 to a single block of memory associated to MCDRAM #0 and its adjacent CHA.

knowledge from the Intel KNL architecture, the goal of this simulator extension is to provide a reliable environment to evaluate and optimize the coherence traffic generated in the NoC. In addition, this extension is intended to be applicable to other modern MI architectures, including the latest Intel Xeon Scalable.

In the following sections we will discuss the extension developed for the simulator, and the assessment against different benchmarks.

3.3 Tejas Simulator: Architecture and Extensibility

Tejas is an open source architectural simulator written in Java and C++ [113]. The simulation model is semi-event-driven: predictable activities follow an interactive cycle-level approach, such as advancing micro-operations in the pipeline, whereas unpredictable tasks, such as load/store operations, follow an event-driven approach based on priority queues. Tejas decodes binaries dynamically during their execution in order to recreate the micro-operations that are used to feed simulation components, such as processor pipelines. Tejas also employs the McPAT [75] and Orion2 [63] power models in order to get statistics about energy consumption.

This framework has been validated against real hardware in terms of cycles, reporting acceptable mean errors varying between 11.45% for serial workloads and 18.77% for parallel workloads [112]. The simulator can be seen as two separate modules or layers: the emulator (front-end) and the simulation engine (back-end). The front-end instruments the code being executed, ensuring full functionality. It is ISA-dependent, and in our case the x86 implementation is used. The back-end receives events from the front-end and simulates them in the configured architectural model. The following subsections describe each of these components in more detail.

3.3.1 Front-end: the emulator

The emulator performs dynamic binary instrumentation in order to inspect the instructions executed by the input program. Then, these instructions are translated into virtual micro-operations recognizable by Tejas. This process is done in two steps (see Figure 3.2):

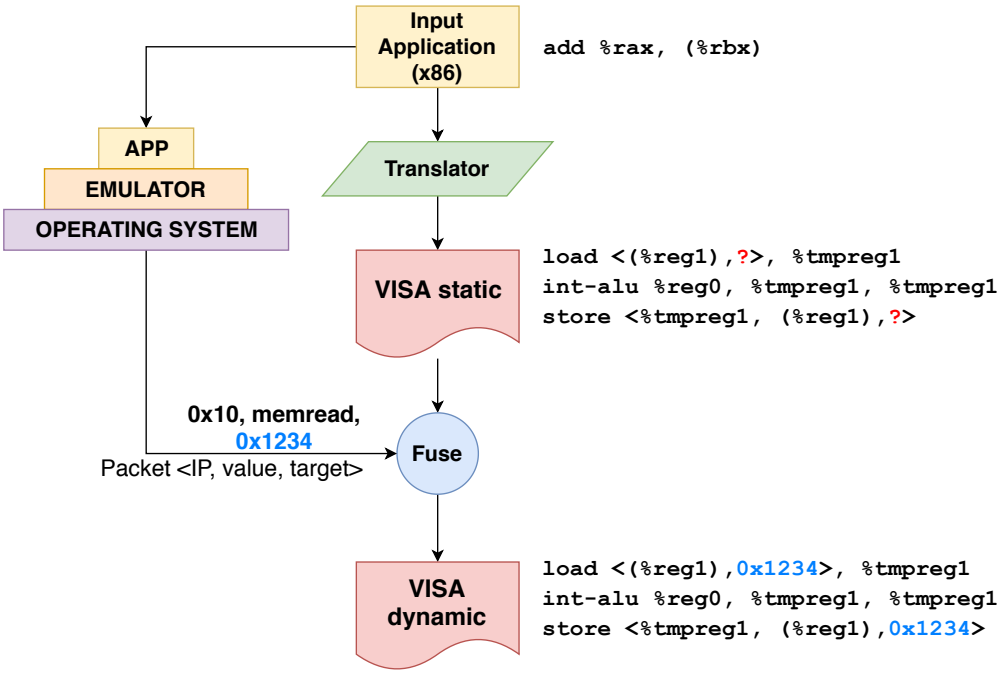


Figure 3.2: Translation process in Tejas from the binary instrumentation to the VISA.

1. *Riscify* process: converts CISC x86 instructions to a simplified ISA called Virtual ISA (VISA). This process is done statically and does not include source or target memory addresses.

2. *Fuse* process: dynamically fills the missing memory addresses of the VISA instructions created in the previous step.

This module of Tejas is written in C++ using Pin [79] and, therefore, is basically an instrumentation of the code that is being executed. What the emulator collects is used to feed the simulation engine (the back-end), written in Java, which is in charge of starting all the pipelines and processing elements, and collecting all the statistics at the end of the execution. The communication between the emulator and the simulation engine is done through a shared memory region.

The main limiting factor of the front-end is the translation between the real micro-operations and the VISA. Some CISC micro-operations are translated into functionally equivalent VISA instructions. An example are SSE instructions, which are translated into equivalent scalar floating-point ones, even though the VISA does

not implement SSE. In other cases, some instructions may be ignored. This is the case for instructions such as AVX-512F (which are 512-bit vector instructions released with Knights Landing). This limitation is imposed due to the complexity of CISC architectures and in order to keep the simulated pipelines simple. Tejas' developers chose to focus on the most common x86 micro-operations, usually obtaining a coverage of more than 99% of the dynamic binary instructions [113].

3.3.2 Back-end: the simulation engine

The simulation engine can be seen as a set of interconnected components, sending and receiving messages through their ports. For the scope and interest of this work, we will briefly review how cores, memory system and interconnection networks, as well as their corresponding configuration parameters, are implemented in this framework.

Cores

A Tejas core is a wrapper entity containing a pipeline and both data and instruction private caches. Two pipelines have been implemented in Tejas: Multi-issue in-order and Out-of-Order (OoO or O3). We used the O3 implementation, depicted in Figure 3.3. It consists of nine stages: instruction fetch, instruction decode, rename, instruction window push, instruction select, execute, wake-up, write-back and commit. The sizes of each functional unit and registers can be configured, allowing for a wide variety of possibilities.

Memory system

The Tejas memory system is composed of a set of memory controllers and caches. The memory hierarchy can be seen as an inverted tree of caches where its root is the main memory. Regarding the software implementation, the cache coherence directory inherits from the cache class. A centralized directory is implemented, which is queried by last-level caches to discover the state and location of accessed memory lines.

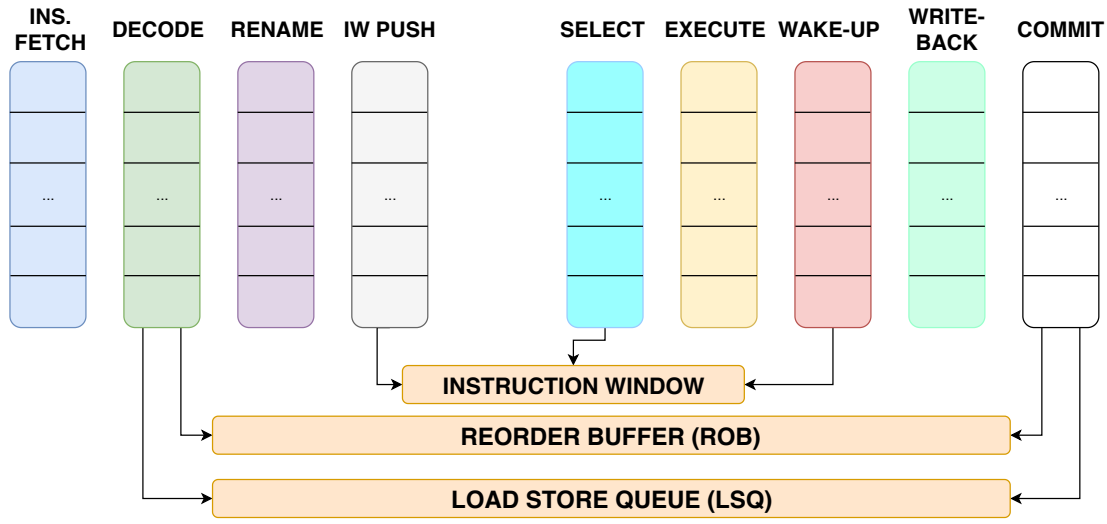


Figure 3.3: Stages and main registers of the Out-of-Order pipeline in Tejas.

The memory system is flexible, allowing to configure each cache as private or shared among a set of cores. Thus, buffer sizes, cache lines, MSHR (Miss Status Holding Registers), write mode and number of cache ports can be changed, opening a wide range of possible configurations.

The flow of a simulated cache request is depicted in Figure 3.4. When the request is received, it is first checked whether it is a hit or a miss. In the latter case, a new entry is added to the MSHR, allowing to perform other tasks while miss requests are handled by other parts of the mesh, either other caches or memory interfaces. When a lower level response is received, the event is removed from the MSHR and the response is processed: it can be a miss response to fulfill the missed request, a write ACK in order to mark the line as dirty, or an evict ACK in order to mark the line as invalid. When there is a miss in the last-level cache (LLC), the request is forwarded to the centralized cache directory, which will point to the location of the line, satisfying the request. The directory also handles coherence, using a simplified version of the MESI protocol. For instance, when a line is in shared state, the directory selects one of the sharers without taking into account priority or proximity.

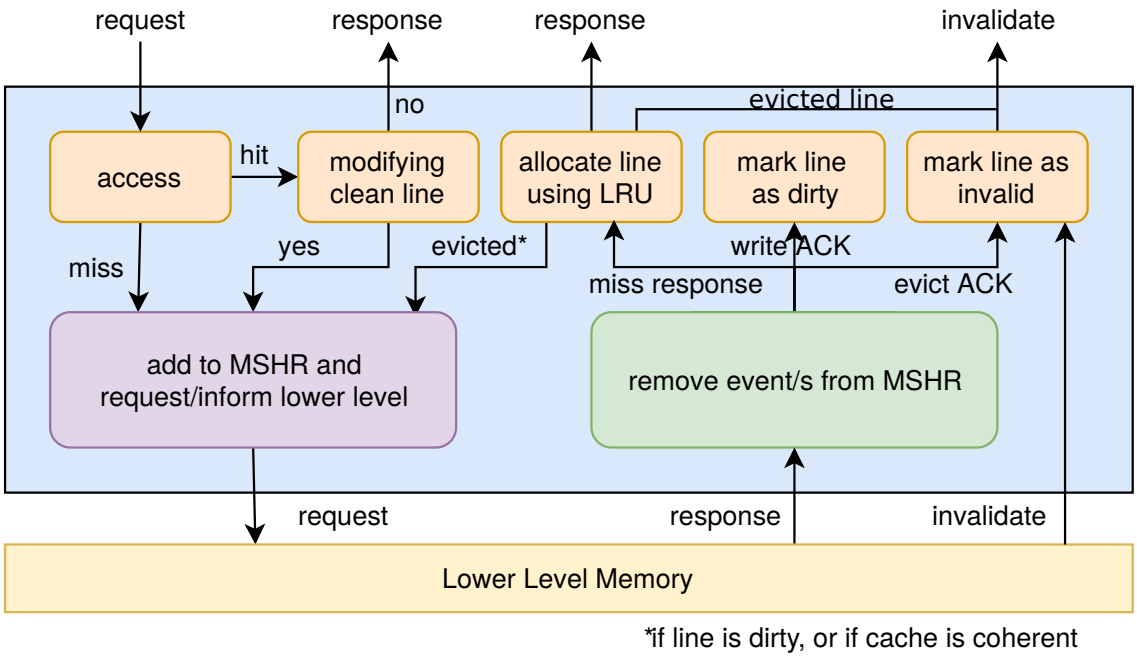


Figure 3.4: Cache behavior in Tejas (extracted from [113]).

Network-on-chip

Tejas implements a generic NoC which can be configured in different manners. It is a crucial component, since it connects all the different elements present in an architecture. Many topologies (bus, ring, mesh, torus...) and routing algorithms (west-first, simple-XY, north-last...) are available.

Both the shape and the low-level parameters of the NoC are highly customizable. In the same manner, the dimensions and type of the topology, as well as the router latencies and capacities can be modified. In Tejas, each router connects elements to the network and different elements can be connected to the network using the same interface.

Configuration

We have talked about the flexibility and highly customizable configuration of the Tejas simulator, which is contained basically in two files:

- *XML configuration file*: it contains all the parameters relative to the emulator and the simulator. Regarding the emulator, the region of interest (ROI) can be configured, as well as paths to libraries. For the simulation engine, all parameters referring to cores, caches, NoC type and routing algorithm, energy values, etc. can be set.
- *Topology file*: it is a plain text file which contains the dimensions of the NoC, as well as the organization of the different components.

3.4 Modeling KNL in Tejas

This section covers the approach we followed to implement the KNL architectural model in Tejas. This approach can be extrapolated to other architectures featuring Intel Mesh Interconnect such as Cascade Lake. We particularly focus on the distinguishing features of the MI: the 2D structure with tiles integrating cores and CHAs, employing a distributed directory for inter-tile coherence, and with distributed memory interfaces. The Tejas extension described in this section has been made publicly available in [48].

Each of the following subsections focuses on one relevant aspect of the implementation, describing the modifications applied to the simulator using a high level of abstraction.

3.4.1 Tiles and cores

A tile is essentially a wrapper of cores, VPUs, caches and the CHA (see Figure 2.2). We implement that abstraction as two cores, a private L1 instruction and L1 data cache for each core, a shared L2 and a CHA (described in more detail in Section 3.4.2) connected to the same router, and therefore sharing the same logical position in the network. The detailed Tejas configuration for each tile is shown in Table 3.1. We did not make any structural changes in the Tejas' pipeline implementation, as in this study we were focusing on modeling the memory system and the interconnection network.

Table 3.1: Tejas configuration for modeling tiles in the KNL architecture [62].

	Tejas structure	KNL parameters	Details
L1	Size	32 KiB	2 read ports
	Associativity	8-way	1 write port
	Write mode	Write-back	
	Latency	4 cycles	
L2	Size	1 MiB	
	Associativity	16-way	
	Write mode	Write-back	
	Latency	13 cycles	
TLB	DTLB	256 entries	1-cycle access
	iTLB	48 entries	4-cycle penalty
Pipeline buffers	ROB	72 entries	
	Memory reservation station	12 entries	
	LSQ	12 entries (load buffers) 16 entries (store buffers)	
Integer unit	ALU	1 cycle	2 units 12-entry RS
	Mult	4 cycles	
	Div	20 cycles	
FP unit	ALU	2-3 cycles	2 VPUs per core 20-entry FP operation RS
	Mult	6 cycles	
	Div	25 cycles	
Branch predictor	Mode	GSkew	
	BHR	12 bits	
	Miss penatly	11-13 cycles	

3.4.2 Memory system

The memory subsystem has suffered the most important modifications with respect to the original version of Tejas. KNL introduces: 1) different memory domains, 2) a different cache coherence protocol (MESIF), and 3) a distributed cache directory (CHA). Therefore, in the simulator we must be able to control the access to the different domains, modify the transitions between cache line states, and reimplement the cache directory in a distributed fashion.

Accessing different memory domains

In KNL, programs can choose from two different memories to allocate data when in Flat mode: MCDRAM or DDR. Memory addresses above `0x3040000000` lie on MCDRAM, while addresses below that are bound to the DDR system. However, Tejas works with virtual memory addresses only, making it difficult to distinguish between different memory domains.

Using configuration files, we provide the ranges of physical memory that correspond to different memory domains. During emulation, we instrument memory allocations and find the mapping between virtual and physical pages. The virtual-to-physical page translation is passed to the simulation engine, in charge of the architectural modeling. When an LLC miss occurs, the simulation engine will employ the appropriate memory controller to fulfill the request.

MESIF protocol

The main difference between MESI (included in Tejas) and MESIF is the inclusion of an F (Forward) state. As mentioned in Section 2.2.2, a cache in F state will be in charge of serving requests. The requestor will acquire the block in F state and the sender will change to S (shared) state. This avoids all sharers of a cache line responding in the interconnection network.

Figure 3.5 illustrates how the MESI state diagram is modified to include the new F state. From the implementation point of view, this requires changes to the `CacheLine` class in Tejas, which now stores the forwarder CHA in addition to all sharers.

Caching/Home Agent (CHA)

The original Tejas implementation of a centralized tag directory requires it to be placed in a fixed location in the mesh. Unlike Tejas, the MI architecture features a distributed system in which each CHA holds a portion of the full directory. We have implemented a `CHA` class as an extension of the `Directory` class, which is modified

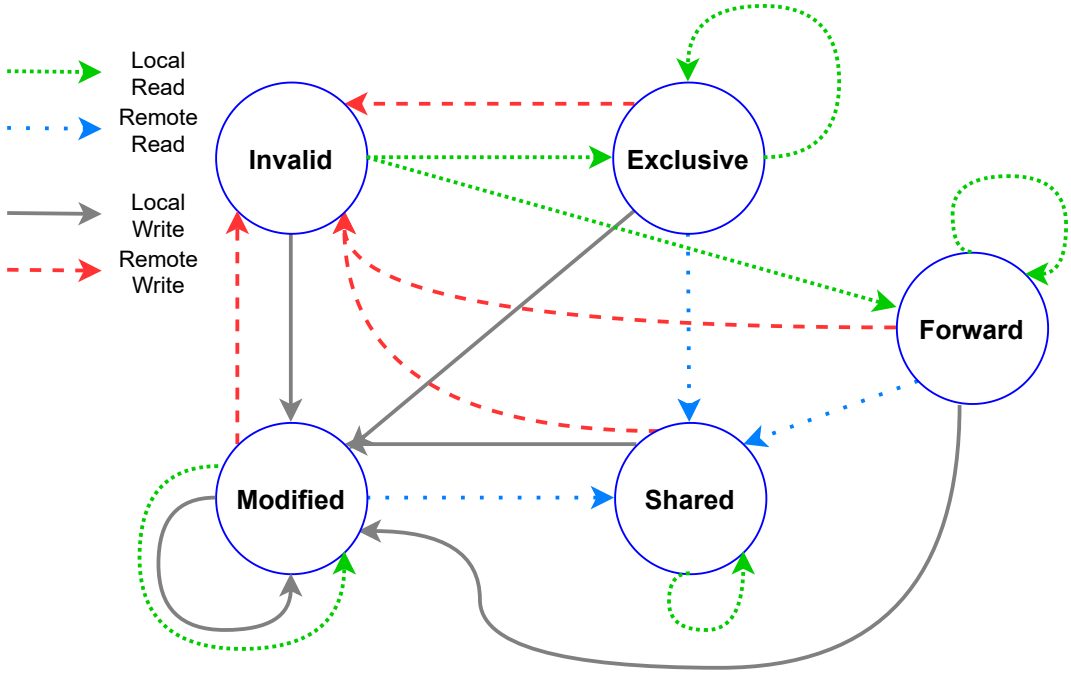


Figure 3.5: MESIF protocol implementation based on MESI.

to include cache lines in Forward state. This new component is replicated once per tile. The LLC on each tile will send a miss request to its own CHA, which will respond if the data is locally available or will forward the request to the appropriate CHA for resolution. Each CHA holds up to 65,536 entries.

In order to accurately simulate the distribution of memory lines across CHAs, we have used the block-to-CHA mapping uncovered in Section 2.3. We could have also used the equations reverse engineered in Section 2.6, but we use instead the pre-computed mapping to accelerate the simulation avoiding those computations.

3.4.3 Interconnection network

Although Tejas already implements a 2D mesh, it does not include a YX routing option such as the one used by the MI. Our implementation is basically a modification of the already included XY routing. All parameters are described in Table 3.2. For Tejas we modeled the routers to have 4 ports, even though theoretically we should also put the port from the tile to the router, i.e., the local connection [110],

Table 3.2: Tejas configuration for modeling the KNL NoC.

Tejas structure	KNL parameters
Topology	2D mesh
Number of ports	4
Buffer size (port)	4
Routing algorithm	YX
Latency	2 cycles (x-direction)
	1 cycle (y-direction)

but that one is implicit in our implementation. The NoC in Tejas is a packet-switched network. As such, we cannot specify the size of the flit FIFOs on each port, only the number of maximum packets. In our simplification, we modeled the routers to use 4-size packet FIFOs.

Furthermore, in Tejas the NoC decides which memory controller is chosen when an LLC miss occurs. Our implementation delegates this decision to the CHA containing the coherence information, as per Intel's design.

3.4.4 Other considerations

Note that not all of the architectural aspects of KNL have a direct translation to Tejas objects. For instance, Tejas employs a 9-stage pipeline, whereas KNL cores feature 5-stage pipelines. Additionally, Tejas does not support vectorization simulation.

These shortcomings can make the simulation of the cores' behavior inaccurate, but note that our main goal is the analysis of the traffic on the interconnect network and the behavior of the memory system. As such, our simulator constitutes a very powerful tool for the architectural and behavioral analysis of the MI.

3.5 Validation

We validated our KNL model in Tejas by comparing its behavior against a real KNL system. We have chosen two benchmark suites: PolyBench/C [103] and Parboil [121]. Both of them include benchmarks from very different domains, such as

linear algebra computations, image processing, physics simulations, dynamic programming, or data analytics. These algorithms are widely used in both scientific and industrial applications. The main difference between both suites is that PolyBench/C benchmarks are polyhedral, originally single-threaded kernels, whereas Parboil includes more complex multithreaded applications. Using this mix allows us to analyze the accuracy of a single core working alone with the memory system, and also how the results vary when the traffic density in the system increases as more cores become active. This experimental design pursues a wide scope of the validation process.

3.5.1 Experimental setup

Benchmarks were compiled using GNU GCC 4.8.5 with the `-O1` optimization flag. This ensures a good dynamic coverage of binary codes by Tejas, which is reduced when using more recent GCC versions or when enabling `-O2`. As mentioned in previous sections, vectorization is disabled as AVX-512 is not supported by Tejas. The `-static` flag is used to link system libraries statically and improve static coverage. We fixed the core frequency at 1.3 GHz, disabling DVFS in order to minimize experimental variability. The most important KNL configuration parameters are summarized in Table 3.3.

We have used the PAPI/C library for measuring hardware events in the real machine, an Intel Xeon Phi x200 7210 processor. PolyBench/C integrates a set of macros for easily handling the configuration of this library. We have extended these

Table 3.3: Experimental setup for our KNL implementation.

Parameters		KNL	
		PolyBench/C	Parboil
Core	Number	1	64
	Frequency	1.3 GHz	
	Cluster mode	Quadrant	
Memory	Type	MCDRAM (16 GiB)	
	Mode	Flat	
	Page size	4 KiB	
Compiler	Version	GNU GCC 4.8.5	
	Flags	-O1 -static -fopenmp	

Table 3.4: Equivalence between the event to measure, the PAPI event, and the event programmed in Tejas.

Event (V)	KNL PAPI (V_{knl})	Tejas simulator (V_{tejas})
Cycles	CPU_CLK_UNHALTED:THREAD_P	Cycles taken
Instruction count	INSTRUCTIONS_RETIRED	Instructions executed
L1 data accesses (L1a)	MEM_UOPS_RETIRED:ANY_LD MEM_UOPS_RETIRED:ANY_ST	Memory Requests
L1 data misses (L1m)	MEM_UOPS_RETIRED:LD_DCU_MISS	L1[core] Misses
L2 accesses (L2a)	LLC_REFERENCES	L2[tile] Accesses
L2 misses (L2m)	LLC_MISSES	L2[tile] Misses
MCDRAM responses (MC)	OFFCORE_RESPONSE_0:MCDRAM	$\sum_{MC=0}^{7} Responses_{MC}$
MCDRAM far (MCfar)	OFFCORE_RESPONSE_0:MCDRAM_FAR	$\sum_{MC \notin Q(c)}{}^{*} Responses_{MC}$
MCDRAM near (MCnear)	OFFCORE_RESPONSE_0:MCDRAM_NEAR	$\sum_{MC \in Q(c)}{}^{*} Responses_{MC}$

* $Q(c)$ returns the number of quadrant for a core c

macros to use PAPI/C in multithreaded applications [47], in order to conveniently measure events in Parboil codes.

Table 3.4 summarizes the equivalence between the events we want to measure in KNL and Tejas. Some Tejas events have been adapted to those measurable in KNL. For instance, there are no PAPI events or IMC counters in KNL for measuring the number of total MCDRAM accesses *per se*, its equivalence is the number MCDRAM responses to L2 read misses.

In order to test the memory system modeled in Tejas, we have ensured that all the benchmarks handle a workload that does not fit in the LLC, i.e., the footprint of each benchmark must exceed 1 MiB for sequential workloads and 32 MiB for parallel workloads. This guarantees that all components of the memory hierarchy play a part in the compound behavior of the system.

3.5.2 Results

We present the results for each event using two different metrics. The first one, shown below in Equation 3.1, is the relative error of the Tejas simulation with respect to the performance counters read in the KNL system (which will be referred to as REL). This metric has the disadvantage of not contextualizing the error in the scope of the execution, as the error increases exponentially when the event count tends to zero. For example, if a benchmark only makes a few hundred accesses to

main memory, then a difference of tens of accesses will cause a large relative error, but this may have almost no effect in the context of the full execution if the data being read from memory is then heavily reused. In order to better contextualize errors, we also provide the number of errors per memory instruction, as shown in Equation 3.2 (which will be referred to as INS). This metric has advantages for our purposes: it is zero-centered and it does not suffer exponential variations for linear changes in its inputs. Both metrics become zero when the simulation exactly matches the behavior of the real system, and they are positive when the simulation overestimates a parameter and negative when it is underestimated. Note that both metrics become the same for the L1a event (number of L1 data accesses).

$$\forall k_i \in V_{knl}, \quad t_i \in V_{tejas}$$

$$REL_i = \frac{t_i}{k_i} - 1 \tag{3.1}$$

$$INS_i = \frac{t_i - k_i}{k_{L1a}} \tag{3.2}$$

In order to account for experimental variability we have executed each benchmark 10 times. The mean for each measured event is reported. Furthermore, when analyzing the results of multithreaded workloads we report the mean values across all cores.

PolyBench/C results

Figure 3.6 shows the number of errors per memory instruction (INS metric) for the PolyBench/C codes. The maximum error found is 2.98 errors per each 100 memory instructions for the L2 accesses of the `syr2k` benchmark. For all events the mean error values are at or below one error per each 500 memory accesses. These results demonstrate that the simulated system is closely capturing the behavior of the real system, and the largest source of inaccuracy is the behavior of the local caches, which is known to be hard to simulate as it depends on other factors such as the interference or noise caused by the operating system, opaque techniques like hardware prefetching, etc.

Table 3.5 displays the results of Figure 3.6 (INS metric) in numerical form, together with the relative error (REL metric). As can be seen, sporadic high relative

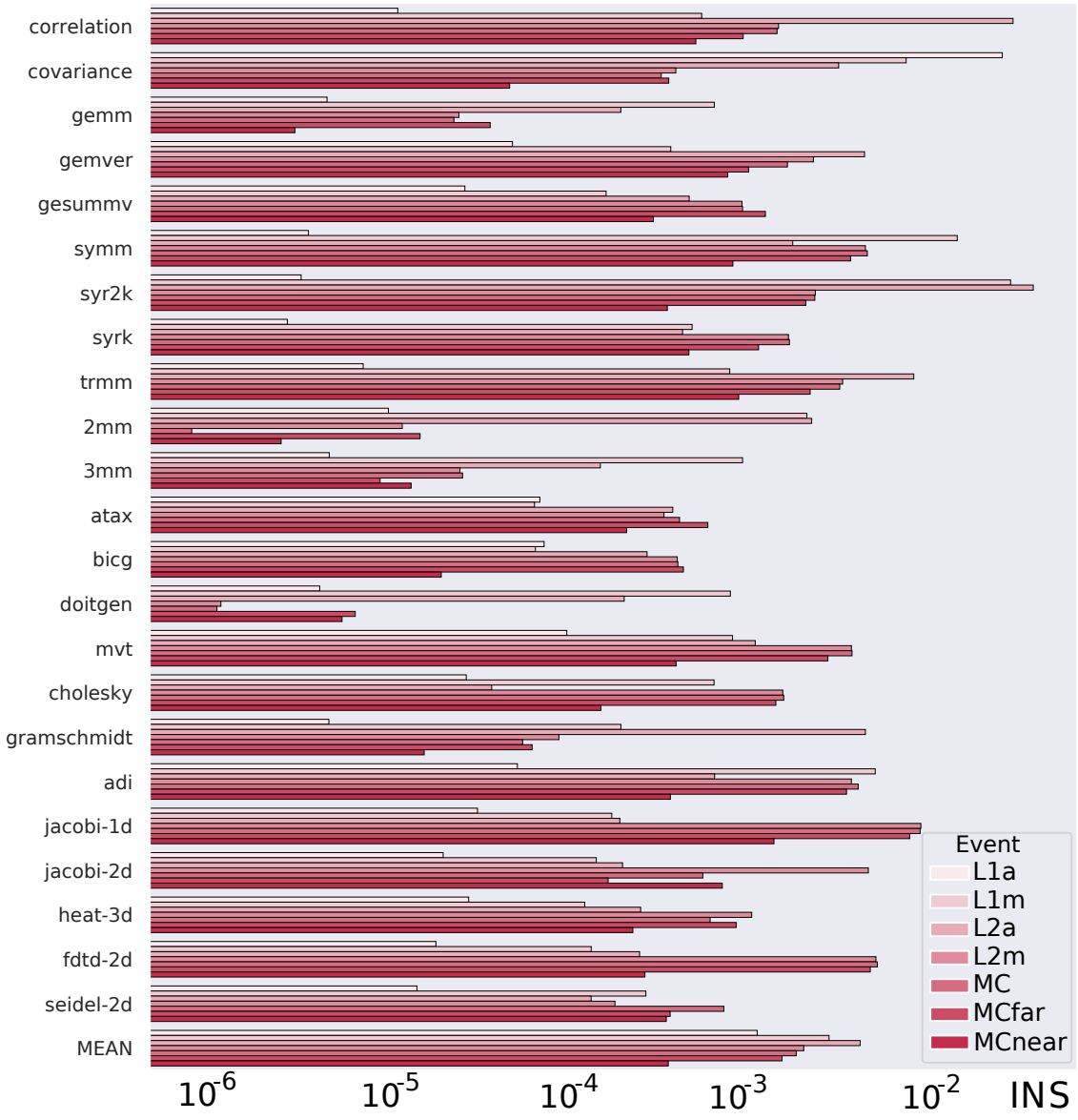


Figure 3.6: *INS* error metric for PolyBench/C benchmarks (logarithmic scale).

error values are much lower when put in context by the number of total memory accesses. For instance, the 30.57% relative error achieved for the number of L2 misses of `trmm` might seem high, but it only accounts for 300K misses out of 56.8M total L2 accesses, i.e., the difference in global L2 miss rate is only 0.5%. High relative errors usually occur for benchmarks with a memory footprint very close to the total available cache space. The simulator tends to underestimate the number of misses in this case, as it does not take into account the impact on cache occupancy of small low-level routines, such as those being run by the operating system.

The results presented in this section do not include the values for execution cycles and instruction count. The simulation of these events turns out to be very inaccurate with respect to the real execution, achieving relative errors above 50%. This is to be expected, since Tejas is not simulating the execution pipeline but translating CISC instructions to the Tejas ISA (VISA), and this is not the focus of our simulation.

Nonetheless, what is remarkable in our work is the accuracy in terms of the memory system. Figure 3.6 illustrates the good results obtained for these benchmarks. The maximum observed error is around 3.5% for L2a in a benchmark, however almost all measurement errors are below 0.1% using our metric. This means that for every +1000 memory requests there is 1 incorrect at most, which is a very low rate. As we can see, L1a error is negligible, as well as the L1m. However, the error increases a bit for L2a and L2m, propagating the error to the MCDRAM accesses either near and far (MC, MCnear and MCfar). Having said this, the errors obtained are low and very acceptable. The main source of the increase in the error rate as we go down the memory hierarchy is that the number of accesses decreases by an order of magnitude in level $L+1$ with respect to level L. As such, the same difference in absolute terms in L and $L+1$ becomes higher in relative terms for level $L+1$.

Table 3.5: *REL* and *INS* error metrics for each event analyzed in the PolyBench/C kernels. The mean absolute value of all benchmarks for each event and metric is shown in the last row.

kernel	L1a		L1m		L2a		L2m		MC		MCfar		MCnear	
	REL	INS	REL	INS	REL	INS	REL	INS	REL	INS	REL	INS	REL	INS
correlation	$\simeq 0$	$\simeq 0$	0.0009	0.0004	-0.0454	-0.0231	-0.1137	-0.0012	-0.1118	-0.0012	-0.0968	-0.0007	-0.1593	-0.0004
covariance	0.0201	0.0201	0.0121	0.0059	0.0051	0.0025	0.0358	0.0003	0.0296	0.0003	0.0432	0.0003	-0.0174	$\simeq 0$
gemm	$\simeq 0$	$\simeq 0$	0.0211	0.0005	0.0063	0.0002	0.0753	$\simeq 0$	0.0703	$\simeq 0$	0.1572	$\simeq 0$	0.0389	$\simeq 0$
gemver	$\simeq 0$	$\simeq 0$	0.0034	0.0003	-0.0384	-0.0035	-0.1095	-0.0018	-0.0812	-0.0013	-0.0659	-0.0008	-0.1497	-0.0006
gesummv	$\simeq 0$	$\simeq 0$	-0.0042	-0.0001	-0.0119	-0.0004	0.0241	0.0007	0.0243	0.0007	0.0434	0.0010	-0.0314	-0.0002
symm	$\simeq 0$	$\simeq 0$	0.0415	0.0114	-0.0049	-0.0014	0.2466	0.0035	0.2541	0.0036	0.2734	0.0029	0.1837	0.0007
syr2k	$\simeq 0$	$\simeq 0$	-0.1890	-0.0224	-0.2370	-0.0298	0.1352	0.0019	0.1341	0.0019	0.1603	0.0017	0.0830	0.0003
syrk	$\simeq 0$	$\simeq 0$	0.0020	0.0004	-0.0017	-0.0003	-0.2331	-0.0013	-0.2353	-0.0013	-0.2142	-0.0009	-0.2657	-0.0004
trmm	$\simeq 0$	$\simeq 0$	0.0013	0.0006	-0.0130	-0.0065	-0.3057	-0.0026	-0.2980	-0.0026	-0.2768	-0.0018	-0.3305	-0.0007
2mm	$\simeq 0$	$\simeq 0$	-0.0566	-0.0017	-0.0598	-0.0018	0.0235	$\simeq 0$	-0.0016	$\simeq 0$	0.0394	$\simeq 0$	-0.0201	$\simeq 0$
3mm	$\simeq 0$	$\simeq 0$	0.0029	0.0007	-0.0005	-0.0001	-0.0452	$\simeq 0$	-0.0467	$\simeq 0$	-0.0220	$\simeq 0$	-0.0977	$\simeq 0$
atax	-0.0001	-0.0001	0.0034	0.0001	-0.0191	-0.0003	0.0177	0.0003	0.0217	0.0003	0.0414	0.0005	-0.0441	-0.0002
bicg	-0.0001	-0.0001	0.0034	0.0001	-0.0138	-0.0002	0.0210	0.0003	0.0212	0.0003	0.0303	0.0004	-0.0042	$\simeq 0$
doitgen	$\simeq 0$	$\simeq 0$	0.0207	0.0006	0.0053	0.0002	0.0025	$\simeq 0$	0.0024	$\simeq 0$	0.0185	$\simeq 0$	-0.0469	$\simeq 0$
mvt	-0.0001	-0.0001	0.0046	0.0007	-0.0061	-0.0009	0.1463	0.0030	0.1476	0.0030	0.1419	0.0022	0.0613	0.0003
cholesky	$\simeq 0$	$\simeq 0$	-0.0159	-0.0005	-0.0010	$\simeq 0$	0.0867	0.0012	0.0878	0.0013	0.1060	0.0011	0.0345	0.0001
gramschmidt	$\simeq 0$	$\simeq 0$	-0.0006	-0.0002	-0.0134	-0.0035	0.2682	0.0001	0.1540	$\simeq 0$	0.2450	0.0001	0.1937	$\simeq 0$
adi	$\simeq 0$	$\simeq 0$	-0.0634	-0.0040	-0.0041	-0.0005	0.0845	0.0030	0.0928	0.0032	0.1062	0.0028	0.0341	0.0003
jacobi-1d	$\simeq 0$	$\simeq 0$	-0.0056	-0.0001	-0.0031	-0.0002	0.1670	0.0072	0.1651	0.0071	0.1942	0.0062	0.1032	0.0011
jacobi-2d	$\simeq 0$	$\simeq 0$	-0.0059	-0.0001	-0.0042	-0.0002	0.1044	0.0037	-0.0128	-0.0004	0.0051	0.0001	-0.0653	-0.0006
heat-3d	$\simeq 0$	$\simeq 0$	-0.0024	-0.0001	-0.0037	-0.0002	0.0300	0.0008	0.0178	0.0005	0.0330	0.0007	-0.0266	-0.0002
fdtd-2d	$\simeq 0$	$\simeq 0$	-0.0019	-0.0001	-0.0034	-0.0002	0.0742	0.0040	0.0756	0.0041	0.0919	0.0038	0.0158	0.0002
seidel-2d	$\simeq 0$	$\simeq 0$	-0.0186	-0.0002	-0.0094	-0.0001	0.0130	0.0001	-0.0519	-0.0006	-0.0350	-0.0003	-0.1000	-0.0003
MEAN	0.0009	0.0009	0.0209	0.0022	0.0222	0.0033	0.1027	0.0016	0.0929	0.0015	0.1061	0.0012	0.0916	0.0003

Parboil results

Among the different benchmarks included in the Parboil suite, we have selected those with a significant number of accesses to main memory with respect to the number of instructions executed, in order to stress the simulator's memory system and interconnection network, as more data and coherence traffic is generated. Figure 3.7 shows the boxplots of the number of errors per memory instruction (*INS* metric) for these benchmarks. Boxplots are used instead of static bars, as the benchmarks are multithreaded and there is a different number of errors for each simulated core. Table 3.6 shows the relative (*REL*) and normalized (*INS*) errors expressed as the mean of all cores. In general, both metrics increase with respect to the single-threaded PolyBench/C benchmarks. The reason is that the number of instructions, being

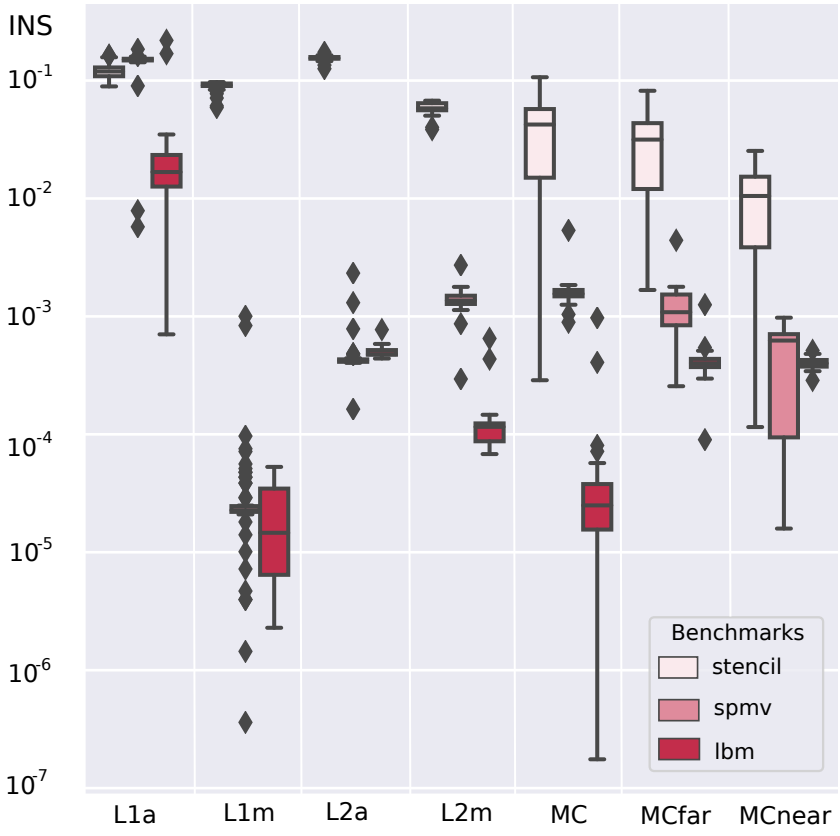


Figure 3.7: *INS* error metric for Parboil benchmarks. Boxplots represent the variability of the results obtained by each core (logarithmic scale).

Table 3.6: *REL* and *INS* error metrics for each event analyzed in the Parboil benchmarks expressed as the mean of all cores.

kernel	L1a		L1m		L2m	
	REL	INS	REL	INS	REL	INS
stencil	0.1137	0.1137	-0.2747	-0.0905	-0.3679	-0.0593
spmv	0.1459	0.1459	-0.0010	$\simeq 0$	0.0584	0.0014
lbm	0.0178	0.0178	0.0016	$\simeq 0$	-0.0038	-0.0001
kernel	MC		MCfar		MCnear	
	REL	INS	REL	INS	REL	INS
stencil	0.9477	0.0378	0.9501	0.0284	0.9479	0.0095
spmv	0.0687	0.0016	0.0696	0.0012	0.0693	0.0004
lbm	-0.0003	$\simeq 0$	-0.0005	$\simeq 0$	0.0001	$\simeq 0$

divided across 64 threads, is now much smaller, and the errors introduced by the emulation system represent a larger share of the total. Note how the most significant distortion in the results corresponds to the number of L1 accesses, which is completely dependent on the way in which the emulator translates CISC instructions into VISA in multithreaded codes. This in turn results in larger errors in L1 misses and, therefore, L2 accesses. The memory hierarchy gradually "absorbs" these errors, and results become more accurate as we descend towards slower memories.

3.6 Case Study: Analysis of Coherence Traffic Optimizations

In Chapter 2, Section 2.3 described how the location of the coherence data in the mesh is of great importance for memory latency in this architecture. As depicted in Figure 3.1, the time to access a memory line that resides in a nearby tile can be high if the coherence data is stored in a farther one. We proposed different techniques in Sections 2.7 and 2.8 to alleviate this effect based on the maximum distance between cores and the coherence data. This optimization, however, shows a complex trade-off between different factors such as the number of TLB misses, the reduction in access latencies, and the contention on different areas of the mesh.

We modified a `jacobi-1d` stencil from the PolyBench/C suite so that cores i and $i+2$ swap their data at the end of each timestep. The reason is to reuse data from

adjacent threads in order to quantify the impact of the physical distance between a requestor (core) and the responder (CHA). We used two different maximum core-to-CHA distances, 16 and 0, so that all cores write to memory lines whose coherence information resides in any tile in the mesh (maximum distance = 16) or in their local tile (maximum distance = 0). We then ran the experiment using 64 threads and two different mappings: scattered and co-located. The first one is a direct thread-to-core mapping, i.e., thread i is assigned to core i. Since cores are cyclically distributed among the quadrants (as previously described in Chapter 2 and shown in Figure 2.4), cooperating threads will be scattered across the mesh. In the second mapping we take advantage of the reverse-engineered floorplan in Figure 2.4 to optimally co-locate cooperating threads in neighboring tiles.

We analyzed the behavior of the benchmark for both mappings and both core-to-CHA distances (16 and 0), measuring different events with performance counters. Results are shown in Table 3.7. As can be seen, setting the maximum core-to-CHA distance to zero provides a performance benefit of 6.25%, and controlling the location of threads using the co-located mapping increases this benefit further to 7.76%. However, simply controlling the location of threads on the mesh, but not the core-to-CHA distance has a negligible effect. The counters offer no evidence of any significant difference in cache misses or memory accesses which may account for the performance difference between the configurations mentioned above. On the contrary, the number of memory accesses increases for the fastest configurations (those with zero core-to-CHA distance), due to the increase in TLB misses. We

Table 3.7: Selected events for the executions of the modified `jacobi-1d` stencil by thread 0. The rows present two different maximum core-to-CHA distances (16 and 0 cycles). The columns show the counters being measured and the thread mapping employed for each execution. Values are reported in millions.

distance	cycles		L1m		L2m	
	scat	co-loc	scat	co-loc	scat	co-loc
16	84.127	83.986	0.571	0.571	0.561	0.561
0	78.865	77.597	0.571	0.571	0.569	0.581
distance	MC		MCfar		MCnear	
	scat	co-loc	scat	co-loc	scat	co-loc
16	0.019	0.016	0.014	0.013	0.005	0.004
0	0.053	0.043	0.023	0.016	0.022	0.026

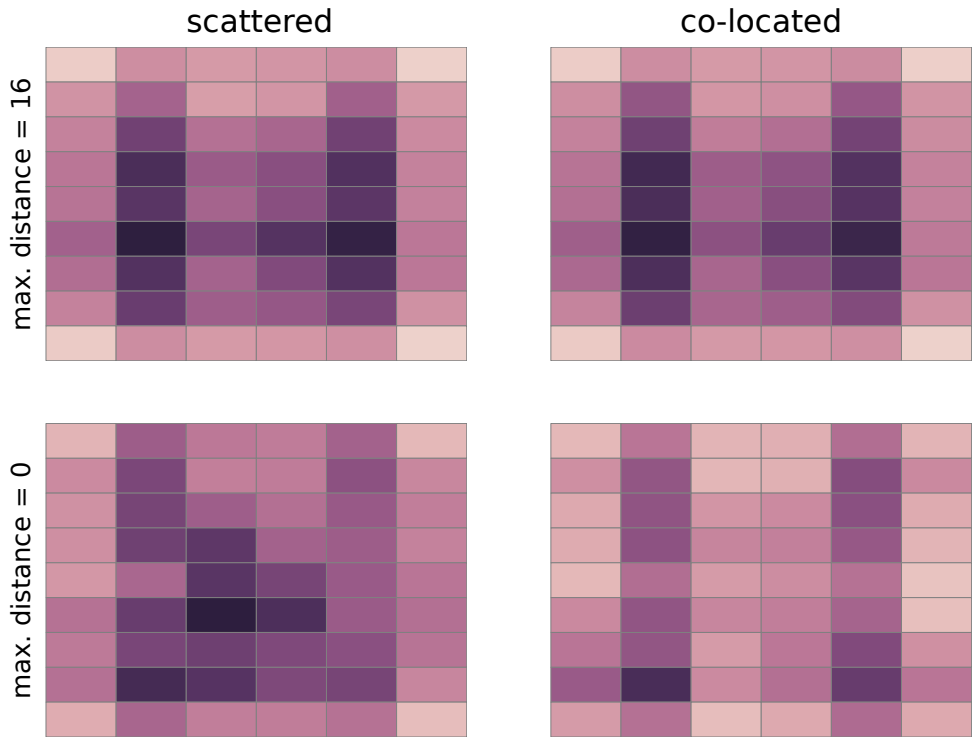


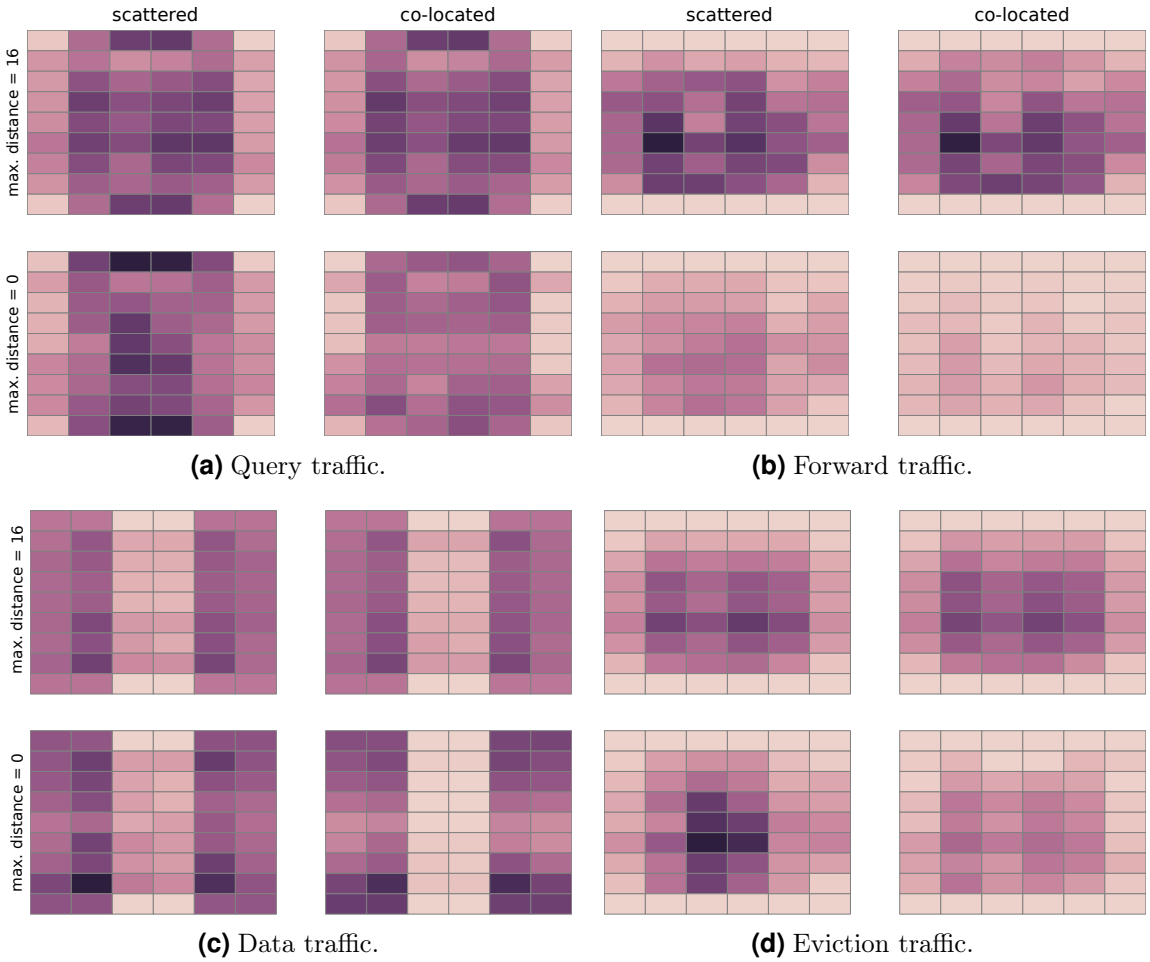
Figure 3.8: Heatmap of the number of packets across the mesh for two different core-to-CHA distances and thread mapping. When the maximum distance is 16 cycles, a block assigned to a core can have its coherence data residing in any tile across the mesh. On the other extreme, when the maximum distance is 0 cycles, the coherence data of the block must reside in its local tile. Darker shades indicate higher traffic density.

need to analyze the low-level traffic of the mesh in order to understand the reasons for the difference in performance.

The idea behind these optimizations is to eliminate the majority of the coherence traffic through the mesh. If most of the blocks requested by a core have their coherence data stored in the local CHA, the coherence traffic should be significantly reduced. We employed our simulator to analyze the traffic passing through each router, as shown in the heatmap of Figure 3.8. The total reduction in the number of packets traveling through the network is 18.7% when optimizing both the location of the coherence data (i.e., maximum core-to-CHA distance = 0) and the mapping of the cooperating threads (co-located mapping). The reduction in total traffic might be surprising, taking into account that the number of L2 misses increases very

slightly. In order to understand the low-level behavior of the system, we studied how the reduction in traffic is distributed across the four different types of packets: *query* (requests for data directed to a CHA), *forward* (a request from a CHA to an L2 cache or MCDRAM interface to forward a block to another L2), *data* (a response to a forward request containing the requested block), and *eviction* (a write-back message from an L2 cache to an MCDRAM interface containing an evicted data block). Data packets are not reduced, as we are not changing cache locality whatsoever. Query data is reduced by 20.4% and forward data by 343%. Furthermore, the average time from the generation of a query packet to the delivery of the corresponding data packet is reduced from 76.84 to 50.87 cycles using both optimizations (maximum distance 0 and co-located mapping), i.e., the number of hops performed by each coherence packet is drastically reduced. Note that an L2 miss being resolved in its local CHA will generate a query packet and a data packet in the local router, but no forward packets. This explains the significant reduction in forward traffic as compared to query traffic.

Figure 3.9 details the breakdown of the network traffic into each packet type. The drastic reduction in the forward traffic is clear from the figure. It can be observed how data traffic across different quadrants is almost nonexistent when applying both optimizations. The eviction traffic in the central region of the mesh is similarly reduced. The reason is that each tile works now with data residing in the local MCDRAM interfaces of its quadrant, and therefore most eviction packets will not cross inter-quadrant boundaries. As such, traffic density is higher towards the center of each quadrant, instead of towards the central region of the mesh, as it happens with no optimizations. This is a result of the YX routing protocol: a drastic reduction in the number of collisions, i.e., situations in which a packet is stalled while in transit to its destination because the next router in its path has no available buffer space (4 buffers per port/router, see Table 3.2). Figure 3.10 shows the collision density on the NoC. Again, the collisions are mostly confined to the center of each quadrant when applying the optimizations. The total number of collisions shows a reduction of 76.1%. The number of collisions per hop is reduced from 0.18 in the original code to just 0.05 in the optimized version.



(a) Query traffic. **(b)** Forward traffic.

(c) Data traffic. **(d)** Eviction traffic.

Figure 3.9: Breakdown of the different packet types across the NoC. Darker shades indicate higher traffic density.

3.7 Related Work

Several papers have explored the performance of the Knights Landing architecture, mainly through the analysis of well-known benchmarks, machine learning applications, and parallel workloads [15, 18, 35]. This type of works analyze the scalability of the processor and provide the observed trends in terms of performance of real workloads, which are then compared against the theoretical performance. Furthermore, these works provide a reference when comparing this architecture with others. Nevertheless, none of these works undertake the analysis of the particular characteristics of the KNL interconnect.

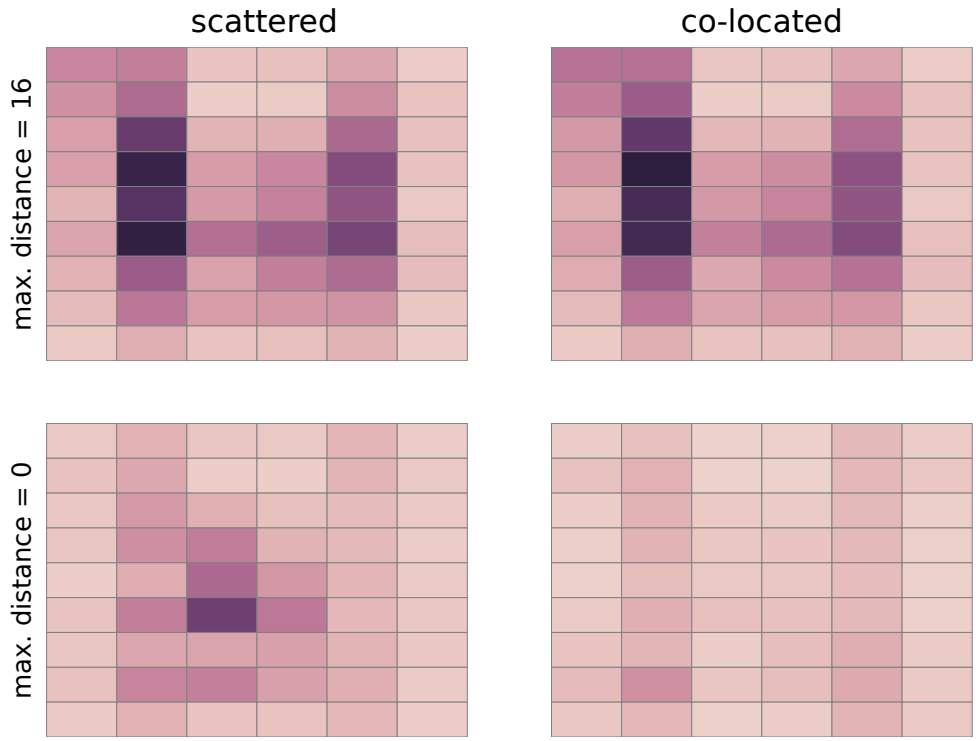


Figure 3.10: Density of collisions on the network. Darker shades indicate higher number of collisions.

Other papers offer a more in-depth look at the KNL internals. Rho et al. [106] study and compare the behavior of each of the offered cluster and memory modes by analyzing the behavior of MPI workloads. They propose an approach to optimize the scheduling at different granularities dynamically based on the characteristics of each workload. McCalpin [85] studies the degradation in performance due to snoop filter evictions, resulting from associativity conflicts in the snoop filters. These conflicts are traced to interactions of data physical addresses with the hash function distributing addresses across the agents. This degradation is also relevant for our approach, as the simulator does not consider this issue. It can also be a source of error when comparing to real hardware in our model, and it should be considered for increasing the accuracy of the model.

Other works have proposed ways to discover architectural features, or to automatically tune applications in modern, highly complex systems. Yotov et al. [137] develop a set of micro-benchmarks specifically designed to measure memory hierarchy parameters, such as cache associativity, block size, capacity, or TLB parameters.

Wang et al. [128] identify the increase in complexity associated with modern computational systems, in particular the trend to include a very large number of different architectural configurations. They argue that static discovery of optimal configuration parameters is a fundamentally flawed approach, and propose a configuration interface to allow users to specify performance constraints that should be satisfied at runtime. Mishra et al. [92] propose to use an automatic learning system to manage resources towards meeting specific latency and energy constraints. The resource allocation is performed in two different steps: learning how allocation affects parameters and controlling them during runtime.

Finally, there are many architectural simulators available nowadays. Some of them have very detailed pipelines, such as gem5 [11], some others are able to simulate up to thousands of cores, such as ZSim [111], PrimeSim [34], Graphite [91], McSimA+ [6] or Sniper [16]. In this work we have chosen to build upon the Tejas simulator due to the convenient compromise between the pipeline detail, the scalability of the simulation, and the low-level implementation of the distributed cache directory for cache coherence. Furthermore, it is an actively developed software project with a very responsive community.

3.8 Conclusions and Future Work

In this chapter we have analyzed in detail the Intel Mesh Interconnect architecture through the study of the Intel Knights Landing. Leveraging the knowledge presented in Chapter 2, we have created an architectural model and implemented it on the Tejas simulator, which enables an in-depth analysis of the behavior of this highly complex NoC. We validated our KNL model in Tejas by comparing its behavior against a real KNL system using the PolyBench/C and Parboil benchmarks. More specifically, we have validated the implementation of the memory system in terms of mispredictions against the number of accesses and misses in the memory hierarchy, including the MCDRAM. The results obtained present a good ratio for the relative metric *INS* presented in Section 3.5.2, which indicates the number of errors per memory instruction. For single-threaded PolyBench/C benchmarks we obtain almost all measurement errors below 0.1%. For multithreaded Parboil benchmarks the results are not that promising, mainly due to the bad translation from multi-

thread x86 directives to Tejas VISA, resulting in a larger error rate in L1 misses and L2 accesses.

Finally, we have also presented a case study analyzing the low-level behavior of the interconnection network. We modified a `jacobi-1d` stencil from the PolyBench/C suite to exploit mesh locality. Based on the reverse engineered floorplan described in Chapter 2, we ran multithreaded experiments using different affinities for the threads: scattered and co-located. As expected, the co-located approach reduces drastically the contention in the network.

This work can be extended by adapting the simulator to modern x86 manycore architectures, such as the latest Intel Xeon Scalable (Cascade Lake and Ice Lake architectures) and AMD EPYC (Zen3). This extension should also improve the micro-architectural details, in order to fully support, for instance, vector instructions (e.g., AVX2). In the same line, it would also be required to extend the Tejas VISA, in order to improve the coverage of multithreaded applications, and reduce the errors in the simulation. On the other hand, Tejas is quite limited in that regard as it deliberately avoids considering all x86 instructions to keep the design simpler.

"Computers are good at following instructions, but not at reading your mind."

–Donald Knuth

4

MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis

Chapter's contents

In this Thesis we explore and develop SIMD optimizations, as it will be described in Chapter 5. This type of optimizations is present in all modern compilers, guided by cost models which determine the profitability of vectorizing certain regions of code. The result of them depends not only on the code, but also on the architecture and the flags used. Because of this, compiler performance benchmarking is an error-prone, tedious and repetitive task. Configuring a new experimental environment

frequently requires writing new scripts or ad hoc programs in order to correctly and properly instrument specific regions of code and hardware events. These artifacts are normally neither reusable nor maintainable, since they are dependent on specific problems and, in some cases, platforms. In order to avoid repeating efforts and help building performance cost models, we have developed MARTA (Multi-configuration Assembly pRofiler and Toolkit for performance Analysis). It is a fully customizable toolkit that aims to increase productivity by generating benchmark templates, compiling them, and monitoring the regions of interest (ROI) specified, as well as performing static code analysis. MARTA's approach may be applied to existing kernels or applications, and it only requires to write a simple configuration file. MARTA uses the PAPI/C library for instrumenting the code and accessing hardware counters in the host platform. Furthermore, the toolkit integrates fine-grained directives for instrumenting and monitoring small regions of code, enabling micro-benchmarking analysis. In an orthogonal dimension, the system is able to run data analytics on these performance measurements. For this purpose, the toolkit applies data mining and machine learning or AI-based techniques for classification and regression, automatically extracting the features of the experimental setup which have the most impact on performance, given a large set of experiments and dimensions to consider. These post-processing tasks are valuable for deriving knowledge from experiments and are not included in most profiling tools. In this chapter we also provide a set of case studies to illustrate the ability of the solution to conveniently create a reliable environmental setup for high performance computing experiments.

4.1 Overview and Motivation

Performance analysis is required in any discipline of computer engineering, for both hardware and software effects. From real-time systems, where it is needed to assess the latencies of instructions, to high performance computing, where codes need to be highly optimized and tuned to the execution infrastructure to extract the maximum possible throughput, profiling is crucial to characterize systems, either in a holistic manner or at a fine-grain level [4, 56].

In order to assess the validity of an experiment, measurements are performed in a platform under a set of conditions. For this reason, measuring events in a system

requires the setup of a controlled environment in order to ensure reproducibility and repeatability for a set of experiments, e.g., setting the scaling governors of processors, setting maximum frequencies of clocks, the memory page size used, core affinities, etc. Additionally, passing arguments, macros and other variables to programs is also an error-prone task, leading to false positives or other undesired and undefined behaviors within the experimental chain. Checking and setting all these conditions and parameters is usually a manual task.

Orthogonally, when profiling a specific region of code or application, there could be different dimensions or parameters of interest for the user to consider and configure, e.g., the number of rows and columns for a matrix-vector multiplication kernel, data precision (32-bits vs 64-bits), number of iterations and step of a loop, access stride, array padding, etc. Analyzing which of these dimensions has the most significant impact in terms of performance can be automated in a similar way, avoiding manual and repetitive post-processing tasks.

We present a toolkit for automating the experimental setup configuration, compilation, execution and collection of data (static and dynamic) for a region of code or application. In addition, using data mining and machine learning or AI-based techniques for classification such as decision trees and random forests, it generates categories for analyzing the influence of parameters and input arguments in the execution of a code. We make the following contributions:

- The design and implementation of MARTA, a framework that fully automates experimental setup, compilation, execution and performance data collection for a computational kernel or application. The system supports the analysis of post-mortem execution data, the static analysis of binaries through LLVM-MCA, and the automated inspection of compilation logs and optimization reports.
- We automate the specialization of template codes and header files including C/C++ macros for efficiently versioning micro-benchmarks, changing compile-time and runtime features such as allocation strides or array padding, or enabling/disabling compiler optimizations such as dead code elimination or loop jamming that interfere with the correct instrumentation of the region of interest.
- We present case studies that illustrate the potential of MARTA in action,

showing how to quickly generate a space of program variants, run them, and analyze the results. We use decision trees for building predictors and visualizing the sub-spaces of interest generated with the performance data collected.

The rest of the chapter is structured as follows: Section 4.2 describes the architecture of the presented framework. Section 4.3 overviews key aspects of the methodology for reproducibility. Section 4.4 explores the configuration parameters available in the toolkit. Section 4.5 presents a set of case studies that exemplify potential uses of the tool. Section 4.6 analyzes the state of the art of profiling and monitoring tools. Finally, Section 4.7 discusses the strengths, capabilities and limitations of the proposed approach.

4.2 MARTA: System's Architecture

The proposed framework is composed of two main modules: the Profiler and the Analyzer. The Profiler is in charge of the compilation, execution and collection of data. The Analyzer inspects the data provided by the Profiler, applying data mining and machine learning or AI-based techniques. Both modules are primarily written in Python 3, C, and Makefile language. The framework works on any operating system, but the compilation and execution facilities are designed specifically for POSIX-compliant systems. MARTA can run on any architecture, the only limitation being the naming of hardware events, specified through configuration files. The high-level architecture of our approach is depicted in Figure 4.1. The two components of the system, Profiler and Analyzer, are independent between them, and can operate autonomously, as they only interface through CSV files containing profiling data. In the rest of the section we describe both modules in detail.

4.2.1 Profiler

The Profiler module is designed for parsing the configuration files, compiling all the binary versions specified in them, and running the generated binaries, collecting execution data. The strength of this module lies in its ability to generate as many different executable versions as necessary, as defined by the Cartesian product of

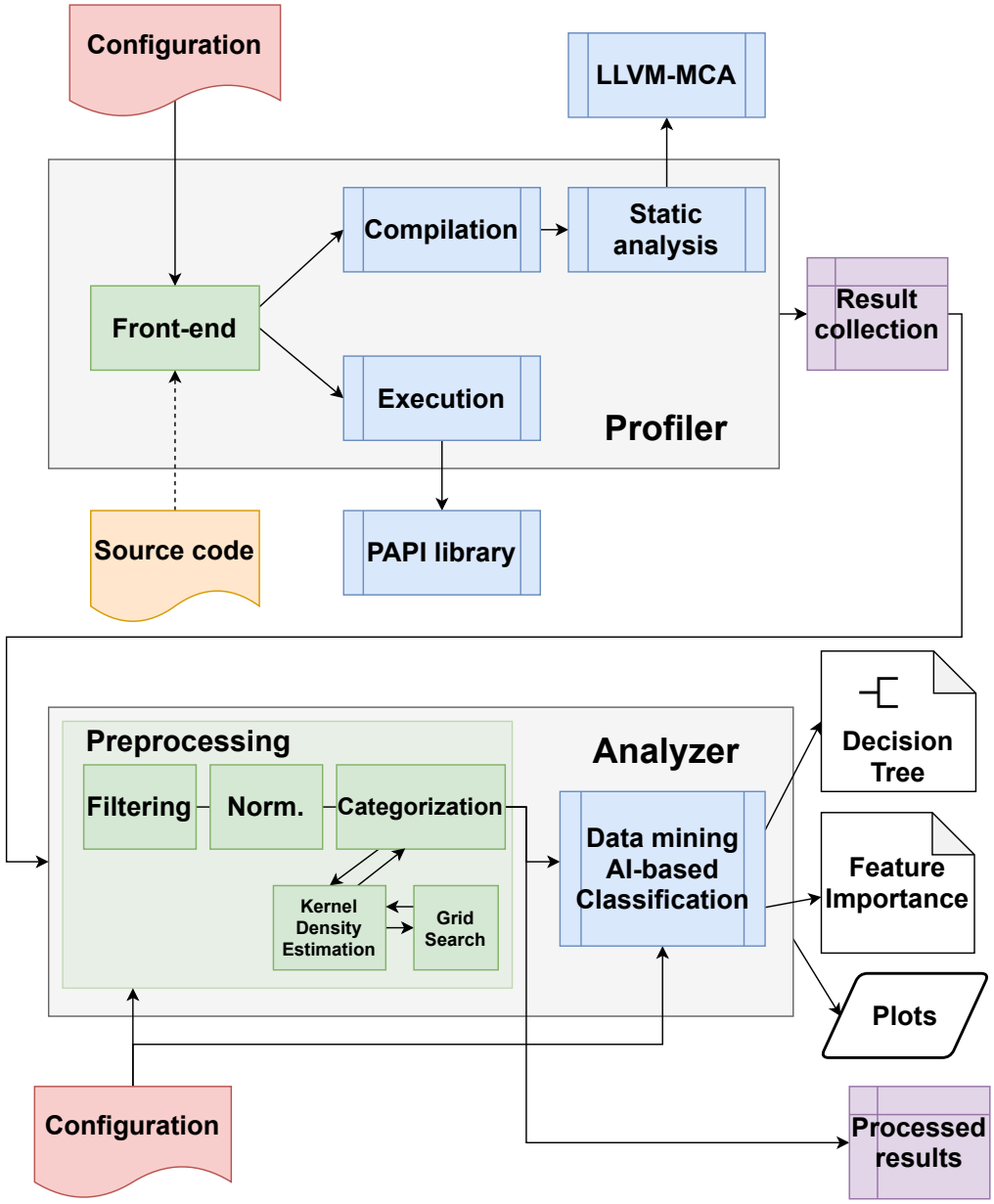


Figure 4.1: High-level architecture of the toolkit, composed of the Profiler and the Analyzer modules, which operate independently.

the sets of different options in the configuration, e.g., compile-time options (e.g., whether to enable or disable particular optimizations), program inputs, or program features (e.g., `-D` flags enabling different code paths). The generation of different program versions, which is often a bottleneck in micro-architectural exploration, can be done in parallel. In order to achieve maximum reliability, the Profiler integrates with several different tested-and-true software packages such as the PolyBench/C library [103], using their low-level configuration and measuring capabilities. The upper part of Figure 4.1 details the design of this module. The Profiler receives two inputs:

- **Configuration file**: a structured YAML file containing all parameters related to compilation (e.g., `-D` flags, compilers and their flags, etc.), execution (e.g., threads to launch and their affinity, number of repetitions, maximum deviation in measurements, etc.), and data collection (e.g., output format, dimensions to include, static code analysis parameters, etc.). For convenience, some of these parameters can be overwritten by using CLI arguments. The format and contents of this file are further described in Section 4.4.1.
- **Source code/application**: typically a C/C++ program whose execution prints in standard output values collected from hardware counters, as well as the execution time and values reported by the Time Stamp Counter (TSC). The system helps to produce this output format by including a set of functions and macros at runtime.

In order to instrument binaries, MARTA follows the steps detailed in Algorithm 4.1. The `execute` function used in that approach is disclosed in Algorithm 4.2. The output generated by all the executions in the experimental setup is encoded into a CSV file, which is passed as input to the Analyzer module.

4.2.2 Analyzer

The Analyzer integrated in the tool is meant for processing raw data, typically the output of the Profiler, and mining knowledge from these data, primarily through the use of scikit-learn [98]. It can also generate relational plots given a set of dimensions of interest. The inputs to this module are:

Algorithm 4.1: MARTA generates different binary versions for collecting the execution time, the TSC value and the PAPI events. Each version is executed `nexec` times. Values that deviate farther than a user-specified threshold from the mean are discarded as outliers.

Input: Executable *binary*, boolean *discard_outliers*, float *threshold*
Result: Dictionary of values

```
values = {};
for type in [TSC, time, PAPI_counters] do
    data = [];
    execute_preamble_commands();
    for i ← 0 to nexec do
        data.append(execute(type, binary));
    end
    execute_finalize_commands();
    if discard_outliers then
        condition = (abs(data - data.mean()) <= threshold * data.std());
        data = data.select(condition);
    avg_value = sum(data) / data.size();
    values[type] = avg_value;
end
return values;
```

- **Configuration file**: a structured YAML file specifying data wrangling parameters (including filtering, normalization and categorization) as well as classification and plotting parameters. For classification customization all parameters follow the same naming or API as in scikit-learn. A full description is provided in Section 4.4.2.

- **Input data**: CSV file labeled according to the dimensions of interest described in the configuration file. The Profiler output is fully compatible with this description, but any CSV file can be used as input to this module.

The preprocessing stage is needed for the classification algorithms to only consider the dimensions of interest, and to know the categories into which data will be classified. Preprocessing is performed as follows:

- **Filtering**: based on the dimensions of interest in the problem, e.g., select columns containing a specific set of values, a range, a concrete value and

Algorithm 4.2: High-level approach of the `execute` function: warm up if specified, and then instruments a number of times the specified region of code.

```
Input: BenchmarkType type, Executable binary
Result: float
if hot_cache then
    for i ← 0 to warmup do
        run_code(type, binary);
    end
v0 = measure(type);
for i ← 0 to steps do
    run_code(type, binary);
end
v1 = measure(type);
return (v1-v0) / steps;
```

discard the rest, etc.

- **Normalization**: values of interest can be normalized using min-max or z-score techniques.

- **Categorization**: dimensions of interest are typically continuous values, e.g., the performance in GFLOPS, or the average of the TSC values reported. The system is able to discretize these values into a collection of bins or categories. This can be configured statically, by describing the number of categories to create in the interval using a constant step, or dynamically, using Kernel Density Estimation (KDE) for guessing the optimal number of categories to generate, as well as their boundaries. For the hyperparameter tuning in KDE grid search is used, Silverman's rule of thumb for normal distributions [118] and Improved Sheather-Jones algorithm [13] for multimodal distributions.

The system randomly splits input data into training and testing subsets, following the Pareto principle or 80/20 rule of thumb, for training classifiers. Currently, the Analyzer implements a decision tree and a random forest classifier. The first one is meant to classify target categories depending on the dimensions of interest specified, and the second one to measure their importance. Adding other classifiers such as SVM, k-means or k-neighbors is trivial thanks to scikit-learn's homogeneous

API. The final output produced by the system is composed of the following optional elements:

- **Classification knowledge**: the system outputs the generated classification model as a decision tree. It also shows the accuracy and the confusion matrix for the model. It is possible to employ `dtreeviz` [97] for improving the visualization of the decision tree.
- **Feature importance analysis**: by applying a random forest classifier, the system is able to extract the impurity-based feature importance. This is computed as the total reduction of the criterion brought by that feature. The system performs feature importance analysis using Mean Decrease Impurity (MDI), which counts the times a feature is used to split a node, weighted by the number of samples it splits.
- **Plots**: it is possible to configure the plotting of different types of graphs: scatter plots, KDE plots, etc.
- **Processed results**: a CSV file, similar to the input, containing the results of these processing steps.

4.3 Measurement Methodology

We briefly overview key features of our measurement methodology. As we particularly target small running times for regions of interest (i.e., micro-benchmarking), we pay special attention to ensure reproducibility by implementing the following principles: 1) the machine shall be configured in a state that can be reproduced (e.g., fixed frequency, thread pinning to cores, aligned memory allocation); 2) each experiment shall be repeated multiple times until a satisfactory confidence on each measurement is reached; and 3) the measurement approach shall be as insensitive to the execution context as possible. We highlight our solutions, inspired by good practice in the field and in particular from the PolyBench/C [103] testing harness system.

Note that conducting experiments correctly so that one can trust in the outcome is a particularly difficult and error-prone process in computer science, due to the

immense variability induced by the execution context and the multiple ways to implement a program that performs a given computation. Blackburn et al. [12] present a framework which provides a high-level checklist for experimenters to use for avoiding unsound claims and properly assess experimental evaluations. This framework emphasizes the correlation of the scope of the evaluation and the scope of a claim in order to make claims sound. In that sense, MARTA can only go as far as automating tasks, but the user remains responsible for correctly setting up the experiments. This includes which knob/s of the experimental setup are fixed or left free such as disabling turbo boost.

4.3.1 Machine configuration

MARTA provides different knobs to control the system that will execute the programs. These include: 1) disabling turbo boost (via MSR); 2) fixing the CPU frequency; 3) pinning threads to particular cores (using OpenMP environment variables or the `taskset` command, and also using the system calls provided by the toolkit directives); and 4) using an uninterrupted process scheduler (the FIFO scheduler). Note that most of these knobs require administrator privileges on the host machine. Turning on all these features would ensure that in between two experiments the effects from the operating system's decisions are mitigated: the frequency is fixed for a concrete core (or set of cores) with the proper thread affinity set, allowing to relate cycles to wall clock time easily and systematically for the whole experiment duration. As an illustrative example, running a DGEMM computation may see a variability of over 20% in terms of cycles between two runs of the exact same software on our testing setup, while this variability reduces to less than 1% with the setup fixed by MARTA.

4.3.2 Repeating runs

A possible approach to increase the confidence in the measured values is to repeat the same experiment multiple times to characterize the variation between runs, and determine whether this variation is acceptable. This is a central aspect of reproducibility. MARTA lets the user determine what is the acceptable variabil-

ity threshold, which depends on the stability of the host machine and how it was configured. It also depends on the data distribution: the variability between runs should be, at least, an order of magnitude lesser than that of the effects that are to be measured.

In MARTA, the default experimental setup is to re-run the same experiment X times, remove the largest and smallest measures from the set (keeping X-2 samples), compute the arithmetic mean of the X-2 samples, and compute the deviation between each sample and the arithmetic mean. If one sample exceeds a threshold T then the whole experiment (the multiple runs of the same program) is discarded, and needs to be repeated. In our tests, we found that X=5 and T=2% are reasonable values for experimental validation (detailed in Section 4.5).

4.3.3 Measuring CPU performance

It is key to ensure that we measure events while understanding their sensitivity to the experimental setup. For example, some hardware counters measure elapsed time (e.g., `CPU_CLK_UNHALTED.REF_P`) while some others are insensitive to frequency and measure elapsed active cycles (e.g., `CPU_CLK_UNHALTED.THREAD_P`). Accurately measuring performance typically involves accurately measuring time. MARTA offers both frequency-sensitive and frequency-independent measurements of time. The number of hardware counters available may be in the hundreds. We preselected in MARTA relevant counters for measuring time, but the user may include other counters to collect data such as data traffic, branch utilization, etc.

Typically processors do not allow to measure more than a handful of counters in the same run in an exact manner. Sampling of the counter value may be implemented instead, and some pairs of counters simply cannot be measured at the same time. To avoid any issue with PAPI counter multiplexing, MARTA performs one experiment per counter to be monitored (exact value, no sampling), running the program with only that counter and the timestamp counter being monitored. For each counter, multiple runs are launched and variability is assessed as described above.

4.4 Configuration

Both the Profiler and the Analyzer require a configuration file to specify the tasks that each of them must perform. These configuration files are simple, yet powerful. For simplicity, these files are written in YAML, which allows to express hierarchical relations with minimal syntax, being easy to read. The system performs sanity checks over the configuration files giving the user feedback about the correctness of parameters declared and/or missing. Sections 4.4.1 and 4.4.2 describe the most relevant available parameters for the Profiler and the Analyzer, respectively.

4.4.1 Profiler

The configuration information for an application to be profiled is divided into four categories, as a nested set of dictionaries:

1. **Metadata, preamble and finalization**: useful for describing commands or predefined actions at the beginning and end of the experimental process. This is useful for tasks such as tuning the processor or host platform, cleaning temporary files, etc.

2. **Configuration**: describes the set of different options and argument values to be used for generating different binary versions, e.g., compile-time options, program inputs, and `-D` flags.

3. **Compilation**: lists the compilers to use for binary generation, as well as specific flags for each of them.

4. **Execution and output**: defines the parameters relative to the execution process, e.g., the number of executions for each binary version, hardware counters to read, the threshold for outlier detection, etc.

Table 4.1 describes the main options for the Profiler, which are described within the `kernel` dictionary. MARTA will generate predefined headers and Makefiles within the `path` directory that are necessary for compilation. One of the main advantages of this format is the ability to generate a vast set of combinations for the

Table 4.1: Description of all available options within each `kernel` dictionary in the configuration file for the Profiler.

Name	Description	Type
`name`	Name of the kernel or program.	str
`path`	Folder containing the sources.	str
`preamble`	Commands to execute before compilation: tuning CPUs, allocating huge pages, etc.	str
`finalize`	Tasks to execute after the experiments. See Table 4.2.	dict
`configuration`	Cartesian product of the list of parameters. This includes Makefile options, `-D` definitions, etc. See Table 4.3.	dict
`compilation`	Compiler configurations. See Table 4.4.	dict
`execution`	Execution parameters. See Table 4.6.	dict
`output`	Output options, such as name and format. See Table 4.7.	dict

Table 4.2: Description of all available options within each `finalize` dictionary in the configuration file for the Profiler.

Name	Description	Type
`clean_tmp_files`	Clean temporal files.	bool
`clean_asm_files`	Clean assembly files generated.	bool
`clean_asm_files`	Clean binary files.	bool
`command`	Execute a command after the execution of the set of experiments.	str

Table 4.3: Description of all available options within each `configuration` dictionary in the configuration file for the Profiler.

Name	Description	Type
`kernel_cfg`	Options passed to the Makefile.	str list
`d_features`	`-D` flags passed to the program. Each of them can be described as in Table 4.5.	dict
`flops`	Expression to compute the number of FLOPs. This can be expressed dynamically using `d_features` dictionaries.	str

Table 4.4: Description of all available options within each `compilation` dictionary in the configuration file for the Profiler.

Name	Description	Type
`enabled`	Enables/disables compilation. Useful for pre-generated binaries.	bool
`processes`	Number of processes to use for compilation.	int
`compiler_flags`	Dictionary of compilers with a list of specific flags for each of them.	dict of lists
`main_src`	Main source file to be compiled.	str
`kernel_inlined`	If true, kernel definition within the main source file. If false, kernel definition in `kernel_src` file.	bool
`loop_type`	"`asm`" or "`C`". Determines the language for MARTA instrumentation insertion.	str
`asm_analysis`	• `syntax`: ASM syntax. • `count_ins`: counts the number and type of ASM instructions in the ROI. • `static_analysis`: performs code analysis using LLVM-MCA.	dict

Table 4.5: Description of all available options within each `d_features` dictionary in the configuration file for the Profiler.

Name	Description	Type
`type`	Type of expression: `static`, `dynamic`, or `dependent`. `static` for `list` arguments, `dynamic` for iterators, e.g., itertools. If `dependent`, then the value will be computed according to the variable specified in the expression `value` (see below).	str
`val_type`	Value of the expression: `numeric`, `string`.	str
`value`	Expression generating the list of values, e.g., `[0,1,2,3]`, `itertools.product([0,1], [10,20])`, etc.	`Object`
`restrict`	If set, MARTA discards the inclusion of the feature if the condition is satisfied. This is useful to avoid the product of redundant combinations; e.g., `restrict: "N > 4"` means that the variable will only be considered if `N` is higher than 4.	`str`

Table 4.6: Description of all available options within each `execution` dictionary in the configuration file for the Profiler.

Name	Description	Type
`enabled`	Enables execution.	bool
`papi_counters`	List of PAPI counters to read.	str list
`time`	Measures execution time with `gettimeofday`.	bool
`tsc`	Measures TSC cycles using `rdtsc`.	bool
`nexec`	Repetitions per each configuration.	int
`threshold_outliers`	Threshold for outlier detection.	int
`mean_and_discard`	Computes average values after discarding outliers.	bool
`nsteps`	Number of iterations of the loop containing the ROI if specified.	int
`intel_turbo`	Enables or disables turbo boost on Intel processors via MSR.	bool
`max_freq`	Sets maximum CPU frequency via MSR.	bool
`cpu_affinity`	Logical CPU ID for pinning single-thread measurements.	int
`cache_flush`	Cache flush enabled for architectures supporting `CLFLUSH`.	bool

Table 4.7: Description of all available options within each `output` dictionary in the configuration file for the Profiler.

Name	Description	Type
`name`	Name of output file.	str
`columns`	Outputs columns. If "`all`", then all dimensions used in the configuration: `compiler`, `d_features`, `kernel_config`, `papi_counters`, etc.	str
`report`	Generates a log file with all information related to the experiment: host machine, elapsed time, standard output, standard error, etc.	bool

```
#include "marta_wrapper.h"
void foo(int N, int M, float *restrict y,
            float *restrict A, float *restrict x) {
    for (int i = 0; i < N; ++i)
            for (int j = 0; j < M; ++j)
                y[i] += A[i*N + j] * x[i];
}
MARTA_BENCHMARK_BEGIN;
POLYBENCH_1D_ARRAY_DECL(y, float, N);
POLYBENCH_1D_ARRAY_DECL(A, float, N*M);
POLYBENCH_1D_ARRAY_DECL(x, float, N);
PROFILE_FUNCTION(foo(N,M,y,A,x));
MARTA_AVOID_DCE(y);
MARTA_BENCHMARK_END;
```

Listing 4.1: Toy example of a benchmark using macros included in MARTA. We have used macros included in PolyBench/C for declaring variables, even though this is not a requirement. The code implements a matrix-vector multiplication kernel, which is executed 1,000 times in a loop for its instrumentation.

same kernel or program, as defined by the Cartesian product of, e.g., the compiler flags, the input to the kernel, or the program features employed. Lists can be specified in a *pythonic* manner, both dynamically (e.g., using iterators) or statically (i.e., defining a concrete set of elements). Parameters are passed to the target binary using `-D` preprocessor macros. Tables 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7 list the possible compilation, flags parametrization, execution and output options.

MARTA includes a set of tested directives or macros for writing new benchmarks and/or profiling a function of interest in an already existing program. These directives are meant to avoid low-level details for starting, stopping and printing values in hardware counters, avoid loop optimizations if the ROI is instrumented within a loop, declare new variables, avoid dead code elimination (DCE) optimizations, flush cache memories, or initialize data. These macros can be used as an independent library as well, but their main goal is to integrate the instrumentation of the code with the toolkit seamlessly. Table A.1 in Appendix A describes these predefined directives, whose usage will be exemplified in different case studies in Section 4.5. For completeness, Listing 4.1 illustrates how these macros can be used for writing a new benchmark in a few lines. This benchmark is fully compliant with the system,

Table 4.8: Description of all available options within each `kernel` dictionary in the configuration file for the Analyzer.

Name	Description	Type
`input`	Input data in CSV format.	str
`output_path`	Output path.	str
`prepare_data`	Preprocessing configuration. See Table 4.9.	dict
`plot`	Plotting parameters. See Table 4.10.	dict
`classification`	Parameters for classification analyses, e.g., decision trees.	dict
`feat_importance`	Parameters for feature importance analyses, e.g., random forests.	dict

requiring only the addition of a header and a set of directives.

4.4.2 Analyzer

Besides the CSV file containing the input data, the Analyzer also requires a configuration file to specify the data wrangling, classification, feature evaluation, and plotting tasks to perform. Table 4.8 details the list of parameters available to this module. Table 4.9, within the `prepare_data` dictionary, details the configuration parameters for the preprocessing stage. Currently, the system supports building de-

Table 4.9: Description of all available options within each `prepare_data` dictionary in the configuration file for the Analyzer.

Name	Description	Type
`cols`	Columns or dimensions to consider.	list
`rows`	Values of rows to filter.	dict
`target`	Dimension of interest, e.g., FLOPS.	str
`norm`	Normalization of values for the target dimension: `minmax` or `zscore`.	str
`categories`	Dictionary containing meta-information for the categories: • `num`: number of categories to generate statically. • `grid_search`: uses KDE and performs grid search for bandwidth and kernel parameters. • `mode`: if `normal`, Silverman is used for KDE. If `multimodal`, Improved Sheather-Jones is used.	dict

Table 4.10: Description of all available options within each `plot` dictionary in the configuration file for the Analyzer.

Name	Description	Type
`sort`	Dimension to use for sorting values.	str
`type`	Type of plot: relplot, scatterplot, lineplot or kdeplot.	str
`format`	Output format: png, pdf, eps, ps or svg.	str
`x_axis`	Dimension for the X axis.	str
`y_axis`	Dimension for the Y axis.	str
`hue`	Dimension to group by color.	str
`size`	Dimension to group by size.	str
`log_scale`	Applies logarithmic scale.	bool

cision tree classifiers, for discriminating between categories of the target dimension and the values of dimensions of interest; and random forest classifiers, for learning about the importance of each feature in the dataset. It is straightforward to incorporate other classifiers and analyses to the framework, such as SVMs, k-means clustering, etc.

The Analyzer can also generate different types of plots. The parameters for the plotting step are described in Table 4.10. The configuration file allows the user to set all available parameters in the scikit-learn library for both classifiers [114] and [115].

4.5 Evaluation: Case Studies

In order to illustrate the performance and capabilities of the tool, this section describes five different case studies. Each of them contains the motivation and description of the space to explore, the advantages of using MARTA, and the use of the tool in order to evaluate the case study. Note that all the plots in this section have been automatically generated by the framework, directly using the output of the Profiler, together with a configuration file as the input to the Analyzer.

4.5.1 Micro-benchmarking gather

Packing random operands into a single vector using one instruction was not possible until AVX2 with the addition of the gather instruction. This instruction is decoded into many micro-operations, depending on the architecture. In addition, the latency of a gather is not constant, depending on the type and form of the operands [1]. In this case study we want to quantify the impact of the number of elements and cache lines touched by this instruction when the cache is cold for an Intel Xeon Silver 4216 (Cascade Lake) and an AMD Ryzen 9 5950X (Zen3).

Definition of the exploration space

Gather is a complex x86 macro-instruction introduced in AVX2 for loading random data points given a starting address and a set of indices. This instruction has been reported to deliver variable latencies [1], depending on the source and destination operands. Hofmann et al. [43] demonstrate the performance variability of the gather instruction for the Intel Knights Corner and Intel Haswell architectures, depending on the number of elements fetched by the gather instruction from a cache line.

Differently, the present experiment explores the impact of the number of cache lines touched by a gather instruction but considering a cold cache, i.e., when data fills come from main memory. We actively avoid any cache prefetching impact, and analyze the real cost of gathering random data elements from main memory.

MARTA in action

The source code employed to explore this search space using MARTA is detailed in Listing 4.2. The `DO_NOT_TOUCH(var)` directives avoid any compiler optimization on variable `var`, e.g., dead code elimination. The assembly code generated for this input is shown in Listing 4.3. As it can be seen, the instrumentation overhead is minimal. Similarly, any assembly code can be plugged directly into the input source passed to MARTA for compilation and execution.

```
#include "marta_wrapper.h"
#include <immintrin.h>
void gather_kernel(float *restrict x) {
    __m256i index =  _mm256_set_epi32(IDX7, IDX6, IDX5, IDX4,
                                      IDX3, IDX2, IDX1, IDX0);
    __m256 tmp = _mm256_i32gather_ps(x, index, 4);
    DO_NOT_TOUCH(tmp);
    DO_NOT_TOUCH(index);
}
MARTA_BENCHMARK_BEGIN;
POLYBENCH_1D_ARRAY_DECL(x, float, N);
init_1darray(POLYBENCH_ARRAY(x));
MARTA_FLUSH_CACHE;
PROFILE_FUNCTION(gather_kernel(POLYBENCH_ARRAY(x) + OFFSET));
MARTA_AVOID_DCE(x);
MARTA_BENCHMARK_END;
```

Listing 4.2: Input C code for micro-benchmarking the gather FP instruction. The configuration file for this benchmark will declare all possible values for the IDX# variables.

As an example of our configuration, the possible list of values for the `IDX#` variables for gathering 8 elements are:

- `IDX0`: [0]
- `IDX1`: [1, 8, 16]
- `IDX2`: [2, 9, 32]
- `IDX3`: [3, 10, 48]
- `IDX4`: [4, 11, 64]
- `IDX5`: [5, 12, 80]
- `IDX6`: [6, 13, 96]
- `IDX7`: [7, 14, 112]

The Cartesian product of these lists generates a space of more than 2K elements, including all combinations of these gather instructions touching any number of cache

```
1     ...
2     vmovaps ymm1, YMMWORD PTR [rsp]
3     vmovdqa ymm2, YMMWORD PTR .LC1[rip]
4     call    polybench_start_timer@PLT
5     test    eax, eax
6  begin_loop:
7     vmovaps ymm3, ymm1
8     vgatherdps  ymm0, DWORD PTR [rax+ymm2*4], ymm3
9     add rax, 262144
10    cmp rbx, rax
11    jne begin_loop
12    call    polybench_stop_timer@PLT
13    ...
```

Listing 4.3: Example of 256-bit-register assembly code generated for the gather experiment: `rax` holds an offset to avoid data reuse, `ymm2` is used to compute the indices, and `ymm3` holds the mask (e.g., when gathering less than 8 elements using 256 bits).

lines from 1 to 8. The same reasoning is extended for the remaining gather experiments, varying from 2 to 8 data points or elements. In total, we generate more than 3K combinations for each platform. The total execution time for all these experiments is roughly three hours on each machine, including the compilation process.

Evaluation

Our analysis focuses on two factors: 1) the impact of the number of distinct cache lines on the gather instruction, and 2) the difference in performance between two of the latest Intel and AMD architectures. The target performance metric is the TSC cycles, in order to be frequency agnostic.

MARTA generates categories based on the density distribution of the values obtained. Figure 4.2 illustrates the resulting distribution plot, showing the different centroids and the categories induced by MARTA according to the KDE approach. The legend in this figure describes the categories generated for the TSC values. Even though some of them are not visible in the figure due to their order of magnitude, the system helps to locate them by displaying the peak centroids in the distribution.

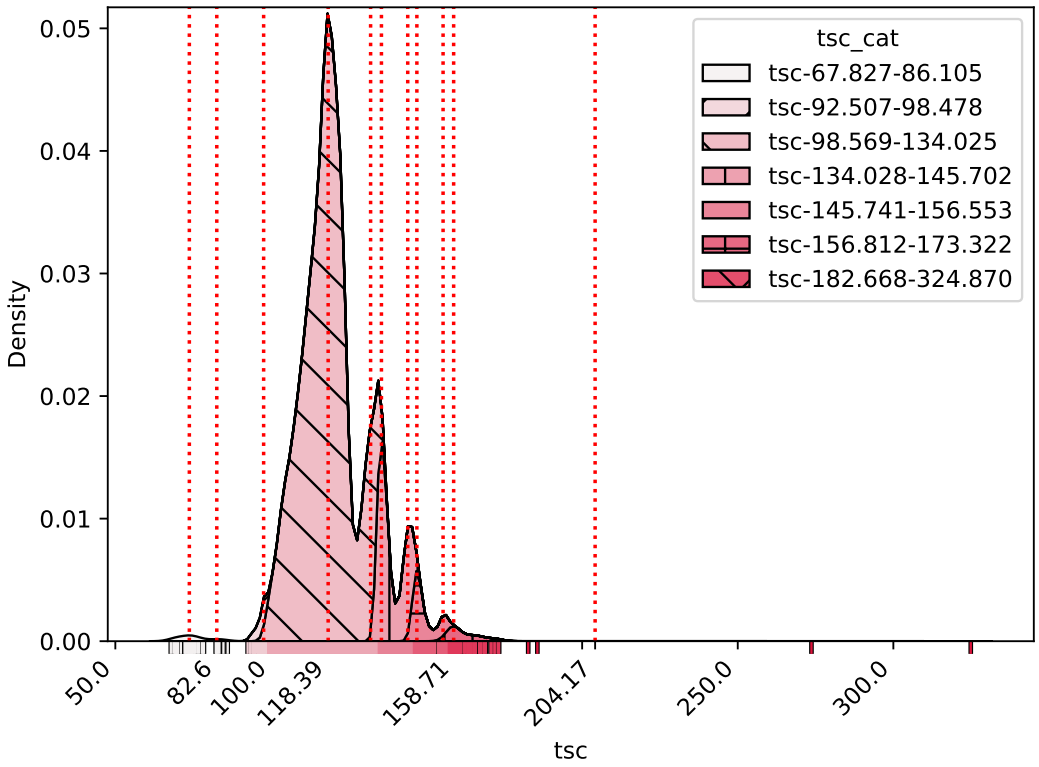


Figure 4.2: Distribution plot for gather with respect to its performance in terms of TSC cycles (log scale). Vertical dashed lines mark the peak centroids of each category found.

Based on this model, the system builds a decision tree as shown in Figure 4.3, with an accuracy rate of $\approx 91\%$. The model uses the number of cache lines touched by the instruction (`N_CL`), the vector width (`vec_width`, 0 for 128 bits and 1 for 256 bits), and the host platform (`arch`, 0 for AMD and 1 for Intel). The structure very clearly gives the intuition that the degradation in performance is related to the increase in the total number of cache lines touched by the gather instruction. However, this simple model can also discover some other hidden architectural effects, e.g., following the decision tree, our model is able to detect that the AMD Zen3 performs better when the number of cache lines touched is 4 when using 128-bit width vectors. This behavior is not present in the Intel machine.

This decision tree also serves to investigate why the predictor missclassifies certain points. In this specific case, after manual exploration, we found out that most

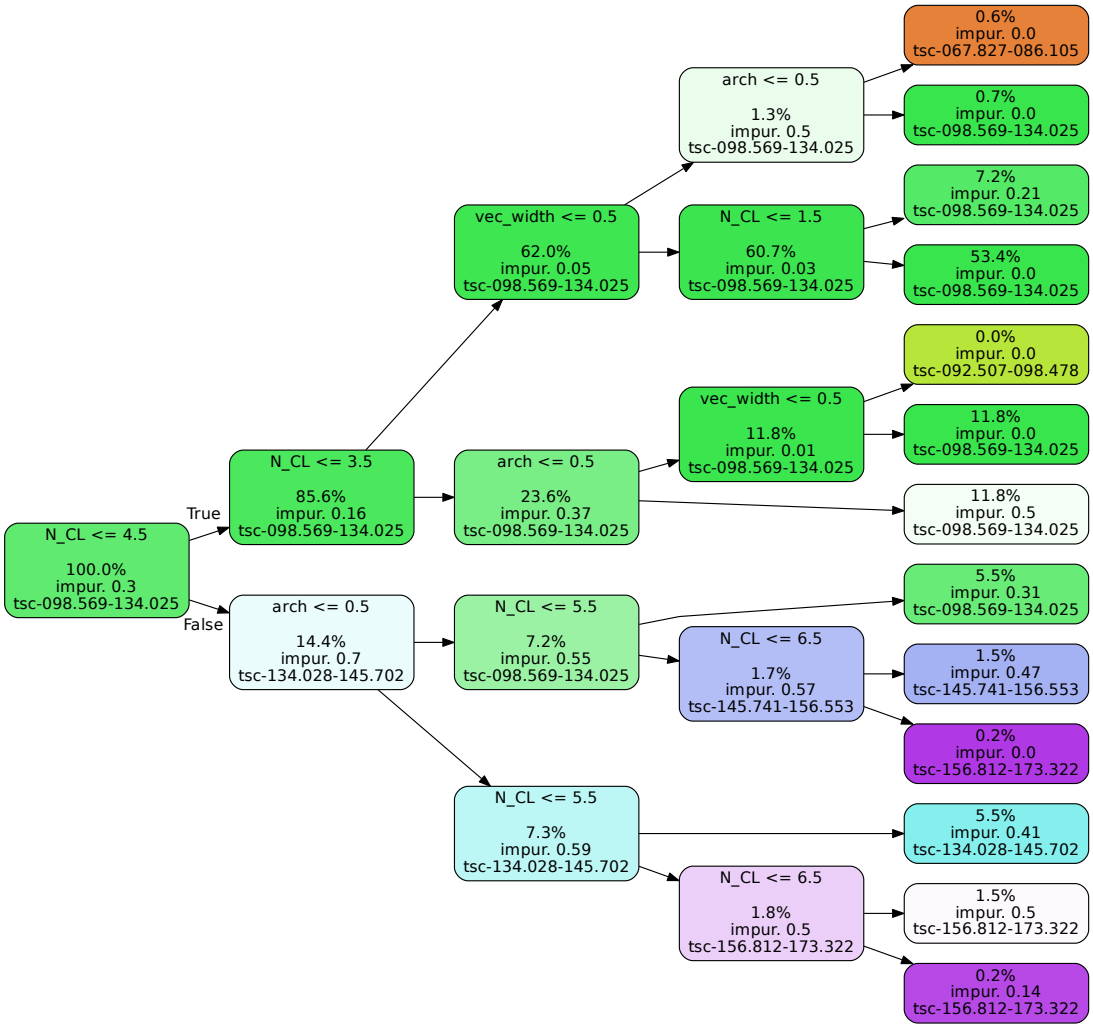


Figure 4.3: Decision tree for predicting the performance of gather based on the categories generated by the system. `N_CL` is the number of cache lines touched by the program, `arch` is 0 for AMD Zen3 and 1 for Intel Cascade Lake. `vec_width` is 0 for 128-bit vectors, and 1 for 256-bit vectors. Nodes in lighter colors represent a higher impurity degree, which is not desirable. Each node is labeled with its assigned category.

errors are attributable to fuzzy categorical boundaries and natural measurement noise.

In this case we focus on decision trees, as they allow to visualize a partitioning of the space in a manner that is intuitively interpretable by the user. Other techniques such as linear regression might provide lower RMSE, but they are also typically much less intuitive and make knowledge transfer harder than, e.g., a small decision tree as built by MARTA. On the other hand, the feature importance analysis (MDI) reports a high difference between the number of cache lines touched, the architecture and vector width: 0.78 against 0.18 and 0.04, respectively.

To conclude this case study and our post hoc analysis, we have demonstrated that, under cold cache conditions, the performance of gather operations is clearly dependent on the number of cache lines used. The degradation is remarkable when increasing the number of cache lines touched by the instruction. On our Intel Cascade Lake processor there is no influence on performance of the vector width or the use of masks. There is, however, a noticeable and interesting difference when using the 128-bit width version on our AMD Zen3 processor.

4.5.2 Empirical throughput of FMA instructions

Modern architectures include Fused Multiply-Add (FMA) units in their designs. In this case study we want to empirically discover the throughput of the FMA instruction for an Intel Cascade Lake and an AMD Zen3 processor.

Definition of the exploration space

FMA instructions perform fused multiply-add operations and were introduced as extensions of the 128- and 256-bit SIMD instructions in x86. There were some divergences between Intel's and AMD's implementation at the beginning, but modern architectures, starting from Haswell on Intel and Zen2 on AMD, implement the FMA3 ISA. All instructions available in this ISA have the form of $d = a \times b + c$, where d must be the same register as either a, b or c. As such, there are different variants of these same operations, for instance, `vfmadd132ps` and `vfmadd213ps`, which vary the

operands chosen for the multiplication and the destination operand. These instructions have dedicated resources in the pipeline, typically FMA units. However, these units share ports in the pipeline with other architectural units such as the division, integer (e.g., ALU, jump, load effective address, etc.), or shift units. In this case we want to get the actual throughput of any FMA instruction for a given platform, regardless of data type, vector width, or interferences with any other instruction.

MARTA in action

This experiment requires micro-benchmarking, and MARTA includes a specific configuration for this purpose. It also requires hot cache conditions in order to get the maximum throughput of consecutive and independent FMA instructions. We consider two or more FMA instructions to be independent iff there is no data dependency among them. MARTA is able to automatically generate the C code required for benchmarking a list of assembly instructions. It can also generate all the possible permutations of the subsets of this instruction list. This is useful if our experiment requires to consider all possible orderings of the instructions to measure.

For this purpose, we specify the list of assembly instructions to benchmark in a configuration file, as described in Listing 4.4, or using the CLI (see full list of options in Appendix A.1), e.g., `marta_profiler perf --asm "vfmadd213ps %xmm2, %xmm1, %xmm0"`. MARTA is also in charge of unrolling these instructions, for reproducibility reasons, and executing warm up iterations. All these parameters are also configurable during runtime.

Extending the example in Listing 4.4, MARTA makes it straightforward to write more benchmarks but changing the registers (i.e., vector width) and the data type (`ps` suffix in the code). MARTA automates the generation of these combinations according to the number of consecutive independent instructions we want to issue, from only the first instruction up to all of them.

```
asm_body:
    [
        "vfmadd213ps  %xmm11, %xmm10, %xmm0",
        "vfmadd213ps  %xmm11, %xmm10, %xmm1",
        "vfmadd213ps  %xmm11, %xmm10, %xmm2",
        "vfmadd213ps  %xmm11, %xmm10, %xmm3",
        "vfmadd213ps  %xmm11, %xmm10, %xmm4",
        "vfmadd213ps  %xmm11, %xmm10, %xmm5",
        "vfmadd213ps  %xmm11, %xmm10, %xmm6",
        "vfmadd213ps  %xmm11, %xmm10, %xmm7",
        "vfmadd213ps  %xmm11, %xmm10, %xmm8",
        "vfmadd213ps  %xmm11, %xmm10, %xmm9",
    ]
```

Listing 4.4: Example of list of instructions to benchmark for getting the FMA throughput in AT&T format, using 128-bit vectors for floating-point precision.

Evaluation

In this experiment we ran a set of benchmarks varying the following features: 1) number of independent FMA instructions executed contiguously (from 1 to 10), 2) vector width (128 bits, 256 bits and 512 bits, if available), and 3) data type (single and double precision). A total of 60 benchmarks are generated. We ran these experiments on three different machines: Intel Xeon Silver 4216 (Cascade Lake), Intel Xeon Gold 5220R (Cascade Lake), and AMD Ryzen 9 5950X (Zen3). The results are shown in Figure 4.4 in the form of a line plot, colored by the configuration (data type and vector width), and with the line style according to the architecture used. The figure clearly shows the saturation points for these architectures. Conducting such experiment can validate, or even replace, manufacturer's documentation on the throughput of specific instructions.

We observe under which scenario both AMD and Intel machines allow 2 FMAs to be executed in a single cycle, independently of their vector width. It requires to have at least 8 independent FMAs in the loop body to achieve a throughput of 2 FMAs per cycle, as otherwise the throughput is reduced. This experiment highlights how one would fail to achieve a throughput of 2 FMAs per cycle with only two independent FMAs in flight. We suspect this is related to the 4-cycle latency of FMA instructions. For Intel machines using AVX-512, only one FMA can

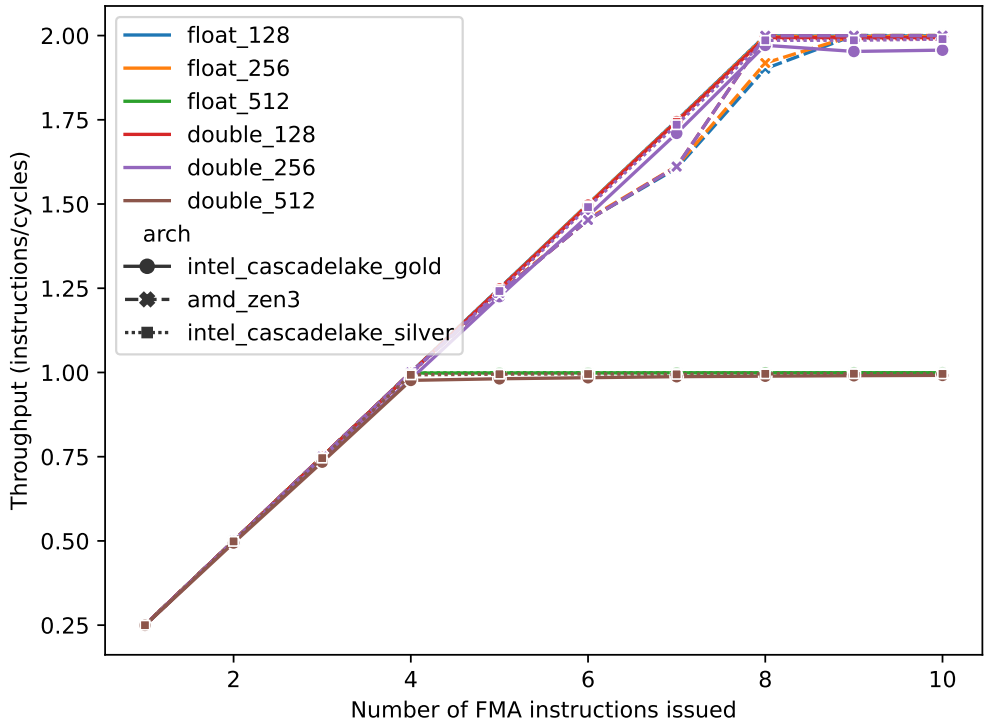


Figure 4.4: Line plot generated by MARTA according to the number of independent FMA instructions issued and the reciprocal throughput obtained, computed as the number of instructions executed divided by the number of cycles. All lines overlap, but the configurations using 512-bit vectors (`float_512` and `double_512`).

be issued per cycle. This indicates most likely the availability of a single AVX-512 FPU.

MARTA can generate a decision tree-based predictor for all architectures, as shown in Figure 4.5. This predictor, while naïve, is able to extract the importance of the features, accurately categorizing all data points.

To conclude our case study, we observed that both AMD Zen3 and Intel Cascade Lake have a maximum throughput of 2 FMAs per cycle using vectors of 128 and 256 bits, i.e., they can issue 2 FMAs in a single cycle, provided there are enough independent instructions in flight. AMD Zen3 does not feature AVX-512, while our Intel Cascade Lake processors feature a single AVX-512 FPU. From an architectural point of view, this is typically done by fusing both 256-bit units when issuing instructions using 512-bit vectors.

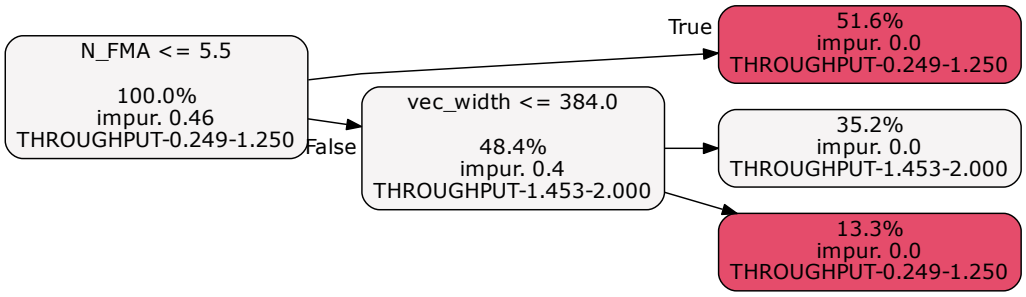


Figure 4.5: Simple predictor synthesized by MARTA according to the number of FMAs issued and the vector width of the instructions used.

4.5.3 Influence of access pattern on memory bandwidth

The performance of memory-bound benchmarks is limited by the main memory bandwidth. This bandwidth is affected by several factors, not all of them intuitive. Modern architectures are designed to efficiently access data in a streamlined fashion by implementing mechanisms such as software and hardware prefetching. In this section, we focus on the analysis of a triad operation on a double-precision floating-point data type, studying the effect of changing the element access order.

Definition of the exploration space

Memory bandwidth is typically computed for contiguous memory addresses, as in STREAM benchmark [84]. The classic triad kernel has the form of $c(i) = a(i) * b(i)$. In this case study, we are going to characterize the influence of the access functions f, g, and h on the kernel $c(f(i)) = a(g(i)) * b(h(i))$.

Intuitively, a purely streamlined access, i.e., $f(i) = g(i) = h(i) = i$, will deliver the best possible bandwidth. But how is this peak bandwidth affected by strided accesses, or even random accesses, in one or more streams, such as could occur for example in Sparse Matrix-Vector Multiplication computations?

```
1 __m256d regA1 = _mm256_load_pd(&a[data_a]);
2 __m256d regA2 = _mm256_load_pd(&a[data_a + 4]);
3 __m256d regB1 = _mm256_load_pd(&b[data_b]);
4 __m256d regB2 = _mm256_load_pd(&b[data_b + 4]);
5 __m256d regC1 = _mm256_mul_pd(regA1, regB1);
6 __m256d regC2 = _mm256_mul_pd(regA2, regB2);
7 _mm256_store_pd(&c[data_c], regC1);
8 _mm256_store_pd(&c[data_c+4], regC2);
```

Listing 4.5: AVX triad kernel used for measuring memory bandwidth.

MARTA in action

We manually write a tuned version of the `STREAM` benchmark implementing the proposed triad operation directly using AVX Intrinsics, to avoid compiler optimization interference in the measurements. The code is shown in Listing 4.5. The accessed elements of each array are determined by the access variables `data_{a,b,c}`. The values of these variables are the ones that drive the study, and determine whether the access to each stream is sequential, strided or random. We set up the experiment so that all three streams and access variables are 64B-aligned all the time (i.e., memory block aligned). This means that strided and random accesses are not defined in terms of individual array elements, but of memory blocks. Once a block is selected, its eight contiguous double-precision elements are accessed (for `a` and `b`) or written (for `c`). We do this so that the total number of data accesses is invariable across different access patterns. Similarly, the total number of cache hits and misses will be invariable in the absence of prefetching, with the rare exception of the same block being selected twice in close succession for the random access experiments. The total number of iterations is equal to the total number of memory blocks in each array, `STREAM_BLOCKS`. The processor used for evaluation is an Intel Xeon Silver 4216 (Cascade Lake), and, for that reason, the size of each array is defined to be 16 Mi elements, i.e., 128 MiB or at least four times the total LLC size of 22 MiB, as recommended by the STREAM author. When accessing streams with a stride S, the benchmark accesses each block of each array exactly once as follows. During a first traversal, only blocks in positions $B \mid B \mod S = 0$ are accessed. In a second traversal, blocks in positions $B \mid B \mod S = 1$ are selected. The process

continues until, in traversal $S-1$, the last untouched blocks are accessed. This avoids unwanted cache reuse with large access strides.

We write the following benchmark versions: one with all sequential accesses that serves as a baseline; four strided versions, one with a stride on `b` only, one with a stride on `c` only, one with a stride on `b` and `a`, and one with a stride on all three streams; and four random versions in which `rand()` is used for each randomly accessed stream, in the same fashion as the strided access.

Evaluation

We use MARTA to automatically run 630 different micro-benchmarks. Each of the 9 different code versions described above is run using 1, 2, 4, 8 and 16 cores. Each strided version is run with S values from 1 to 8 Ki. We build a decision tree that tries to predict the achievable bandwidth by each access pattern given its stride and number of execution threads. This is useful to bound the performance of different classes of kernels, e.g., a Sparse Matrix-Vector Multiplication, which is similar to a triad in which one of the streams is accessed randomly. The decision tree shows remarkable impurity when classifying strided accesses. In order to study this, we first focus on analyzing the results obtained for a single thread. These are shown in Figure 4.6. The bandwidth achieved by fully sequential accesses is approximately 10 times smaller than the peak, at just 13.9 GB/s. Sequential and random accesses are not affected by the `STRIDE` parameter, and appear in this figure as bounds to the actual performance obtainable by the strided versions. The figure clearly shows how the bandwidth drops with the stride: it drops sharply for $S = \{2, 4, 8, 16, 32, 64\}$, to an average of 9.2 GB/s for the case of strided `b` only. The clear reason in this case is the ineffectiveness of the next line hardware prefetcher. There is another sharp drop starting at $S = 128$, to an average of 4.1 GB/s, which is similar to the performance of accesses using `rand()`. This is ultimately an artifact caused by the single-threaded execution, as will be shown with multithreaded experiments.

We analyze the bandwidth evolution as the number of threads increases up to the 16 physical cores available in the processor. We presume that the very low bandwidth achieved for a single thread is caused by front-end stalls and, consequently, trivially parallelizing the triad computation using OpenMP should greatly increase

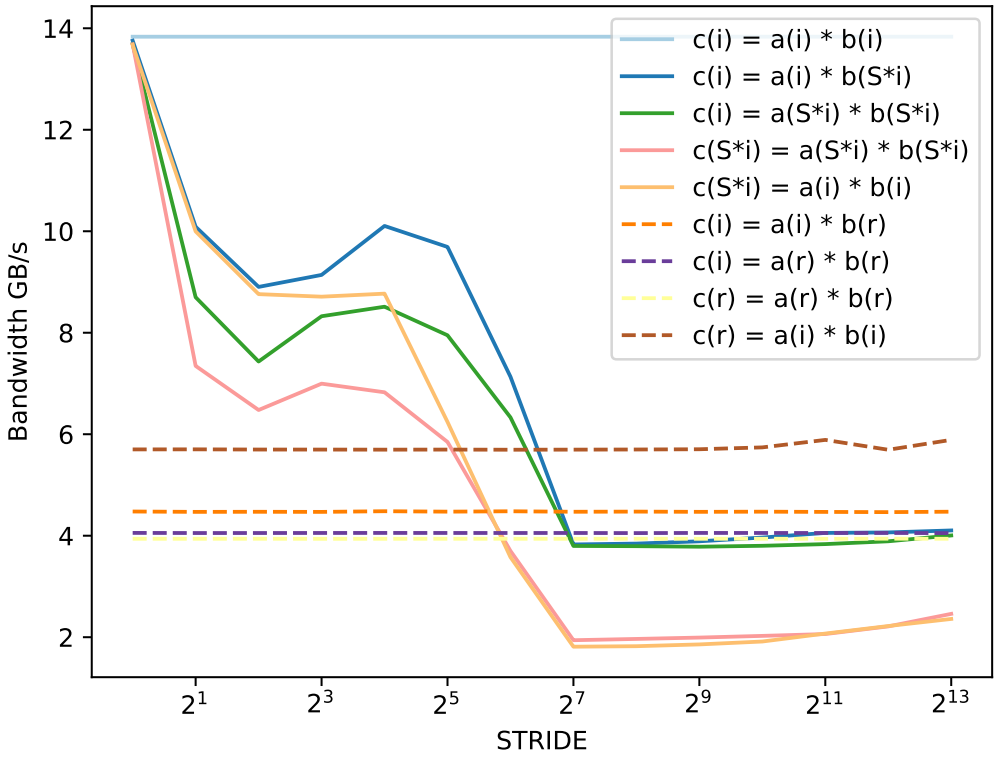


Figure 4.6: Bandwidth obtained for different access patterns using a single thread. Accesses labeled as `x[i]` represent sequential accesses, `x[S*i]` represent strided accesses, and `x[r]` represent random accesses.

the available bandwidth. The results are shown in Figure 4.7. We can see a clear increasing trend for all benchmark versions, except for those calling `rand()`. In this case, using multiple threads to access memory is harmful for performance, achieving a low peak bandwidth of only 0.4 GB/s for the version which accesses three random streams through calls to `rand()`. This low performance is caused by the enormous overhead introduced by the call to `rand()`, as these versions emit, on average, 5x and 6x more memory loads and stores, respectively.

4.5.4 Auto-vectorizing reductions

Compilers are able to vectorize certain types of loops and code blocks [38]. Modern auto-vectorizers are able to detect and vectorize reductions, which are common patterns in applications and kernels such as Sparse Matrix-Vector Multiplication. In

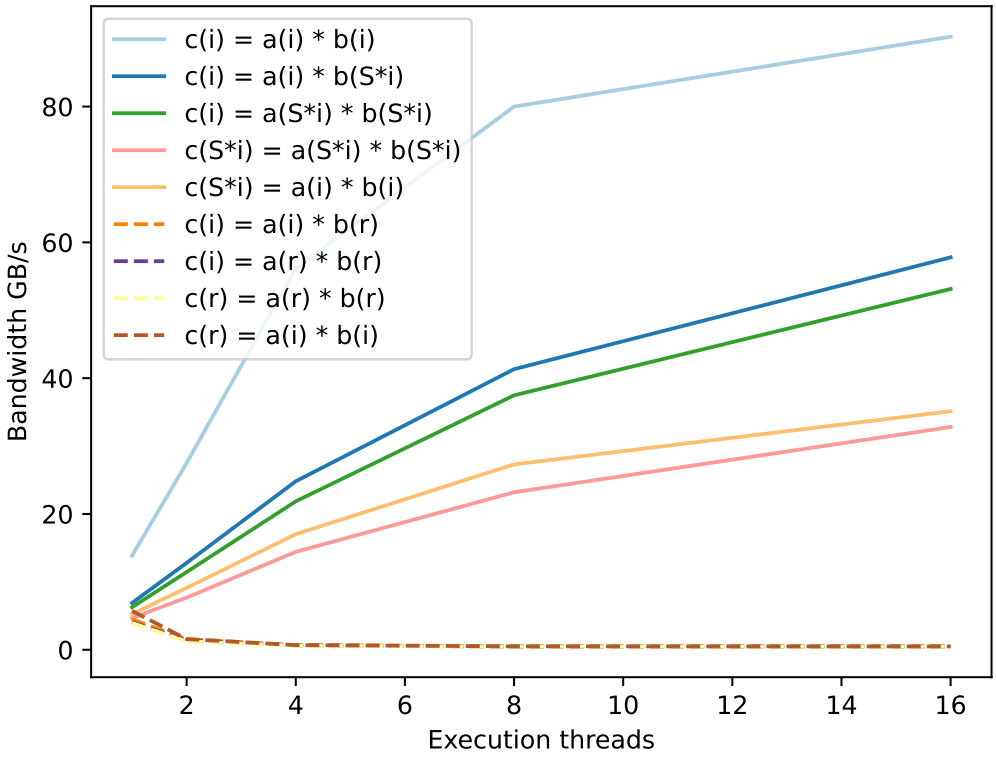


Figure 4.7: Multithreaded bandwidth per stream version. Accesses labeled as `x[i]` represent sequential accesses, `x[S*i]` represent strided accesses, and `x[r]` represent random accesses.

this case study, we assess whether the GCC and ICC compilers are able to vectorize reductions of floats using `-ffast-math` optimization.

Definition of the exploration space

According to the user manual, GCC is able to vectorize floating-point reductions using either the `-ffast-math` or the `-fassociative-math` flags [38]. This statement is true for simple reduction loops, such as the one in Listing 4.6, which features the simplest case in which the loop accesses contiguous array elements, with an initial value and number of iterations known at compile time. Even when the number of iterations is unknown at compile time, the compiler should be able to vectorize the code using versioning and loop peeling.

```
1 for (int i = 0; i < N; ++i) {
2     sum += x[i] * 42.0f;
3 }
```

Listing 4.6: Vectorizable reduction of `N` floating-point values.

However, and depending on the actual values and coding style for the loop step and bounds, the compiler may fail to vectorize the reduction. The aim of this experiment is to test under which conditions the compiler decides to not vectorize the code, whether due to the cost model indicating non profitability or to a coding style problem.

MARTA in action

For this experiment, the reduction is rewritten as shown in Listing 4.7. The list of values for the parameters in the code, whose Cartesian product defines the exploration space, is as follows:

- `N`: [1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 28, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128]
- `INIT_VAL`: [1, 2, 3, 7]
- `LOOP_STEP`: [1, 2, 3, 5]
- `ACC_STRIDE`: [1, 2, 5, 10]

In addition to the parameter values themselves, we also explore how the auto-vectorizer is affected by their being parametric or not, i.e., hardcoding the values as constants in the source code as opposed to using variables. This creates four additional binary conditions, i.e., 2^4 combinations. For completeness, this experiment explores the behavior of ICC in addition to GCC. The full exploration space includes a grand total of more than 53K cases: $2 \times 26 \times 4^3 \times 2^4$.

As described in Section 4.2, MARTA can perform static code analysis, but is also capable of interpreting the optimization reports generated by the compiler in

```
1 void KERNEL(int n, int init_val, int loop_step, int index_factor,
2             DATA_TYPE POLYBENCH_1D(x, N, n)) {
3   DATA_TYPE sum = 0;
4   for (int i = INIT_VAL; i < N; i += LOOP_STEP) {
5       sum += x[ACC_STRIDE * i] * 42.0f;
6   }
7   DO_NOT_TOUCH(sum);
8 }
```

Listing 4.7: Input code to MARTA for benchmarking a single vector-constant multiplication reduction. Each of the features included (`N`, `INIT_VAL`, `LOOP_STEP` and `ACC_STRIDE`) will be compiled as a variable if configured to be parametric, and its value will not be known at compile time.

order to determine whether a particular loop has been vectorized. In addition to the input sources, the configuration for this experiment only requires the YAML file specifying all the features described above: whether the upper bound of the loop is scalar or parametric, the list of compilers to be used and their flags, and the different possible values for each variable.

Evaluation

As we stated in the description of the exploration space, we have run more than 53K experiments, out of which $\approx$ 49% are vectorized, and $\approx$ 51% are not. Figure 4.8 depicts the density graph for vectorized and non-vectorized cases, showing the number of iterations of the loop in the X axis. As can be seen, the majority of non-vectorized cases have low trip counts. The results of the experiments were passed to the Analyzer to build the decision tree in Figure 4.9, which predicts whether a loop will be vectorized or not. The resulting tree achieves an accuracy of $\approx$ 94% with only four depth levels. The tree essentially looks at trip count values, and so loops with 11 iterations or less are usually not vectorized (with the exception of loops where the access stride is either 1 or 2, and the loop step is known at compile time). According to the model, all cases where the access stride is 1 or 2 are vectorized, while those where the loop step is parametric and the number of iterations is lower than 60 are not.

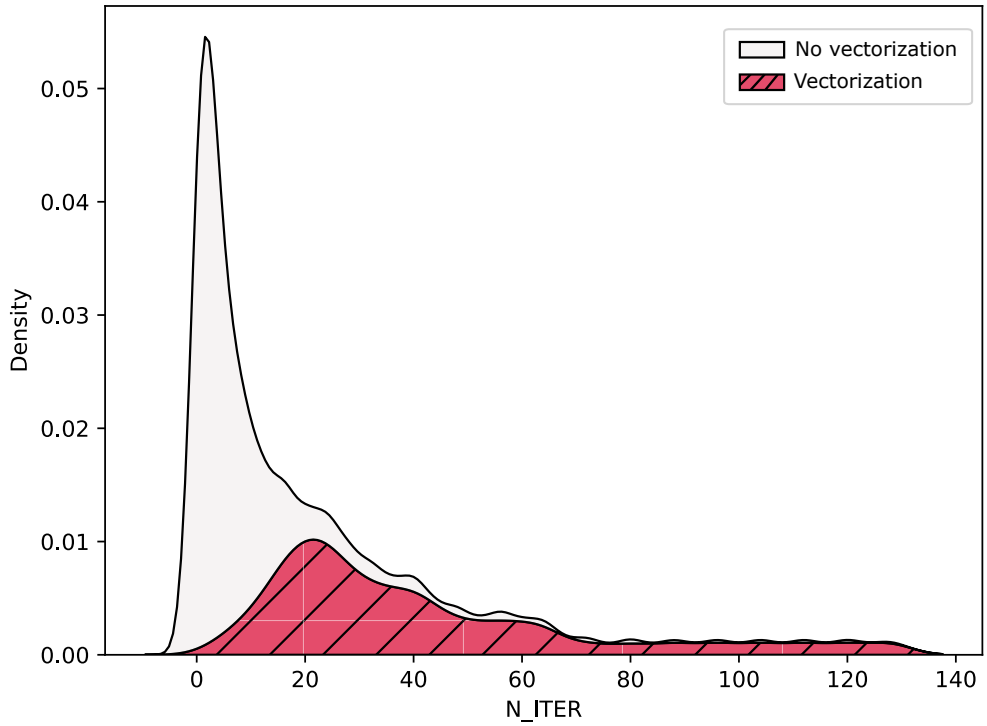


Figure 4.8: Stacked density graph of cases where loops are vectorized (diagonal hatch pattern) and non-vectorized (plain pattern) with respect to the number of iterations `N_ITER` of each loop. `N_ITER` is a derived parameter that MARTA computes as `N` divided by `LOOP_STEP`.

The feature importance analysis reports a 71% weight for the number of iterations, and 12% and 10% for the access stride and parametric condition of the loop step, respectively, which matches the model described above. To close this case study, we conclude that the cost model for the vectorization is driven mostly by the trip count of the loop, for both GCC and ICC.

4.5.5 Cost model for loop permutation

Interchanging the nesting order of a loop nest may improve locality. This technique is widely known and employed in all different compilers. Notwithstanding the potential benefits of this loop transformation, the cost model of the compiler may include other parameters preventing this optimization. In this case study, we

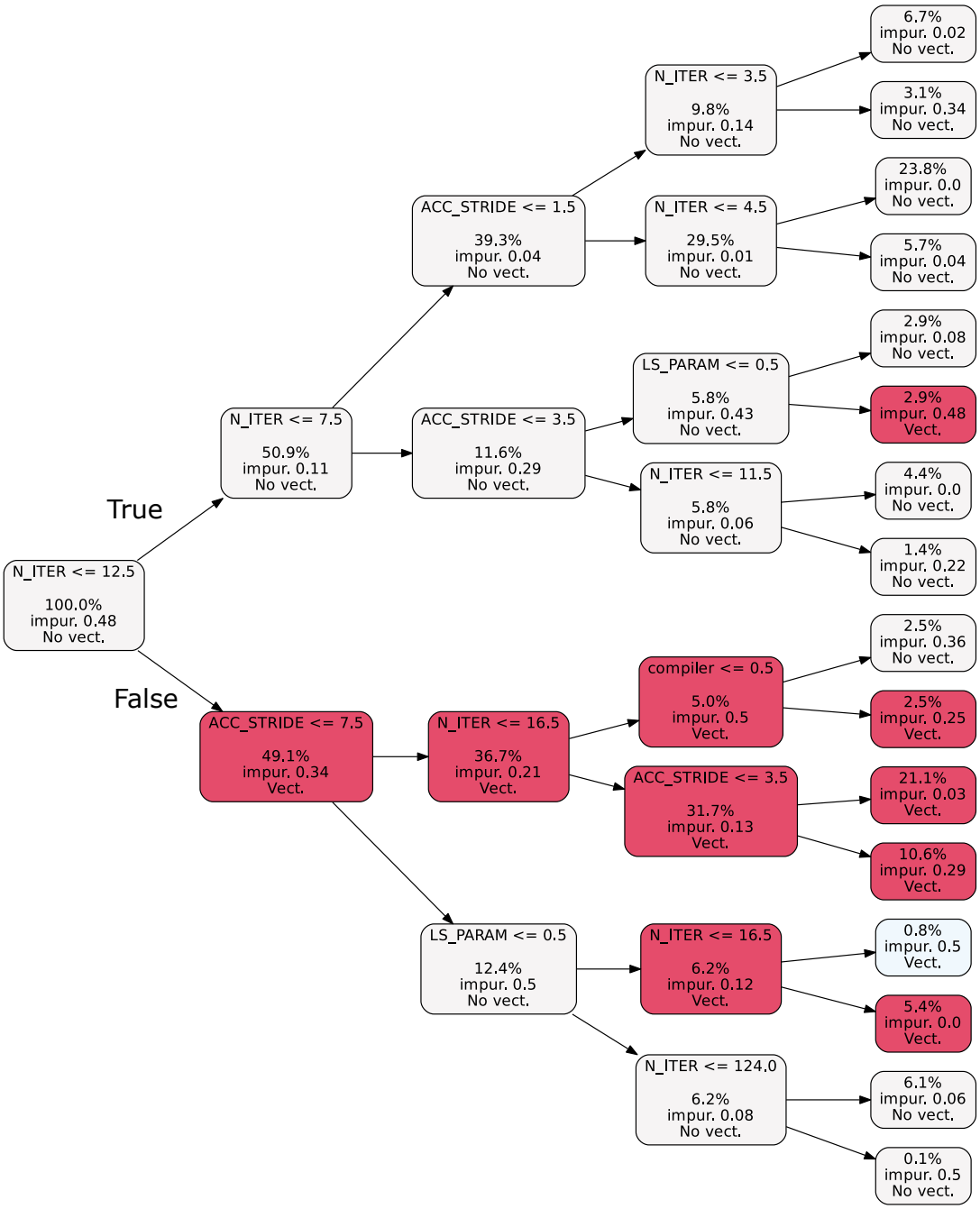


Figure 4.9: Decision tree to predict whether a loop will be vectorized. `N_ITER` is the number of iterations, `LS_PARAM` indicates whether the loop step is parametric or not, and `ACC_STRIDE` is the access stride constant that multiplies the loop induction variable in the array index. Each node is labeled with its assigned category.

investigate the factors that lead to loop permutation being applied by GCC and ICC.

Definition of the exploration space

Loop permutation is a classic loop optimization that reorders the iterations of one or more loop nests in order to improve spatial locality. There are some obvious cases where this technique will be profitable, e.g., favoring row-major accesses when using 2-dimensional matrices. For instance, GCC will permute the loop shown in Listing 4.8. Besides improving locality, this optimization may also enable vectorization. Both GCC and ICC describe in their optimization reports (`-qopt-info-loop` and `-qopt-report-phase=loop`, respectively) if this technique has been applied for a particular loop nest. These data can be leveraged to build a cost model that tries to predict when a compiler will permute a loop nest of interest. According to the manual [39], GCC applies loop permutation to the loops depicted in Listing 4.8, by interchanging the innermost loop with the `j`-loop.

Our experimental setup explores 2- and 3-dimensional loop nests that compute either a multiply-and-add operation or a reduction, varying features such as the loop nest depth, trip count, variable dimensions, and whether bounds are parametric or constant.

MARTA in action

We build the exploration space based on the dimensionality and type of the loop nest (`CFG` variable), the size of the loop nest (`N`, `M` and `L`), whether loop bounds are parametric or not (`LOOP_BOUND_PARAMETRIC`), the dimension of the declared arrays

```
1 for (int i = 0; i < N; i++)
2     for (int j = 0; j < N; j++)
3         for (int k = 0; k < N; k++)
4             c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

Listing 4.8: 3-dimensional loop nest benefiting from loop permutation [39].

(`ARRAY_DIM`), and which compiler is used and its flags. The selected values for all these variables are enumerated below:

- `CFG`: [`LOOP_2D`, `LOOP_3D`, `REDUCTION`]
- `N`: [8, 16, 32]
- `M`: [8, 16, 32]
- `L`: [8, 16, 32]
- `LOOP_BOUND_PARAMETRIC`: [0, 1]
- `ARRAY_DIM`: [1, 2]
- `COMPILER`: [gcc, icc]
- `COMPILER_FLAGS`: [-O3, -Ofast]

The Cartesian product of these sets generates an exploration space of $3^4 \times 2^4 = 1,296$ elements. Configuring MARTA for this task only requires writing the code to profile, shown in Listing 4.9, and the configuration file with the parameters described above. As in the previous experiment, MARTA will automatically parse configuration reports and detect whether loop permutation is applied to a particular loop or not. This information will be included in the CSV output from the Profiler, which is then passed to the Analyzer.

Evaluation

From the performance point of view, all code versions benefit from the manual application of this technique. For the `LOOP_2D` configuration none of the input cases is optimized by GCC or ICC, as detailed in Figure 4.10. We build two different decision trees, illustrated in Figures 4.11 and 4.12. The first one models the influence of dimensions for the 3-dimensional loop nest, and the second one for the 2-dimensional reduction. Each of them have an accuracy of $\approx 99\%$. In both cases the compiler of choice is the most impacting factor (weight of over 70% in the feature analysis report).

```
1  #if defined(LOOP_2D)
2      for (int i = 0; i < _UB_N; i++)
3          for (int j = 0; j < _UB_M+; j++)
4  #  if ARRAY_DIM == 2
5              A[i][j] += B[j][i] * C[j][i];
6  #  else
7              A[i*_UB_N + j] += B[j*_UB_M + i] * C[j*_UB_M + i];
8  #  endif
9  #elif defined(LOOP_3D)
10     for (int i = 0; i < _UB_N; i++)
11         for (int j = 0; j < _UB_M; j++)
12             for (int k = 0; k < _UB_L; k++)
13 #  if ARRAY_DIM == 2
14             A[i][j] += B[i][k] * C[k][j];
15 #  else
16             A[i*_UB_N + j] += B[i*_UB_N + k] * C[k*_UB_L + j];
17 #  endif
18 #elif defined(REDUCTION)
19     DATA_TYPE sum = 0.0f;
20     for (int i = 0; i < _UB_N; i++)
21         for (int j = 0; j < _UB_N; j++)
22 #  if ARRAY_DIM == 1
23             sum += A[j*_UB_N + i];
24 #  else
25             sum += A[j][i];
26 #  endif
27 #endif
```

Listing 4.9: Simplified input source code to explore loop permutation with MARTA.

To conclude this case study, we can state that the cost model implemented by GCC only applies permutation when using `-ffast-math`, 1-dimensional arrays for 3-dimensional loop nests, and the loop upper bound is 32 or greater. As for ICC, it mostly applies permutation in all cases, except for small trip counts and 1-dimensional arrays.

4.6 Related Work

Many different tools and approaches for profiling applications and architectures have been developed, from open source to commercial solutions, most of them based

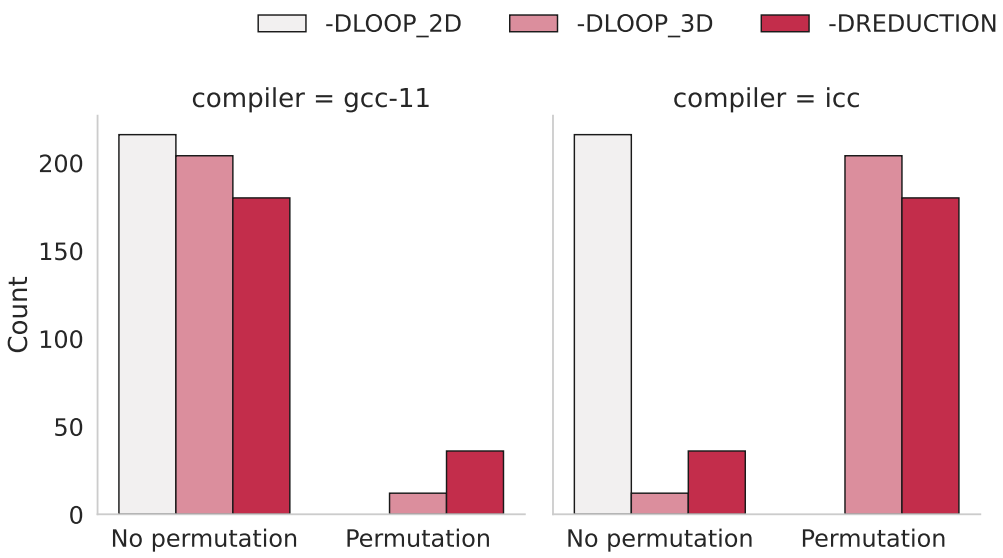


Figure 4.10: Distribution plot for the number of cases where the compiler decides to apply loop permutation, depending on the loop type (`CFG`) and the compiler. No permutation is performed for `LOOP_2D` by any of the compilers.

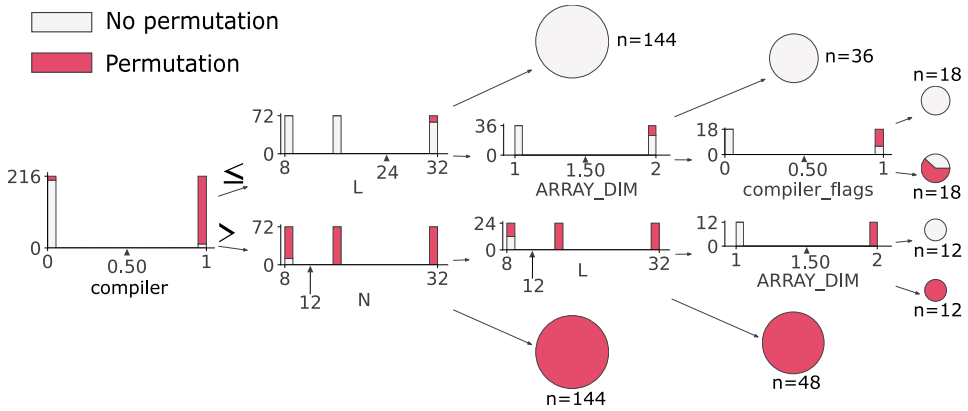


Figure 4.11: Decision tree built by MARTA for the `-DLOOP_3D` kernel. GCC is labeled as compiler 0, while ICC is labeled as compiler 1. `compiler_flags` is 0 for `-O3` and 1 for `-Ofast`.

on instrumentation using hardware counters. Accessing to the values contained in those counters typically requires reading Model Specific Registers (MSR) or Performance Monitor Counters (PMC). Profilers commonly use interfaces for accessing these hardware counters, such as `perf_events`. The API of these interfaces is usually complex and low-level. Therefore, there are many different implementations

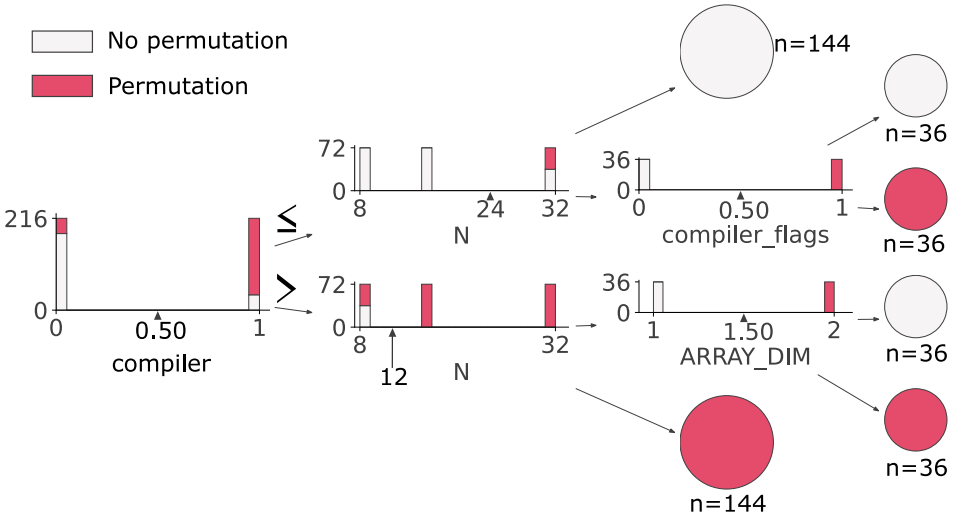


Figure 4.12: Decision tree built by MARTA for the `-DREDUCTION` kernel. GCC is labeled as compiler 0, while ICC is labeled as compiler 1. `compiler_flags` is 0 for `-O3`,and 1 for `-Ofast`.

depending on the operating system and platform. For Linux, `libpfm4` was developed as an almost zero-overhead solution for accessing these counters, and it is used in high-level interfaces such as PAPI. This library provides a flexible API for setting and programming events on these hardware counters. This alternative is intrusive, as the source code must be modified, but it provides accurate measurements (exact values are possible, sampling the hardware counters is optional) with very low overhead, as it also employs `rdpmc`, if possible, for reading performance counters.

There are different alternatives for monitoring and collecting execution data from binaries such as `perf`, which is included in the Linux kernel, OProfile [74] or LIKWID [127], which also integrates a library for accessing hardware counters similar to PAPI. These are low-level tools based on dynamic instrumentation and, therefore, do not require source codes to be recompiled. TAU [117], Extrae [14], Vampir [67], Scalasca Trace Tools [36], HPCToolkit [4], and Intel oneAPI [56] (formerly Intel Parallel Studio XE) provide sophisticated profiling environments, producing detailed and complete trace execution analyses from binaries. This last one employs a top-down methodology [136], which is meant for spotting the bottlenecks in the pipeline from a hierarchical point of view. Abel and Reineke [2] propose a micro-benchmarking methodology based on measuring instruction latencies and re-

ciprocal throughput for x86 architectures. The scope of this approach is meant for measuring instructions only, and not regions of code. Downs [25] presents another micro-benchmarking methodology for the analysis of micro-architectural features based on a set of synthesized benchmarks and a framework for building benchmarks. This toolkit uses `libpfc` [10] (another library for reading PMU using `rdpmc` instructions), and is limited to x86 architectures. timemory [80] is a modular C++ toolkit for performance analysis and logging. It provides a simple interface for instrumenting programs avoiding low level details of the instrumentation back-end and supports different programming languages. kerncraft [42] is a loop kernel analysis and performance modeling tool, which provides a framework for data reuse and cache analysis.

Our approach goes beyond the ones presented above by combining several of their abovementioned features, improving productivity and reproducibility with a simple interface for combining different parameters, which includes execution environment configuration. MARTA is lightweight and highly tunable, allowing to easily create large sets of data to analyze. This enables the analysis of very large exploration spaces for benchmarking via automated data mining techniques. We believe MARTA answers a need for users who drive their work based on experimental data for relevant micro-benchmarks: MARTA automates the experimental setup to ease reproducibility and portability on different machines/setups, and importantly, facilitates the task of analyzing the data produced by building predictors for the data from the input feature values.

4.7 Discussion and Concluding Remarks

We presented MARTA, a fully implemented, open source, and highly configurable toolkit for performance analysis of programs. Productivity and reproducibility are improved with automated benchmark template generation from a simple configuration file, implementing a sound experimental setup exploiting hardware counters in the host platform when available. MARTA also integrates fine-grained directives for instrumenting and monitoring small regions of code, enabling micro-benchmarking analysis. An important aspect of MARTA is to facilitate performance analysis and debugging: the toolkit applies data mining and machine learning or AI-based tech-

niques on the measurements, automatically extracting the features of the experimental setup which have the most impact on performance. These post-processing tasks are valuable for deriving knowledge from experiments, and are often not included in other profiling tools.

MARTA was conceived as a "push-button" system for profiling and performance analysis, basing its reliability on tested-and-true libraries and software. MARTA integrates the PolyBench/C library [103] for instrumenting codes using the PAPI library. It also relies on PolyBench/C directives for declaring and initializing n-dimensional arrays, as well as flushing the cache. The `pyperf` library is used for setting the processor frequency (and the turbo boost in Intel architectures). Data mining and machine learning algorithms primarily from scikit-learn are employed for performance data analysis. For the preprocessing stage, the Analyzer employs `pandas`, `numpy` and `KDEpy` Python libraries. Using this approach, MARTA benefits from the extensibility provided by integrating tested open source components, yet retains control over the profiling and performance analysis processes using a simple configuration file.

MARTA currently supports and targets both single- and multi-thread/core profiling tasks. Post-processing tasks have been optimized for data mining and basic ML classification, regression and clustering. MARTA does not currently implement deep learning algorithms as typically the small number of samples collected might not be sufficient for this type of AI techniques. However, our systematic export of data series to CSV format allows seamless integration with any other data mining or deep learning framework.

"Low-level programming is good for the programmer's soul."

–John Carmack

5

SIMD Optimizations: Random Vector Packing and Reduction Fusion

Chapter's contents

Modern optimizing compilers implement robust auto-vectorization techniques targeting rich SIMD ISAs. However, the usual practice is to generate machine-specific assembly code, which exploits the SIMD units of the target processor. In our work we take a different approach, by developing a source-to-source compilation framework targeting the automatic vectorization of specific loop regions. We implement vectorization using an Intrinsics-style approach to facilitate portability to a variety of concrete SIMD ISAs. With the help of MARTA, presented in Chapter 4, we develop machine-independent cost-driven algorithms to efficiently pack arbitrary or random operands and operations into SIMD vectors. Specifically, we support:

1) vector packing across multiple distinct loop nests to maximize vector occupancy, in particular when loops have a very small trip count; and 2) grouping and fusing reductions. We have therefore developed MACVETH, a novel source-to-source compiler implementing these SIMD optimizations. Experimental results are presented for a large set of vectorizable loop shapes, and for several key deep learning inference programs, demonstrating the benefits of random vector packing for efficient and portable vectorization of highly rectangular loops.

5.1 Overview and Motivation

Compilers' auto-vectorizers synthesize machine-specific assembly code exploiting SIMD units of a target processor. By default, most of these techniques are conservative, and they only apply if certain patterns are found in the code and a cost model assesses their profitability. Each compiler has its own algorithm to compute this cost in order to decide whether to vectorize a certain region of code or not. For instance, GNU GCC and Clang/LLVM use both Loop-Level Vectorization (LLV) and Superword-Level Parallelism (SLP) [38, 77, 99]. In the same way, both components use a similar approach in order to assess whether vectorization is profitable or not: unrolling loops using different vector factors, if possible, computing the cost of each vector instruction, and comparing the total cost with the scalar or not vectorized cost.

In an orthogonal dimension, the quality of SIMD code is affected by the knowledge of the target architecture. Some information regarding the performance of SIMD instructions, however, may be missing or non-disclosed by the manufacturer. These data can be used to determine whether a vector operation is beneficial or not, i.e., for building a cost model. There may be architectures using the same ISA and, therefore, where vectorization can be applied using exactly the same instructions, but, because of their micro-architectural designs, their performance might be completely different, leading to the synthesis of particular solutions for each case. The only way to disclose these features is by characterizing each architecture through micro-benchmarking. This process is tedious and complex since there are many ways of implementing the same macro-instruction on CISC architectures such as x86.

In this chapter we take an aggressive approach to auto-vectorization, by developing MACVETH, which stands for Multi-Architectural C-VEcTorizer for HPC applications. This is a Clang-based source-to-source compilation framework targeting the automatic vectorization of specific regions of code delimited by pragmas. We implement vectorization using a SIMD-Intrinsics' style approach, to facilitate portability to a variety of concrete SIMD ISAs. We develop platform-aware cost-driven algorithms to pack efficiently arbitrary operands and operations into SIMD vectors. For this purpose, we have also developed MRKVS (Mega-Random Kernel Vector SMT), a tool for generating candidate combinations of instructions to efficiently pack random elements into vector registers given a concrete ISA and a subset of instructions to consider. Equipped with this model, MACVETH supports vector packing across multiple distinct loop nests to maximize vector occupancy, in particular when loops have a very small trip count, targeting operations such as reductions. These codes are typically found in deep and machine learning kernels such as sparse tensor computations. Experimental results are presented for a large set of vectorizable loop shapes, and for several key deep learning inference programs, demonstrating the benefits of random vector packing for efficient and portable vectorization of these irregular codes.

The main contributions of the work presented in this chapter are described below:

- MRKVS: a novel SMT-based model and system for generating combinations of instructions given an ISA to gather and pack random memory positions within a vector register. The goal is to emulate the behavior of a gather instruction, but with optimal performance. First, we define and explore a search space for all the packing cases or equivalence classes defined by the number and contiguity of elements to pack, data type and vector width. Then, for each of these equivalence classes the system is able to generate up to a given number of candidates based on the maximum number of instructions to use.

- A platform-specific cost model derived from the candidates generated by the MRKVS system. This model is built by micro-benchmarking all these candidates for each possible case using the MARTA framework (detailed in Chapter 4), and selecting the most promising and profitable candidate for each platform.

- MACVETH: a Clang AST-based source-to-source compiler for vectorizing affine and irregular codes such as the Sparse Matrix-Vector Multiplication (SpMV). This compiler is able to vectorize multiple reductions within the same vector register, and to fuse independent reductions using the same vector operations, and even across multiple vectors. This solution also includes the platform-aware random packing combinations described above, for efficiently packing random operands in the same vector.

The high-level picture of the toolchain proposed in this chapter is shown in Figure 5.1. The only input to the system is a concrete C/C++ file with marked regions to be considered for vectorization. The output is a SIMD version of that code, if the cost model predicts its profitability; otherwise it just emits scalar code. For simplicity, we focus on x86 architectures with AVX2 only, but the same approach can be applied to other architectures and ISAs.

The rest of the chapter is structured as follows: Section 5.2 describes our approach for generating efficient SIMD code for random vector packing by introducing MRKVS, the SMT-based system proposed for generating random packing combinations. Section 5.3 describes MACVETH in detail, the source-to-source compiler implementing the efficient random packing and the SIMD optimizations for reductions in sparse codes. In Section 5.4 we present a set of experiments to assess the quality of the solutions and use cases of our compiler. Section 5.5 explores the state of the art and the related work regarding compiler support for auto-vectorization

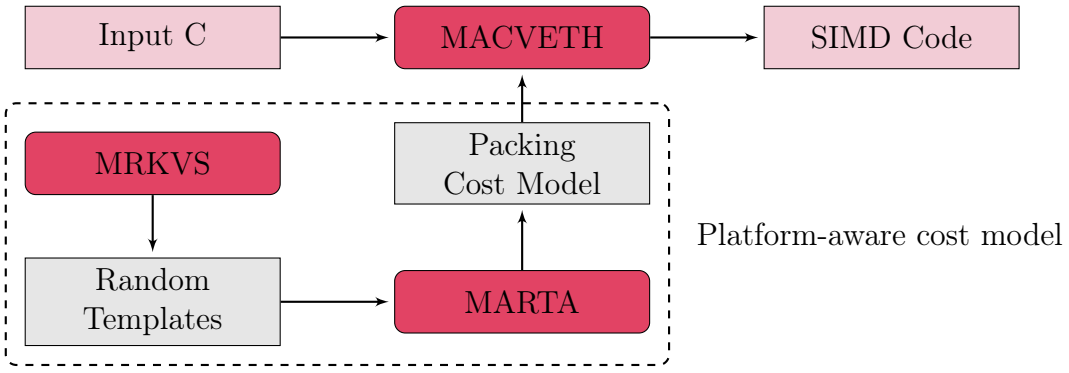


Figure 5.1: High-level picture of the inter-operation between the components presented in the Thesis.

and vectorizers. Finally, we end the chapter with final remarks and discussion in Section 5.6.

5.2 Efficient Random Vector Packing

Gather is an x86 instruction introduced in AVX2 meant to pack a set of points or memory addresses into a vector register in one macro-instruction. To be more precise, gather emits only one instruction, but it is decoded into many micro-operations by the front-end of the core pipeline. The gather instruction receives as input a base address, a scaling factor, and a vector of indices representing the offset between the elements to gather (each index is scaled by the factor). Elements are loaded from addresses starting at the base address and offset by each element in the index register (which are also scaled by a certain factor). The gather instruction presents two significant limitations: 1) the need to compute a vector of indices for describing the positions of the elements to pack, and 2) the requirement of an initial address to compute the offsets of the points to gather. In terms of performance, as described in Section 4.5.1, gather's latency depends on the number of cache lines that are touched by the memory addresses, thus it depends on the memory latency of the system when the cache is cold.

As mentioned before, gather is decoded into many micro-operations, a number that varies depending on the architecture, e.g., in modern AMD architectures, such as Zen2 and Zen3, it can be decoded into more than 30 micro-operations[1], depending on the vector width. This results in the number of retired instructions being higher than that of issued instructions, which is not desirable for energy-aware applications, for instance. As such, gather can be a good idiom for code-size-aware applications, but it is not the best solution in terms of performance when it comes to throughput[2]. Actually, even though gather instruction can save machine code bytes for the L1 instruction cache, it could take up more space in the μop-cache (or L0 cache) than other solutions due to its decoding phase. The μop-cache is a specialized cache for storing pre-decoded micro-operations [119], improving the performance in the

[1] According to values reported in https://uops.info/table.html ([1, 31]).

[2] Typically, throughput in CPU is measured as instructions per cycle, being the reciprocal throughput cycles per instruction. This can be applied to an instruction, a set of instructions, etc.

decoding phase. It is present in modern architectures and it has a significant impact on performance [20], as it allows the decoder to idle when reusing micro-operations.

Efficient random vector packing means combining single instructions available in the ISA to emulate the behavior of the gather instruction. As carefully detailed above, this might be more code-size consuming, but reciprocal throughput can be outperformed. Gather is a valid solution for any set of points starting at a base address, regardless of their contiguity and strides. However, focusing on all the possible combinations for packing N random memory points into a vector according to their contiguity in memory opens opportunities to outperform gather's latency and throughput. This is the motivation for our novel model: create a system for generating random vector packing candidates based on the instructions available in the ISA.

Next, in Section 5.2.1 we describe the instructions included in the exploration space, then we define a formal model to describe the equivalence classes for generating candidates in Section 5.2.2, we detail the SMT-based system for generating these candidates MRKVS (Mega-Random Kernel Vector SMT) in Section 5.2.3, in Section 5.2.4 we introduce a template-based format for using the candidates generated, and finally we evaluate the performance of these candidates against the gather instruction in different platforms in Section 5.2.5.

5.2.1 Instruction set: exploration space

In order to provide alternative implementations to the gather instruction, we must first define the set of instructions to consider. Modern vector ISA extensions in x86, such as AVX2, implement load and swizzle instructions, among others. The x86 SIMD extensions are very fragmented, as they have been evolving over the years and the architectures must be backwards compatible. This causes some performance issues in older architectures when mixing different SIMD ISAs in the codes, such as SSE and AVX, since their encoding is different. Nowadays, modern architectures solve this issue by VEX (Vector EXtensions) encoding all these vector instructions. It is remarkable that non-VEX instructions (e.g., SSE `movaps`) are transformed into prefix-VEX instructions (i.e., `vmovaps`) when using `-mavx` or more recent SIMD flags.

Next, and for simplicity, we present the subset of Intrinsics that we consider for modeling our approach, only using load and swizzle instructions for floating-point data types, and using only up to AVX2 (we do not consider AVX-512). We also describe the limitations of the Intrinsics and a small extension of the Intrinsics to consider in our model. In this way, the union of these subsets defines the set of instructions within the exploration space $\mathcal{I}$.

Load instructions

These instructions bring data from memory to vector registers. Instructions of this kind considered for our model are described in Table B.1 (Appendix B). Each of the load instructions has only one possible output if they do not use any kind of mask or indices, e.g., in Table B.1 the "`movaps xmm, MEM`" instruction loads 4 contiguous positions in memory starting at a given address, and therefore it has no other possible output. The only exceptions are masked loads (`_mm_maskload_ps()` and `_mm256_maskload_ps()`), which also use a mask for zeroing vector slots when loading from memory.

Swizzle instructions

Swizzle instructions from Intel Intrinsics are meant to pack/unpack, shuffle, blend and permute vector slots between vector registers. These instructions can only have vector registers or immediate values as operands. Instructions of this kind considered for our model are described in Table B.2. Typically, shuffle and permute are just different names for the same type of operations: those which rearrange elements according to a control value. Shuffle was the default naming until AVX, and permute was used afterwards. Blend instructions, as the name suggests, are meant to fuse the values of two different registers according to a control parameter. Extract operations obtain concrete positions of a register, while insert operations place the content of a vector slot into another. With all these operations we cover all the necessities for packing memory positions into vector registers.

```
1  #ifndef _CUSTOM_SIMD_H
2  #define _CUSTOM_SIMD_H
3
4  #define _mv_insert_mem_ps(I, O, VAL, POS)                                \
5          __asm volatile("vinsertps %1,%2,%3,%0\n"                         \
6                         : "+x"(O)                                         \
7                         : "i"(POS), "m"(VAL), "x"(I));
8
9  #define _mv_blend_mem_ps(O, A, M, I)                                     \
10         __asm volatile("vblendps %1,%2,%3,%0\n"                          \
11                        : "+x"(O)                                         \
12                        : "i"(I), "m"(M), "x"(A));
13
14 #endif // !_CUSTOM_SIMD_H
```

Listing 5.1: Ad hoc SIMD instructions to consider in our model.

Limitations of Intel Intrinsics

There are some Intrinsics that only allow register operands. Most of them have a direct translation into assembly instructions, e.g., `_mm_insert_ps()`, which basically inserts a value from one register into a concrete position of another. This generates a `vinsertps` instruction (VEX-prefixed, `insertps` in original SSE). Nonetheless, this instruction has `xmm1, xmm2, xmm3/m32, imm8` as operands, thus allowing the third operand to be a memory address. This is not part of the original Intel Intrinsics API and can be useful for merging the load and the insert operations. An example of a header file containing these new added instructions to the model is presented in Listing 5.1. In this case we just define new macros to inline the assembler.

5.2.2 Simplifying the search space

Once we have chosen the set of instructions used by our model, we need to establish rules and relations between them, as well as their semantics. First of all, Equation 5.1 represents the content of the target vectors $\vec{V}$, where their elements are not contiguous in memory. Since we are targeting the x86 architecture, registers and vectors in our notation are represented using a little-endian format. Any memory address p plus one refers to the next memory address for elements of the same type,

i.e., $p_{i-1} + 1 = p_i$. This concept is similar to pointer arithmetic.

$$\begin{aligned} &\vec{V} := \{v_{n-1}, \ldots, v_0\} \\ &\exists i > 0, v_i, v_{i-1} \in V / v_i \neq v_{i-1} + 1 \end{aligned} \tag{5.1}$$

Using a similar notation, we can define each of the load instructions as a function that, for a given address p, returns a set of contiguous positions in memory using a little-endian format, as in Equation 5.2.

$$f(p) := \{v_{n-1} = f(p[n-1]), ..., v_0 = f(p[0])\} \tag{5.2}$$

This generic function is specialized according to the semantics of each Intrinsics regarding its documentation. For instance, `_mm_loadu_ps(p)` in our model is formalized as in Equation 5.3.

$$loadu\_ps(p) := \{v_3 = p[3], v_2 = p[2], v_1 = p[1], v_0 = p[0]\} \tag{5.3}$$

Using the same type of specialization, the function for `_mm_load_ss(p)` is described as in Equation 5.4.

$$load\_ss(p) := \{v_3 = \emptyset, v_2 = \emptyset, v_1 = \emptyset, v_0 = p[0]\} \tag{5.4}$$

Again, this is only another way of representing the behavior of the instructions selected in the exploration space $\mathcal{I}$ (defined in Section 5.2.1) according to their documentation. So, representing swizzle instructions in our approach can be done as in Equation 5.5 for the `_mm_shuffle_ps(a,b,m)` instruction.

$$\begin{aligned} shuffle\_ps(a, b, m) := \{&v_3 = f(b, m[7:6]), \\ &v_2 = f(b, m[5:4]), \\ &v_1 = f(a, m[3:2]), \\ &v_0 = f(a, m[1:0])\} \\ &where \\ f(src, c) := &\, src[31 + 32 * c : 32 * c] \end{aligned} \tag{5.5}$$

However, in this case, since the output of this instruction depends on the value of the mask m, we would need to compute all possible outputs for all possible mask values (256 different values since the mask width is 8 bits), as in Equation 5.6.

$$\begin{aligned} &shuffle\_000\_ps(a,b) := \{v_3 = b[0], v_2 = b[0], v_1 = a[0], v_0 = a[0]\} \\ &shuffle\_001\_ps(a,b) := \{v_3 = b[0], v_2 = b[0], v_1 = a[0], v_0 = a[1]\} \\ &\vdots \\ &shuffle\_255\_ps(a,b) := \{v_3 = b[3], v_2 = b[3], v_1 = a[3], v_0 = a[3]\} \end{aligned} \tag{5.6}$$

This approach has an obvious drawback: the exploration space created by generating all possible combinations of the instructions grows exponentially and it quickly becomes intractable. In fact, exploring all combinations of memory addresses and their possible packing candidates for getting the optimal recipe is an NP-hard problem, so any purely brute force methodology is intractable. On the other hand, it is possible to tackle this issue by properly defining the exploration space to traverse, and applying heuristics to prune the set of candidates to combine in each step.

We have already defined the set of finite instructions $\mathcal{I}$ in our model. The combinations of these instructions create the infinite space $\mathcal{R}$, so we must define the finite set of combinations to consider, $\mathcal{R}' \subset \mathcal{R}$. The space $\mathcal{R}$ is infinite by definition, as the combinations of instructions for a concrete set of memory addresses are infinite, e.g., by repeating instructions, using load instructions that overlap with other memory addresses in the target, etc. Therefore, this space is also intractable, mostly redundant, and contains a vast majority of non-optimal candidates. Let $\mathcal{P} \subset \mathbb{Z}^n$ be the intractable set of points in memory that can be packed into a vector register. $\vec{P} \in \mathcal{P}$ is defined as $\vec{P} := \{p_{n-1}, \ldots, p_0\}$, where p_i represents a memory address. We define $\mathcal{S}$ as the finite and tractable set of equivalence classes for the vectors $\vec{P} \in \mathcal{P}$. In the same manner, we can define a surjective function f such that for any of these vectors $\vec{P}$, the output S_P will be defined in the set $S_P \in \mathcal{S}$, i.e., $f : \mathcal{P} \to \mathcal{S}$. On the other hand, for each of those equivalences $S_P \in \mathcal{S}$, we can define an injective function with multiple outputs g that generates all possible assembly output candidates in the finite space generated in $\mathcal{R}' \subset \mathcal{R}$, i.e., $g : \mathcal{S} \to \mathcal{R}'$. The idea of these concatenated functions is graphically described in Figure 5.2, and attempts to simplify the intractable infinite space onto a tractable finite equivalent.

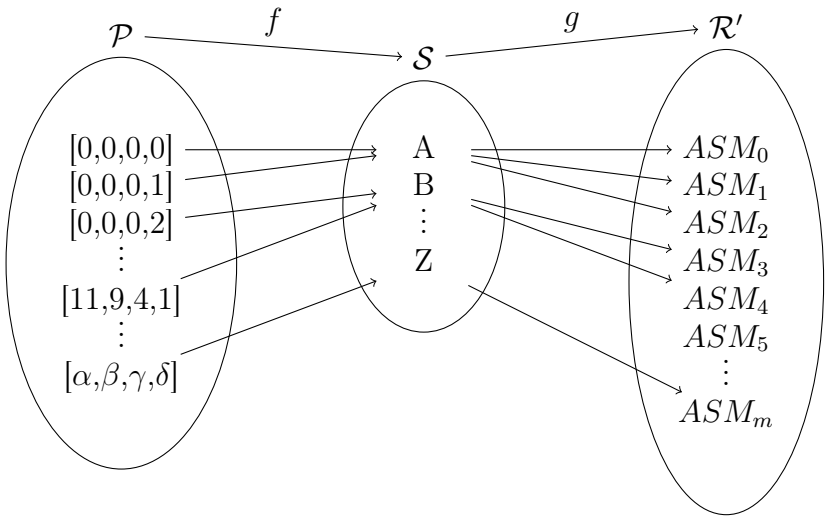


Figure 5.2: Functions and sets defined in the system. $\mathcal{P}$ is the intractable set of combinations of memory points, $\mathcal{S}$ the set of equivalence classes for all of them, and $\mathcal{R}'$ the pruned or finite version of $\mathcal{R}$, containing only a subset of all possible solutions or candidates. This is just a simplification for packing 4 memory addresses.

By definition $\forall \vec{P} \in \mathcal{P}, \exists f(\vec{P}) \in \mathcal{S}$, but most important $\exists \mathcal{P}' \subseteq \mathcal{P} / \forall \vec{P_i} \in \mathcal{P}'$, $f(\vec{P_0}) = f(\vec{P_1}) = \ldots = f(\vec{P_i}) = S_l \in \mathcal{S}$, that could be rewritten as $\{\vec{P_i} \in \mathcal{P}' \subseteq \mathcal{P} : f(\vec{P_i}) \sim S_l \in \mathcal{S}\}$. So, in order to define f, we need to define first what is going to be an equivalence in the system.

We define an equivalence class based on: 1) the contiguity between adjacent elements $\vec{D}$, 2) the vector width W, and 3) the data type T. In this way, we could define $[f(\vec{P_i})] = S_l = \{\vec{D}, W, T\}$, having $|\vec{P_i}| = n$, $\vec{D} \in \mathbb{Z}^{n-1}$, and $d_j \in \vec{D} / d_j = p_{j+1} \oplus p_j$, where the operator $\oplus$ for two points returns 1 if memory addresses are contiguous and 0 otherwise. In order to illustrate this function, we show some examples using all floats for 128-bit vector registers as follows:

- $\vec{P} = [A[40], A[30], A[10], A[0]], f(\vec{P}) = \{\{0, 0, 0\}, 128, float\}$
- $\vec{P} = [A[41], A[30], A[10], A[0]], f(\vec{P}) = \{\{0, 0, 0\}, 128, float\}$
- $\vec{P} = [A[41], A[30], A[1], A[0]], f(\vec{P}) = \{\{0, 0, 1\}, 128, float\}$
- $\vec{P} = [A[4], A[3], A[1], A[0]], f(\vec{P}) = \{\{1, 0, 1\}, 128, float\}$

- $\vec{P} = [A[132], A[131], A[129], A[128]], f(\vec{P}) = \{\{1, 0, 1\}, 128, float\}$

The main goal of this definition is to extract the features of the elements to pack by being memory agnostic and, at the same time, simplify the possible set of cases to consider. Other features of the problems could be added to the definition, such as the number of cache lines or the number of cache sets touched, but they would only increase the size of the space $\mathcal{S}$, and we want to keep it as simple as possible. In this concrete case, this simplification is driven by the x86 ISA, since we can leverage the existence of contiguous memory addresses for being more efficient by issuing fewer instructions. Similarly, in our case, these features would not contribute to the quality of the solutions, as we are only interested in minimizing the number of instructions executed and their latency, and maximizing the throughput.

5.2.3 MRKVS: Mega-Random Kernel Vector SMT

We have defined a finite space of instructions to consider, represented by $\mathcal{I}$, and a finite and tractable set of equivalence classes $\mathcal{S}$ to target. Even though these sets have a limited number of elements, brute force approaches become quickly intractable when increasing the number of elements to pack. E.g., an enumeration-based brute force approach could be suitable for packing up to four 32-bit random memory positions, as the system would only use a small subset $\mathcal{I}' \subset \mathcal{I}$ because in this case 256-bit instructions are useless. But just increasing the number of elements to pack to five makes the problem intractable in terms of execution time due to the exponential explosion in the number of combinations. Therefore, brute force approaches are not suitable for finding all cases in $\mathcal{S}$ regardless of the number of elements to pack.

As described above, one of the issues when enumerating all the candidates in $\mathcal{I}$ is the combinatorial explosion of the number of variants of a single instruction according to its masks or control value, as shown in Equation 5.5. Instead of generating and testing all these possible combinations, a smarter strategy would be to check whether there is any value that for a combination of instructions is able to meet the packing conditions, i.e., the packing of memory points into a vector register in

a certain order. To better illustrate this, consider the vector registers below:

$$a = \{a_3, a_2, a_1, a_0\}$$
$$b = \{b_3, b_2, b_1, b_0\}$$
$$c = \{b_1, b_0, a_1, a_0\}$$

Assuming that `a` and `b` are vectors of float type, and `m` is the control value, we want to find the value `m` such that `c = _mm_shuffle_ps(a,b,m)`. The most naïve approach would be to try all possible candidates for the mask as in Equation 5.6 until we find a value that satisfies the equality. A better approach would be to devise a system capable of analytically deriving whether the equality can be satisfied and, in that case, providing the appropriate mask value. Determining whether a formula can be satisfied or not is studied by Satisfiability Modulo Theories (SMT), which is the problem that, roughly, generalizes the boolean satisfiability problem (SAT). We say that a formula is satisfiable if there exists a set of values of its variables such that the formula evaluates to true. A well-known implementation of SMT is the Z3 Theorem Prover [23].

Wegner developed x86-sat [131], a system for building an auto-generated formal model of x86 Intrinsics by interpreting the pseudo-code in the official documentation, and transforming it into a valid model for Z3. This tool is mainly written in Python, and it can help assess the equivalence of two different ways of permuting values, i.e., to find the equivalence classes, or to find the values of some variables in a formula. For illustrative purposes, Listing 5.2 shows the solution to the example described above using this system.

x86-sat works as follows. A set of assertions or conditions describing the behavior of each instruction according to the documentation are added to the solver using the Z3 library (a custom parser is used to automate this process [130]). Next, the `check` function (line 8 in Listing 5.2) tests if those conditions can be satisfied for those variables or instructions and, in that case, the system also returns a model containing the values required. This system avoids testing all different control value combinations for a concrete instruction. The system is also interesting for performing sanity checks given a set of instructions and the conditions to be satisfied, or even for finding bugs in the documentation. In addition, this system can be easily extended with any other desirable instruction by just using the same syntax as in the Intel

```
# after initializing the library and setting a and b values
# the system generates the functions with the same function
# signatures as in Intel Intrinsics API
a = _mm_set_ps(a3, a2, a1, a0)
b = _mm_set_ps(b3, b2, b1, b0)
c = _mm_set_ps(b1, b0, a1, a0)
imm = Var("imm", "int")
condition, model = check(c == _mm_shuffle_ps(a, b, imm))
# RESULTS:
# condition = sat
# model = {'imm': '0x00000044'}
```

Listing 5.2: Python code needed to compute the value of an immediate control value using the x86-sat system for the previous example.

Intrinsics documentation.

In our approach we leverage the capabilities of the system developed by Wegner for chaining different instructions and testing their satisfiability for each equivalence class described in Section 5.2.2. The system searches the solution space as described in Algorithm 5.1. Our approach follows a deep-first fashion, with a limited number of levels, and where the level is determined by the number of chained instructions used. In this approach the system takes as input an object PackClass S_l, which corresponds to an equivalence class in $S_l \in \mathcal{S}$. The algorithm is a modification of the naïve brute force approach, where the explosion of new combinations is minimized by applying heuristics to prune the set of new candidates, i.e., new instructions to consider, and the use of the SMT system to parameterize and check the satisfiability of the chain of instructions. The system has also a variable stop condition when the number of candidates found has reached, at least, $max\_candidates$.

As shown in Algorithm 5.1, the exploration starts choosing only the relevant load instructions in our space $\mathcal{I}$. It is not worth to consider only swizzle instructions at this stage, as we have not loaded any memory address in any registers yet, so this way we avoid the generation of this useless space. Next, for each of these first load instructions, we generate recursively the exploration of the space considering also the rest of instructions. Algorithm 5.2 illustrates the recursive exploration performed in our approach. It first tests all possible candidate instructions, appending to a list those ones that produce a candidate that satisfies the pack class. Those lists

Algorithm 5.1: High-level approach of the MRKVS system.

```
   Input: Instructions I, PackClass S_l, int max_candidates
   Result: Set of Candidates
 1 candidates = {};
 2 max_ins = compute_max_ins(S_l);
 3 load_ins = prune_load_instructions(S_l, I);
 4 for load in load_ins do
 5 |   if check(load, S_l) then
   |   |   // Base case:  only a load required, no need to explore
 6 |   |   candidates.append(load);
 7 |   |   continue;
 8 |   if new_candidate = recursive_search(load, I, S_l, max_ins) then
 9 |   |   candidates.append(new_candidate);
10 |   if size(candidates) >= max_candidates then
11 |   |   break;
12 end
13 return candidates;
```

of instructions that do not satisfy the packing S_l are later explored using the same approach, in a recursive manner. The system stops generating new candidates when they exceed a threshold in the number of instructions used, making the problem tractable without sacrificing any optimal candidate. We could argue that, as we increase the number of instructions used, the opportunities for finding new optimal or better candidates decrease. This statement is reasonable in our context as we try to minimize the latency and the total number of instructions retired in order to outperform the gather instruction.

It is remarkable that candidate instructions are pruned depending on their type and number in each recursive step, in order to reduce the node explosion as we create new levels in the exploration space. These pruning techniques are ad hoc and dependent on the SA, i.e., the set $\mathcal{I}$, which in our case is a small subset of the AVX2 ISA. Some of these techniques involve limiting the maximum number of instructions of a type to consider in a combination, avoiding the repetition of costly instructions (such as masked loads or blends) more than once for each candidate, or avoiding the use of load instructions to load only one element more than twice.

Algorithm 5.2: *recursive_search* function used to generate the exploration space in MRKVS.

```
Input: Candidate C, Instructions I, PackClass S_l, int max_ins
Result: Set of Candidates
candidates = {};
new_level = {};
C' = {};
candidate_instructions = prune_instructions(S_l, I);
for new_instruction in candidate_instructions do
    C' = chain(C, new_instruction);
    if check(C', S_l) then
        candidates.append(C');
    else
        new_level.append(C');
    end
end
if size(C') + 1 > max_ins then
    return candidates;
end
for candidate in new_level do
    if C' = recursive_search(candidate, I, S_l, max_ins) then
        candidates.append(C');
    end
end
return candidates;
```

5.2.4 Random vector packing templates: format

In order to make these candidates portable to any memory address, we have developed a template-based format to capture their semantics and be input agnostic. In this way, any system can fill any of the candidates generated by MRKVS with the desired input memory addresses or vector registers for packing the operands. These templates are in MACVETH Random Template format (`.mrt`). This extension was meant for the MACVETH compiler, presented in Section 5.3, which leverages these templates to pack random operands filling the input values with the corresponding memory addresses from the code. The syntax of the template is quite simple, where

each line can be expressed in an extended Backus-Naur form as:

$$
\begin{aligned}
\langle \text{syntax} \rangle &\models \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \, \langle \text{syntax} \rangle \\
\langle \text{stmt} \rangle &\models \langle \text{reg} \rangle = \langle \text{inst} \rangle \mid \langle \text{inst} \rangle \mid \lambda \\
\langle \text{inst} \rangle &\models \mathit{signature}(\langle \text{args} \rangle) \\
\langle \text{args} \rangle &\models \langle \text{reg} \rangle \mid \langle \text{mem} \rangle \mid \{\langle \text{args} \rangle\} \\
\langle \text{mem} \rangle &\models \langle \text{index} \rangle : \langle \text{offset} \rangle : MEM \\
\langle \text{reg} \rangle &\models \langle \text{name} \rangle : REG \\
\langle \text{index} \rangle &\models \{\texttt{digit}\} \\
\langle \text{offset} \rangle &\models [-]\{\texttt{digit}\}
\end{aligned}
$$

For indexing these files, each candidate generated is stored in a file with the named as: `<arch>_<isa>_<data_type>_n<#values>_<contiguity>.mrt`. As an example, for Intel Cascade Lake with AVX2 using floats, for packing 4 elements according to the contiguity pattern of all elements scattered (remember the notation $S_l = \{(0,0,0), 128, float\}$), the system uses the template in Listing 5.3. MACVETH replaces the values within the hashtags (`#...#`) with the corresponding values for each case. For instance, the `REG` values are assigned to vector registers, whereas the `MEM` placeholders are assigned to memory addresses deduced from the memory positions to pack.

5.2.5 Generation and evaluation of the cost model

We have used MRKVS to generate random vector packing formulas for AVX2 with floats as data type. The model, as we have discussed before, can be easily

```
1 #r0:REG# = _mm_load_ss(&#0:0:MEM#)
2 _mv_insert_mem_ps(#r1:REG#, #r0:REG#, #1:0:MEM#, 0x0000001c)
3 _mv_insert_mem_ps(#r2:REG#, #r1:REG#, #2:0:MEM#, 0x00000068)
4 _mv_insert_mem_ps(#output:REG#, #r2:REG#, #3:0:MEM#, 0x00000030)
```

Listing 5.3: Example of the `cascadelake_avx2_float_n4_0_0_0.mrt` template.

extended to other ISAs and data types. The output of the SMT-based system is a set of candidates for each equivalence class. As we have described in Section 5.2.2, each equivalence class is defined, in our case, by the contiguity of the memory addresses to pack, vector width and data type. An example of the possible output of the system for AVX2, given the equivalence class for packing only three non-contiguous elements $S_l = \{(0, 0), 128, float\}$, could be the candidates of Listing 5.4 (using the format described in Section 5.2.4), for a given pointer p and the unknown indices `IDX0`, `IDX1` and `IDX2`.

The compilation of these candidates using GCC 11.2.0 on two different architectures, Intel Xeon Silver 4216 (Cascade Lake) and AMD Ryzen 9 5950X (Zen3), produces the assembly code in Listing 5.5. In the Zen3 processor both choices retire the same number of micro-operations, and the cycles consumed are on average the same. Regarding the Intel machine both candidates also have identical performance in terms of cycles, but the number of micro-operations retired is lower for the first candidate. These are the two metrics considered for building our cost model: in the first place the number of execution cycles or cycles consumed, and in case of identical performance (within an error margin due to measurement errors), the number of micro-operations retired. Results obtained in LLVM-MCA (see Section 4.1 and Figure 4.1) for these candidates confirm the values reported by our empirical measurements. The recently released uiCA tool [3] also reports better reciprocal throughput (i.e., cycles per instruction) for the first candidate.

Once we have chosen the best candidates for each platform and for each equivalence class, in order to assess their quality, we compare their performance with that of the equivalent gather instruction. We run these in the same Intel Xeon Silver 4216 and AMD Ryzen 9 5950X machines mentioned above, using AVX2 extensions. The experiments were performed under hot cache conditions, as we are only interested in the latency in cycles due to the instruction pipeline and the number of micro-operations retired. Besides, running these experiments under cold cache reports almost identical latencies for our approach and gather, as memory latency dominates the execution of the block of instructions.

According to our measurements, for Intel Cascade Lake, most of the candidates proposed by our automated system outperform gather in terms of number of execution cycles, as depicted in Figure 5.3a. However, for less than 15% of the cases,

```
1  // Candidate 0
2  __m128 r0, r1, output;
3  r0 = _mm_load_ss(&p[IDX0 + (0)]);
4  _mv_insert_mem_ps(r1, r0, p[IDX1 + (0)], 0x00000014);
5  _mv_insert_mem_ps(output, r1, p[IDX2 + (0)], 0x00000068);
6
7  // Candidate 1
8  __m128 r0, r1, output;
9  _mv_blend_mem_ps(r0, _mm_set_ps(0,0,0,0), p[IDX0 + (0)], 0x0000000b);
10 _mv_insert_mem_ps(r1, r0, p[IDX1 + (0)], 0x00000018);
11 _mv_insert_mem_ps(output, r1, p[IDX2 + (0)], 0x00000028);
```

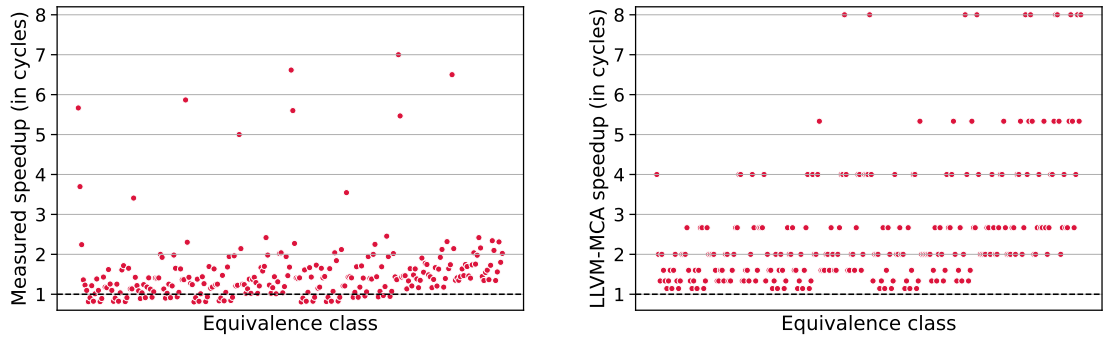
Listing 5.4: Candidates generated by MRKVS for packing three non-contiguous elements. Instructions with `_mv` prefix are described in Listing 5.1.

```
1  ;; Candidate 0
2  vmovss      xmm2, DWORD PTR [r12 + 0x40]
3  vinsertps   xmm1, xmm2, DWORD PTR [rcx], 0x14
4  vinsertps   xmm0, xmm1, DWORD PTR [rdx], 0x68
5
6  ;; Candidate 1
7  vblendps    xmm1, xmm3, XMMWORD PTR [rsi], 0xb
8  vinsertps   xmm2, xmm1, DWORD PTR [rcx], 0x18
9  vinsertps   xmm0, xmm2, DWORD PTR [rdx], 0x28
```

Listing 5.5: Assembly code generated for the example in Listing 5.4.

gather outperforms by 10-15% the latency of our approach. In contrast, according to LLVM-MCA (see Figure 5.3b) all candidates outperform the gather instruction. Our cost model is driven by our measurements and, as such, it will use the gather instruction for those equivalence classes where there is no speedup from the candidates generated.

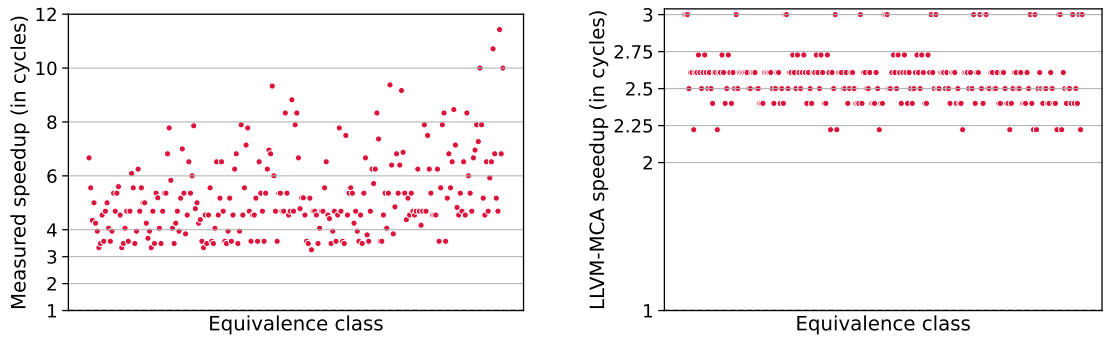
For AMD Zen3 both our measurements and the LLVM-MCA tool report significant speedups for the candidates proposed against their gather counterpart, as depicted in Figures 5.4a and 5.4b. This is because: 1) the set of instructions used for the candidates has, in general, a reciprocal throughput of less than 1 (and thus more than one micro-instruction can be issued in the same cycle), and 2) the reciprocal throughput for the gather instruction is far worse on AMD than on Intel:



(a) Measured cycles: comparison between gather and our approach.

(b) LLVM-MCA cycles: comparison between gather and our approach.

Figure 5.3: Speedups (in cycles) obtained for Intel Cascade Lake. The baseline (value 1.0) is the gather instruction.



(a) Measured cycles: comparison between gather and our approach.

(b) LLVM-MCA cycles: comparison between gather and our approach.

Figure 5.4: Speedups (in cycles) obtained for AMD Zen3. The baseline (value 1.0) is the gather instruction.

8 vs. 4 cycles/instruction for AMD and Intel, respectively[3]. Note that, as commented at the beginning of the Section 5.2, the number of micro-operations retired on AMD for each gather instruction, depending on the vector width, can be up to 40 micro-operations.

In order to recapitulate the contributions presented in this Section 5.2, to perform random vector packing efficiently, first we have generated, using an SMT-based system, a large set of candidates for each equivalence class from the exploration search space $\mathcal{I}$ (see Section 5.2.1). We expressed these equivalence classes based on the contiguity in memory of the elements to pack into the same vector register (see Section 5.2.2). Next, we have chosen the best candidates for each platform based on the performance measured in terms of cycles and micro-operations of the candidates synthesized, and we have compared their latency against the gather instruction. We consider the use of gather instead of our candidates if they do not show speedup for their equivalence classes. In this way, we have built a platform-independent cost model for each architecture.

5.3 MACVETH: Multi-Architectural C-VEcTorizer for HPC applications

We have developed MACVETH (Multi-Architectural C-VEcTorizer for HPC applications), a source-to-source compiler to apply automatic vectorization based on the cost models extracted in the previous section. The input to the compiler is a C/C++ code, and the output a SIMD version of it. This SIMD code is generated using an Intrinsics style. In this way, the code is portable among processors and architectures with the same vector extensions, such as SSE or AVX2. MACVETH integrates a built-in platform-aware cost model for synthesizing the best code for each architecture. For instance, the cost model for packing random vector operands is generated by running all the candidates described in Section 5.2.3 on each platform and choosing the best one for each architecture. MACVETH also targets the grouping and fusion of independent reductions, patterns that appear in codes such as SpMV kernels, typically present in deep and machine learning applications. The

[3]According to values reported in `https://uops.info/table.html` ([1, 31]).

main goal of the optimizations presented here is to maximize the vector occupancy and increase the efficiency of the code.

In this section the MACVETH compiler and the SIMD optimizations implemented are described in detail. Section 5.3.1 introduces the key architectural components of compilers and the LLVM framework. Section 5.3.2 describes the high-level details of the MACVETH architecture. Section 5.3.3 focuses on the design and use of the front-end, Section 5.3.4 details the optimizations performed in the middle-end, and Section 5.3.5 covers the generation of code and peephole optimizations in the back-end. Finally, Section 5.3.6 enumerates the current limitations of the MACVETH compiler.

5.3.1 Compiler architecture: the LLVM Project

Typically, compilers translate high-level programming languages (e.g., C, C++) to a lower level language (e.g., machine code) to generate executable programs. They also perform transformations in the code in order to optimize the output by applying many different techniques, such as vectorization, removing useless or unreachable code (dead code elimination, DCE), performing loop transformations, etc. In our case, the approach is different as our compiler does not generate machine code directly, but a custom SIMD version of the input code. These compilers are typically known as transcompilers, transpilers or just source-to-source compilers. They have different use cases: translating certain codes into other idioms or languages, replacing patterns in the code with other templates, placing pragmas in specific regions of the code, fixing bugs in the code, etc. As depicted in Figure 5.5, compilers have three main parts:

- *Front-end*: this part is in charge of recognizing the correctness of a program, identifying and reporting errors in a useful way. This is done by the lexer and the parser. The output of the parser is, typically, an Abstract Syntax Tree (AST) that holds the grammatical information of the program. Then, the semantic analyzer performs type checking, and variable and function declaration. The final output of the front-end is an Intermediate Representation (IR) of the program, in order to shape the code for the following stages.

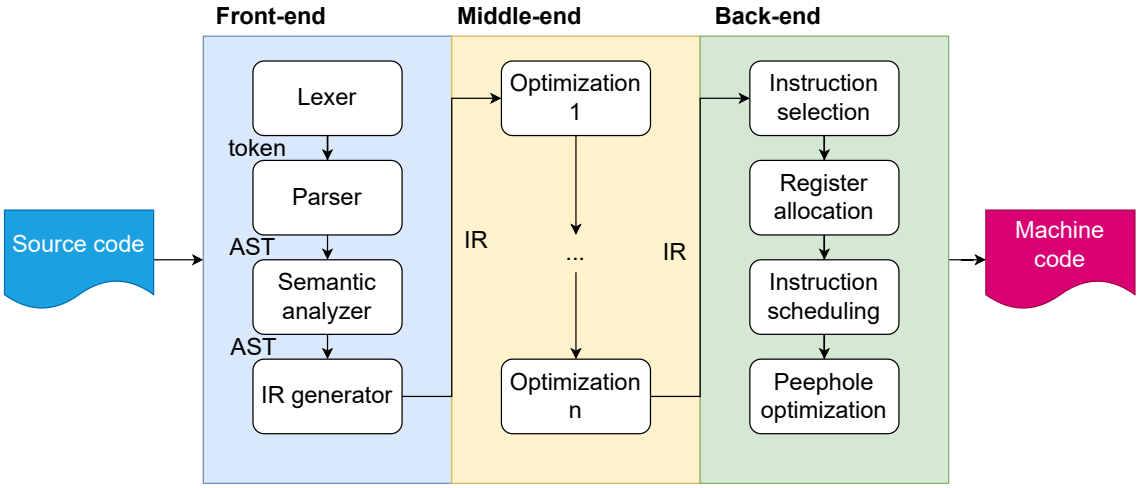


Figure 5.5: Classic high-level diagram of the compiler architecture and its phases for code generation.

- *Middle-end*: this component is meant to perform optimizations in the code. Modern compilers organize optimizations in different passes, which can be applied to the input IR varying the order and the number of times that they are applied. The only condition for any pass is that it must preserve the semantics of the code.

- *Back-end*: the main goal is to translate the IR into machine code (or the target language). It decides the set of instructions or directives to synthesize for the IR operations. The instruction selection block is usually viewed as a pattern matching problem. The register allocation is in charge of synthesizing values in registers analyzing their liveness within the program, and the instruction scheduling attemps to optimize the order of the instructions to be issued (without affecting the meaning of the program). Since different architectures have their own hardware features, other optimizations can also be applied targeting them. These are typically known as peephole optimizations.

As will be detailed in Section 5.3.2, the implementation of MACVETH is based on LLVM [72], which is an umbrella project of several compiler, toolchain and low-level technologies. LLVM was conceived and designed as a set of reusable and modular libraries with well-defined interfaces. LLVM also integrates Clang, its native C/C++/C-Objective front-end, which provides powerful libraries for developing

and extending new tools. For instance, its library LibTooling is designed to help write standalone tools using the Clang AST [53]. This AST is the most important primitive in Clang to extend the logic in the front-end. Note that Clang is not only the C-like front-end of LLVM, but it also provides a compiler driver, which basically concatenates all the phases required to compile the source code using the LLVM modules (e.g., LLVM optimizer passes, the static compiler, the linker, etc.). Figure 5.6 illustrates a high-level toolchain for generating an executable program from source code using the Clang driver. The Clang AST is an abstraction that can be lowered to the LLVM IR to perform optimizations. The LLVM IR is a Static Single Assignment (SSA) [109] based representation, which simplifies a vast number of compiler optimizations. Continuing in the LLVM toolchain, the Selection DAG[4], which is the transition between the middle-end and the back-end, provides an abstraction for code representation in a way that is suitable for instruction selection and instruction scheduling. In the last block, the LLVM Machine IR (MIR) is just another IR for the machine code, which is then synthesized according to the target platform (e.g., x86, AArch64, RISC-V, ARM, MIPS, PowerPC, etc.). In this way, LLVM provides a complete and mature toolchain for parsing, analyzing, transforming, and properly compiling source code. In our case we focus on the Clang front-end and its principal IR, the Clang AST.

The Clang front-end also integrates a preprocessing stage in the lexer. This enables the detection of new directives such as pragmas. These are useful when focusing on very concrete sets of transformations in order to guide the compiler to just consider a certain region of code. This technique is also employed by other program interfaces such as the OpenMP API, where the user must indicate, using pragmas, the loop or region of interest to parallelize. For our purpose, we will leverage the capabilities provided by the Clang front-end, in order to detect and

[4]This is equivalent to the Register Transfer Language (RTL) in GCC.

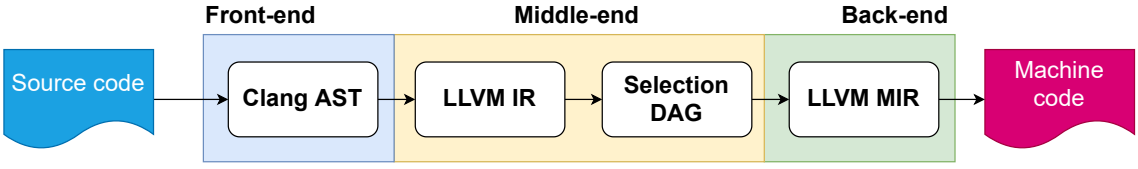


Figure 5.6: High-level LLVM toolchain for the generation of machine code.

delimit the regions of code of our interest, and get a well-formed Clang AST to analyze.

5.3.2 High-level architecture of MACVETH

MACVETH was designed and conceived to target sparse or irregular codes, e.g., SpMV kernels. These kernels are widely used machine and deep learning applications. Our approach relies on the Clang AST for parsing the input code. Instead of lowering this abstraction to the LLVM IR, our compiler rewrites the original code using, whenever it is profitable, SIMD directives in an Intrinsics style, thanks to the Clang's LibTooling library that supports rewriting the original source code. The high-level picture of the system's architecture is depicted in Figure 5.7. We logically divide our source-to-source compiler architecture into front-end, middle-end and back-end.

MACVETH handles different abstraction levels and IRs in order to facilitate the vectorization process. The input is the Clang AST, which has already been described. From there, MACVETH generates a Three-Address Code or TAC representation (SSA form), which facilitates the creation of a Directed Acyclic Graph (DAG). This structure is suitable for finding patterns in the code such as reductions. In this way, DAG's operations and operands are packed, when possible, in order to generate vector operations. These are generated in the SIMD back-end according to the heuristics and the architectural characteristics of the target platform. Then, using the Clang framework, the front-end rewrites the original source code synthesizing the SIMD code generated in the back-end. The rest of subsections delve deeper on each of the components of the architecture presented here.

5.3.3 Front-end: the driver for parsing and rewriting

The front-end in MACVETH is responsible for parsing the CLI input options (listed in Appendix C.1), initializing all the Clang LibTooling structures required to generate the AST, and interpreting the Clang AST and the input pragmas within the code (see Appendix C.2 for more details of the syntax). In addition, as the front-end is the part of the compiler that works with the Clang library, it is also responsible for

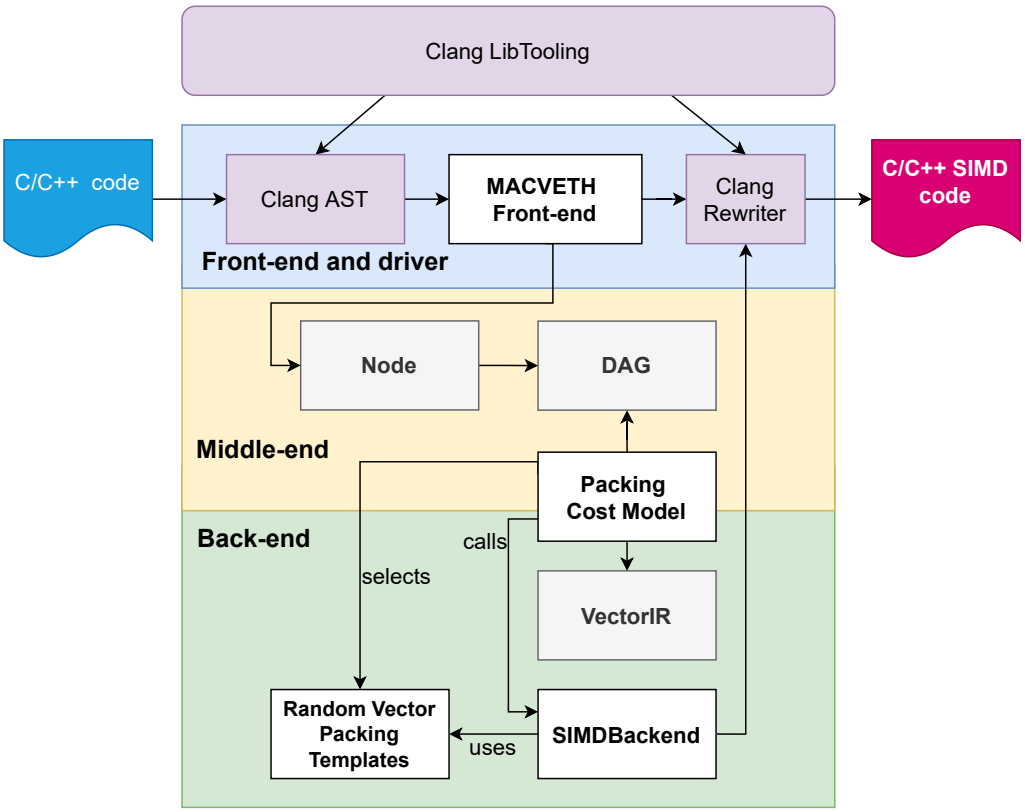


Figure 5.7: High-level diagram of MACVETH's architecture showing the different IRs used by the system.

rewriting the new SIMD code from the original source input. Figure 5.8 illustrates the most relevant components and the toolchain of MACVETH's front-end.

Clang implements complex expressions in order to handle any form or type of code. These expressions provide many possibilities when it comes to parsing the code. Nonetheless, MACVETH simplifies the complexity of Clang's expressions by wrapping all categories into a small set of expressions. `MVExpr` is an abstract class that can be specialized for any type we want to represent from the Clang AST, or to generate other abstractions. Besides, the idea of this class is to provide a set of non-standard transformations for the expressions, e.g., loop unrolling. Thus, `MVExpr` objects are instantiated using a factory.

We have implemented the following specializations, which are enough in order to represent any value referenced in the regions of interest:

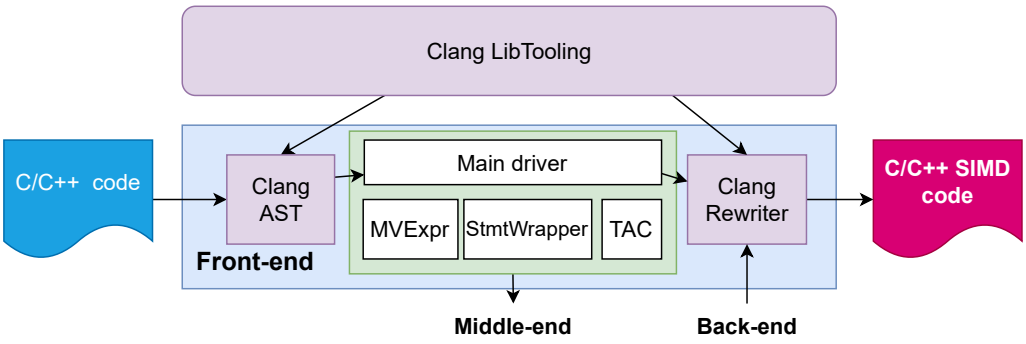


Figure 5.8: MACVETH's front-end components.

- `MVExprArray`: represents any n-dimensional expression in the code. It holds information about the number of dimensions and name or value of the indices. This class is very useful when performing unrolling, as it provides methods for computing deltas between two indices of the same or different form; e.g., for two positions in the array A with affine indices $(i+1)*4$ and $(i+1)*2$, $\forall i \geq 0$ the difference $(i+1)*4-(i+1)*2>1$, so the positions referred with these indices are not contiguous in memory.

- `MVExprVar`: regardless of the type of the variable, this abstraction basically represents any `DeclRefExpr` from the Clang AST. `DeclRefExpr` expresses a reference to a declared variable, function, enum, etc., encoding all the information about how a declaration is referenced within an expression. In our case, we oversimplify this complexity.

- `MVExprFunc`: this is a recursive abstraction that holds the name of the function and the parameters it receives. Parameters are also `MVExpr` objects.

- `MVExprLiteral`: any number (integer, float, double), char, etc. value that does not fit any of the other abstractions.

These `MVExpr` expressions belong to statements in the code. In MACVETH, these statements are wrapped into `StmtWrapper` objects, including loops. This facilitates the unrolling of loops and finding their location in the code, as we will detail next. Statements may differ in number of operations, data handled, or even types. This makes it difficult to handle them properly. The Three-Address Code (TAC) representation is used to translate any statement (S) into an SSA-based IR.

There are different ways of representing this format, but in our case we use 4 tuples, as described in Definition 1.

Definition 1. *A TAC is a 4 tuple T=(a,b,c,⊕) that represents the assignment* $a = b \oplus c$. *If the* $\oplus$ *operator is unary, then c is null, so* $a = \oplus b$.

Any statement of a program is composed of a concatenation of operations, e.g., assignments (=) and binary operations (+, a function f, etc.), which are split into TACs respecting their operational order. Thus, when any statement generates more than one TAC, temporary registers are generated in SSA form. These assignments are responsible for connecting TACs, and therefore represent the logical order of the original statement. In essence, connections between TACs generate a tree structure. In order to perform this translation, we have implemented a recursive process which is listed in Algorithm 5.3. In the algorithm, we simplify the statements to be in the form `LHS = RHS` (left- and right-hand side, respectively). For the minimum case of an induction, i.e., the assignment $p = q$, the process is the same as for those statements with unary operators, and the result would be $T = (p, q, null, =)$.

Corollary 1. *Any statement S whose TAC representation produces more than one tuple can be represented as a set of interconnected TACs:* $S = \{T_i\}/a^{T_n}, b^{T_n}, c^{T_n} \in T_n/\forall T_i \in S \Rightarrow \exists T_j/a^{T_i} = (b^{T_j}|c^{T_j}) \vee a^{T_j} = (b^{T_i}|c^{T_i})$.

The TAC representation is widely used in compilers. Its main advantage resides in the simplicity of handling operations with the same number of operands. Besides, this format is very easy to handle in programmatic terms.

Superword-Level Parallelism (SLP) enables vectorization within a basic block of code of single independent isomorphic statements [71], i.e., those statements that contain the same operations. On the other hand, loops can be unrolled and decomposed into independent single statements enabling SLP algorithms to vectorize more code [107]. In that way, MACVETH also exploits this type of vectorization across those basic blocks and loops. The loop unrolling approach in MACVETH is also performed using the TAC format, following the iterative process shown in Algorithm 5.4, and according to the parameters specified in the pragmas. MACVETH uses pragmas to indicate the region of interest to consider, and also to configure the options for unrolling the loop or loop nests (if needed): full unrolling, unrolling

Algorithm 5.3: *translateStmtToTAC* recursive function to translate statements into TAC format.

```
   Input: Statement S
   Result: List of TAC T
 1 T = [];
 2 Res = getResultOrTempReg(S);
 3 Lhs = getLHS(S);
 4 Rhs = getRHS(S);
 5 if isNonTerminal(Lhs) then
 6 |   TACList = translateStmtToTAC(Lhs);
 7 |   T.append(TACList);
 8 |   Lhs = TACList.back().Res;
 9 end
10 if isNonTerminal(Rhs) then
11 |   TACList = translateStmtToTAC(Rhs);
12 |   T.append(TACList);
13 |   Rhs = TACList.back().Res;
14 end
15 T.append(new TAC(Res, Lhs, Rhs, getOp(S)));
16 return T
```

according to a numerical factor, unroll-and-jam (for loop nests), etc. These options are described in detail in Appendix C.1. This mechanism permits to generate a new list of unrolled TACs keeping their identity, i.e., the execution order in the program. MACVETH's approach is to create new TAC objects when unrolling. Even though this is costly in terms of memory, it enables the ability to handle them individually. By doing this, another type of packing can be performed when grouping operations and operands, apart from packing together unrolled expressions.

As an example, consider the statement `D = (A + B) * C`. Its equivalent TAC representation in MACVETH is shown in Listing 5.6. As we can see, this statement is decoded into three TACs, which are connected by temporary values. Similarly, a set of concatenated reductions such as the one in Listing 5.7a is translated into the TAC representation in Listing 5.7b.

The front-end in MACVETH also acts like a driver of the compiler and is responsible for rewriting the code generated by the back-end (see Section 5.3.5). For this purpose, the IRs described before also keep track of the `SourceLocation`, a class in

Algorithm 5.4: Unrolling for a list of TACs.

Input: List of TAC T, int $UnrollingFactor$, $LoopNests$
Result: List of TAC T'

```
1  T' = [ ];
2  Step = 0;
3  foreach LoopNests do
4      for Step++ < UnrollingFactor do
5          foreach TAC in T do
6              NewTAC = {};
7              foreach Expr in TAC do
8                  NewExpr = unrollExpr(Step, LoopNests, Expr);
9                  NewTAC = placeExprInTAC(NewExpr);
10             end
11             T'.append(NewTAC);
12         end
13     end
14 end
15 return T';
```

```
1 t0 = A + B
2 t1 = t0 * C
3 D = t1;
```

Listing 5.6: TAC translation for `D = (A + B) * C`.

the Clang AST that holds a pointer to the original source location in the input file of an expression or statement. In addition, LibTooling provides a `Rewriter` class, which is able to insert, replace or remove lines in the original code.

5.3.4 Middle-end: identifying and grouping reductions

When it comes to scheduling the different TACs in the region of interest of our program, we need a representation that can handle the dependencies between the statements and some structures that store the information about the placement of these statements in the execution. In our approach, we represent a forest of TACs as a set of interconnected nodes, where each node can be a memory operation (load,

```
1 R += A;
2 R += B;
3 R += C;
4 R += D;
```

(a) Example of 4 reductions.

```
1 t0 = R + A;
2 R = t0;
3 t1 = R + B;
4 R = t1;
5 t2 = R + C;
6 R = t2;
7 t3 = R + D;
8 R = t3;
```

(b) TAC translation.

Listing 5.7: TAC translation for 4 reductions.

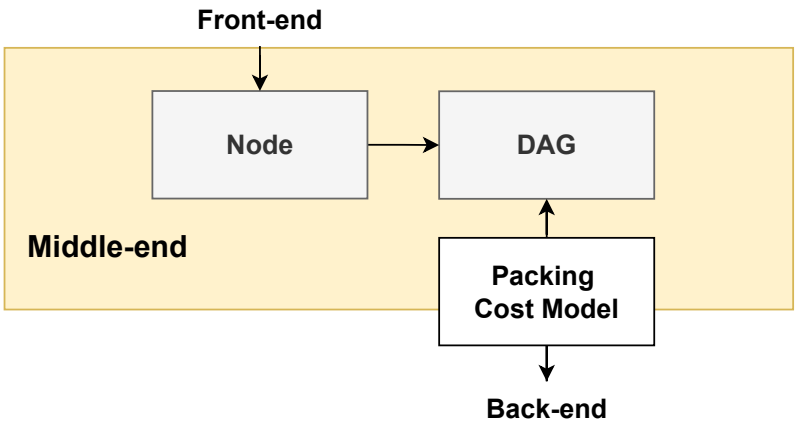


Figure 5.9: MACVETH's middle-end components.

store) or any other type of operation (addition, multiplication, built-in function, etc.). For this purpose we use a DAG [124], i.e., a graph $G = (N, E)$, where $N = \{n_0, \ldots, n_{n-1}\}$ represents the set of nodes, and $E \subseteq \{(e_0, e_1) | (e_0, e_1) \in N \times N : e_0 \neq e_1\}$ is the set of ordered edges representing the data dependencies between the nodes. In addition, this graph has no cycles, i.e., if a walk $w = (e_0, \ldots, e_{n-2})$ in G is a finite sequence of edges $e_i \in E$ joining a sequence of nodes $(n_0, \ldots, n_{n-1}) \subseteq N$, then $\nexists\, w : e_0 = e_{n-2}$.

Figure 5.9 illustrates the main components interacting in the middle-end. The DAG is built using structures implemented as `Node` objects. For each TAC a set of three interconnected nodes is created, where there are two inputs and an operation/output node. Input nodes can be load nodes, from memory or literals, or they

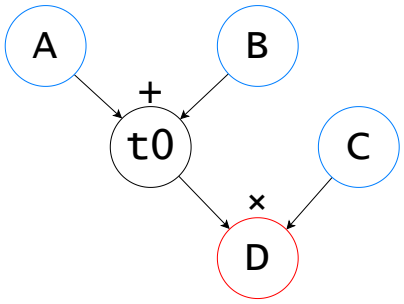


Figure 5.10: DAG generated from the example in Listing 5.6.

could come from the result of other TAC, i.e., from another node holding a temporary value. Operation/output nodes can be terminal or temporary. If the node is terminal, it performs an operation and an assignment, otherwise just an operation whose output will be used by another operation node. These `Node` objects also shelter other information related with the order in the original code and the dependencies with other operations and operands. This information is important when choosing the nodes that can be packed together, or when identifying reductions. As such, we compute the Free Scheduling (FS) value, which represents the depth of a node in the DAG. In this way, initially we can select and pack the operation nodes performing the same operation (isomorphic statements) and with the same FS value, as there can be no dependencies between them, but the reduction case is different, as there are data dependencies.

As described before, in MACVETH the DAG will help to understand and identify the patterns present in the code in order to pack operations and operands that can be executed together in a vectorized fashion, or using a known set of operations (e.g., reductions). From the previous examples, taking the TAC representation in Listing 5.6, MACVETH builds the DAG in Figure 5.10. The input nodes are depicted in blue, while the temporary ones are colored in black. As we explained earlier, there is a simplification in the DAG for the assignment operator, as it is depicted in the terminal D node (in red). The DAG does not generate an additional node for the assignment, but the output node D holds that information within the `Node` structure.

The reduction example in Listing 5.7b is synthesized as the DAG in Figure 5.11. Multiple reductions on an element in the code generate a very recognizable shape

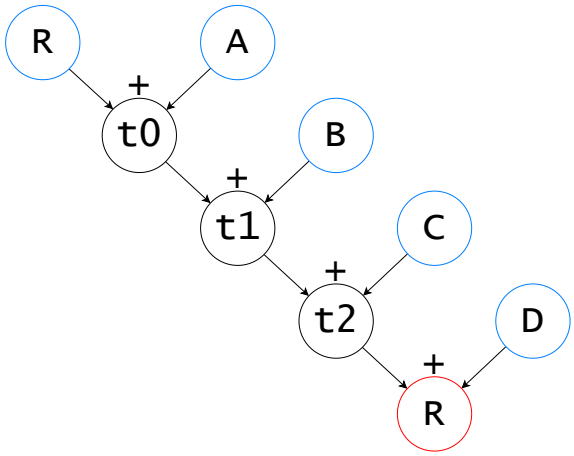


Figure 5.11: DAG generated from the reduction TACs in Listing 5.7b.

in the DAG: starting from the terminal output node (red node R at the bottom), the left input operands are always the temporary values (operation nodes) which are assigned to the original reduction R in the TACs, having a decreasing FS value, i.e., $FS_R > FS_{t2} > FS_{t1} > FS_{t0}$. Following these features, we have implemented an algorithm in MACVETH to identify the independent reductions on any arbitrary element or memory position. Continuing the reduction example, an SpMV code could be as simple as in Listing 5.8a, generating the TAC representation in Listing 5.8b. In this example, we have two independent reductions on two different elements (`y[0]` and `y[1]`). The resulting DAG is depicted in Figure 5.12. As it can be observed, there are two independent reduction trees without any type of connection between them. These patterns are identified as two independent reductions in the code. On the other hand, nodes `t0`, `t2`, `t4` and `t6` in the figure can be packed together in the same operation as they have the same FS value. MACVETH deals with these situations in a best-effort manner, by trying to group and vectorize first the reductions, as we will describe next, and using a greedy algorithm for packing and consuming isomorphic nodes, and with the same FS value.

This last example is the typical scenario for any SpMV code. The size of the target matrices can be very different, featuring from a few nonzero values to tens of millions. For these reasons, it is important to MACVETH to properly identify the independent reductions in the code.

```
1 y[0] += A[0] * x[2];
2 y[0] += A[1] * x[42];
3 y[1] += A[2] * x[11];
4 y[1] += A[3] * x[22];
```

(a) Example of SpMV code.

```
1  t0 = A[0] * x[2];
2  t1 = y[0] + t0;
3  y[0] = y[0] = t1;
4  t2 = A[1] * x[42];
5  t3 = y[0] + t2;
6  y[0] = y[0] = t3;
7  t4 = A[2] * x[11];
8  t5 = y[1] + t4;
9  y[1] = y[1] = t5;
10 t6 = A[3] * x[22];
11 t7 = y[1] + t6;
12 y[1] = y[1] = t7;
```

(b) TAC translation.

Listing 5.8: TAC translation for the SpMV code.

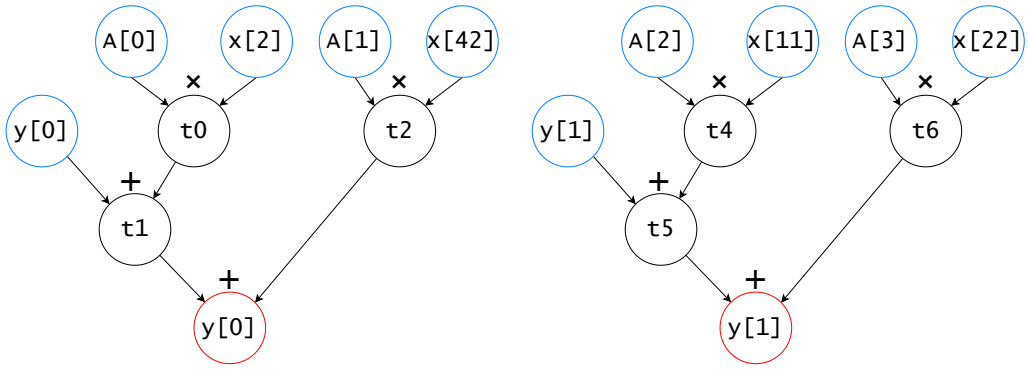


Figure 5.12: DAG generated from the reduction TACs in Listing 5.8b.

Packing cost model: grouping reductions

After identifying all reductions in the code, MACVETH applies a best-effort methodology for packing and consuming those reduction nodes in the code. The compiler implements a bottom-up algorithm detailed in Algorithm 5.5. In order to avoid the inspection of all nodes at the same time in large codes, the algorithm works within a set or window of nodes in the DAG of size *WindowSize*. The main goal of this algorithm is to pack together the largest possible number of reductions in order to maximize vector occupancy. For that purpose, we first create a map (ReductionsMap, line 7 in the algorithm) containing all the reduction nodes in the

Algorithm 5.5: Approach followed by MACVETH for packing reductions.

```
   Input: DAG G, int WindowSize, int MaxPackingSize, int MinPackingSize
   Result: List of SIMD instructions
 1 unvisited_nodes = [ ];
 2 SIMDList = [ ];
 3 i = 0;
 4 for W = G[i : i + WindowSize] do
 5 |   i = i + WindowSize;
 6 |   W = W + unvisited_nodes;
 7 |   ReductionsMap = get_map_reductions(W);
 8 |   NewPacking = [ ];
 9 |   PackingSize = MaxPackingSize;
10 |   while PackingSize > MinPackingSize do
11 |   |   for Map in ReductionsMap do
12 |   |   |   if Map.size() < PackingSize then
13 |   |   |   |   continue;
14 |   |   |   end
15 |   |   |   NewPacking += Map[0:PackingSize];
16 |   |   |   Map.delete(0:PackingSize);
17 |   |   |   if (NewPacking.size() == MaxPackingSize) then
18 |   |   |   |   break;
19 |   |   |   end
20 |   |   end
21 |   |   if NewPacking.size() == 0 then
22 |   |   |   PackingSize /= 2;
23 |   |   |   continue;
24 |   |   end
25 |   |   SIMDList.append(synthesize(NewPacking));
26 |   |   NewPacking = [ ];
27 |   end
28 |   SIMDList.append(synthesize(NewPacking));
29 |   unvisited_nodes.append(get_unvisited_nodes(ReductionsMap));
30 end
31 unvisited_nodes = vectorize_orphan_redux(unvisited_nodes, SIMDList);
   // Issues scalar code with the rest of orphan nodes
32 for Node in unvisited_nodes do
33 |   SIMDList.append(synthesize(Node));
34 end
35 return SIMDList;
```

window grouped by element. The next step is to consume the maximum number of reductions on the same element according to the maximum size vector (*MaxPackingSize*). For simplicity, in the following examples we are assuming floats (32 bits) and a vector width of 256 bits. As such, *MaxPackingSize* is 8. The algorithm first attempts to pack together at least 8 nodes corresponding to reductions on the same element (line 15 in the algorithm). If we have, for instance, 9 reduction statements on the same element, 8 will be packed together in this first pass, and the leftover or orphan node will be considered to be packed with other orphan nodes (line 31); and if not feasible it will be issued sequentially, in scalar form (lines 32–34). But the algorithm also tries to group together independent reductions, if possible. If there are 4 independent reduction statements on two different elements, they will be packed together as well (4+4). The minimum number of elements to pack is also specified by *MinPackingSize*. These parameters constitute the parametric restrictions for the algorithm, which will vary depending on the architecture, the ISA and the data type. These values can also be modified by the user in the CLI (see option `--min-redux-size` in Appendix C.1). For modern architectures, such as Intel Cascade Lake, with AVX2 using floats, we have empirically experienced no benefit in packing less than 4 reductions on the same element (*MinPackingSize*), since the number of instructions and data movements (load and swizzle instructions) introduced compared to the equivalent scalar code has a detrimental impact.

There will also be cases where some or all reductions on an element cannot be packed together since the number of candidate nodes is less than the *MinPackingSize* value. In this case, another search is done in a best-effort manner. The cost model will try to group those isomorphic nodes whose store values are contiguous memory addresses. The process is detailed in Algorithm 5.6. The idea is very simple: find those store values of the list of orphan reduction statements whose memory addresses are contiguous in ascending order. If we find strides of 4 or more, then the algorithm packs together these operations in a vectorized fashion, avoiding issuing those statements as scalars. This option can also be disabled in the CLI (see option `--novec-orphan-redux` in Appendix C.1). To better illustrate this use case consider the code in Listing 5.9, which represents a small window in a larger code. There are 5 reduction statements on `y[0]`, only one on `y[1]` and `y[2]`, and 2 on `y[3]`. The process described in Algorithm 5.5 will pack the first 4 reduction statements on `y[0]`, discarding the consideration of the last one. This will be appended to the

Algorithm 5.6: *vectorize_orphan_redux* function for vectorizing orphan reductions.

```
   Input: List of Nodes N, int MinPackingSize
   Result: List of SIMD instructions
 1 PrevMem = 0;
 2 AlreadyMapped = [ ];
 3 SIMDList = [ ];
 4 for MainNode in N do
 5 |   Nodes = [ ];
 6 |   for NewNode in N do
 7 |   |   NewMem = get_mem(NewNode);
 8 |   |   if PrevMem + 1 == NewMem then
 9 |   |   |   Nodes.append(NewNode);
10 |   |   end
11 |   end
12 |   if Nodes.size() >= MinPackingSize then
13 |   |   SIMDList.synthesize(Nodes);
14 |   end
15 end
16 return SIMDList;
```

unvisited_nodes list, as the algorithm only packs a number of reductions multiple of 2. `y[1]` and `y[2]` will be appended to this list as well, and since there are only 2 reduction statements on `y[3]`, the algorithm will not consider their packing, and they will also be appended to the *unvisited_nodes* list. Following this reasoning, the content of this list, after consuming all reduction nodes in DAG G (line 31 in the algorithm), will be the reduction statements corresponding to {`y[0]`, `y[1]`, `y[2]`, `y[3]`, `y[3]`} (lines 5–9 in Listing 5.9). With these nodes, the Algorithm 5.6 will pack together the first four reduction statements ({`y[0]`, `y[1]`, `y[2]`, `y[3]`}), since their store values are contiguous in memory and their operations are isomorphic, which makes these nodes suitable for vectorization. The last reduction on `y[3]` would be issued as scalar (see lines 32–34 in Algorithm 5.5), if it cannot be packed with any other nodes in the following windows.

```
1  y[0] += A[0];
2  y[0] += A[1];
3  y[0] += A[2];
4  y[0] += A[3];
5  y[0] += A[4];
6  y[1] += A[5];
7  y[2] += A[6];
8  y[3] += A[7];
9  y[3] += A[8];
10 ...
```

Listing 5.9: Example code where the grouping of orphan reduction nodes applies.

5.3.5 Back-end: fusing reductions and synthesis of SIMD code

In order to tackle all different architectures when generating instructions, we need a generic vector representation of the vector instructions we want to pack in our program. For this purpose, we developed the abstraction layer `VectorIR` in MACVETH, which basically packs together a set of nodes from the DAG into a common structure which represents a vector operation. This vector representation is linked to the target architecture and ISA, as the maximum number of elements in a vector operation depends on the data type and the maximum vector width in the architecture. Thus, each vector operation will also depend on two vector operands which represent a set of nodes in the DAG. The formal representation of this IR is described in Definition 2.

Definition 2. *A vector operation is a 4-tuple* $VIR = (VOp_R, VOp_1, VOp_2, \oplus)$*, being* $C \subseteq DAG$ *such that: 1)* $VOp_R = VOp_1 \oplus VOp_2$*, 2)* $VOp_i \subseteq C$*, 3)* $|VOp_i| \leq ISA_{width}$*, 4)* $VOp_1 \neq VOp_2$*.*

`VectorIR` is a generic way of representing vector operations for the different architectures. Because of this, at this stage there is no fusing operations or any other kind of target specific optimizations. The concrete back-end will be in charge of doing this; for instance, an architecture featuring AVX could not have Fused Multiply-Add (FMA) instructions.

The `SIMDBackend` in MACVETH is an abstraction that has to be implemented

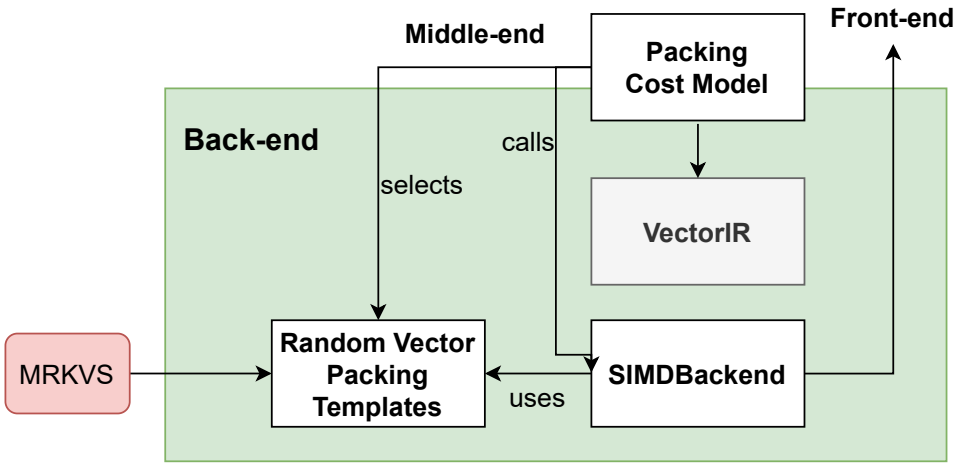


Figure 5.13: MACVETH's back-end components.

for each concrete ISA, and is in charge of generating code from the operations in the `VectorIR`. This toolchain is depicted in Figure 5.13. Therefore, this back-end first performs the instruction selection and then the register allocation. For the reduction case, the back-end has already embedded a set of formulas depending on the type and number of elements to reduce in a vector fashion. These formulas depend on the architecture and ISA. Assuming floats, for instance, for 4 reductions on an element in Listing 5.10a, MACVETH synthesizes the SSE code in Listing 5.10b. Continuing the example, for 8 reductions in Listing 5.11a, MACVETH synthesizes the AVX2 code in Listing 5.11b.

These are ad hoc implementations based on the best empirically found solutions, trying to minimize the pipeline latency and the number of instructions generated. The formulas implemented for performing reductions in MACVETH are intended for SSE and up to AVX2. These are different depending on the data type, e.g., double and float, since the number of elements in the register is different. In the same way, for each new ISA to be supported in the system the same exploration should be done. Newer x86 SIMD extensions such as AVX-512 introduce a new Intrinsics such as `_mm512_reduce_add_ps`, which has the same functionality as described above but for 512-bit registers, and is implemented by compilers as a sequence of other built-in Intrinsics and instructions[5].

[5]GCC implementation for this Intrinsics: https://github.com/gcc-mirror/gcc/blob/9d7e19255c06e05ad791e9bf5aefc4783a12c4f9/gcc/config/i386/avx512fintrin.h#L15928.

```
1 tmp += y[0];
2 tmp += y[1];
3 tmp += y[2];
4 tmp += y[3];
```

(a) Reduction on `tmp`.

```
1 __vop0 = _mm_loadu_ps(&y[0]);
2 __mv_hi128 = _mm_movehl_ps(__vop0, __vop0);
3 __vop0 = _mm_add_ps(__vop0, __mv_hi128);
4 __mv_hi128 = _mm_shuffle_ps(__vop0, __vop0, 0b00000001);
5 __mv_lo128 = _mm_add_ps(__vop0, __mv_hi128);
6 tmp = tmp + __mv_lo128[0];
```

(b) MACVETH output.

Listing 5.10: Example of synthesis in MACVETH for 4 reductions on a float in SSE (AVX2-compliant).

```
1 tmp += y[0];
2 tmp += y[1];
3 tmp += y[2];
4 tmp += y[3];
5 tmp += y[4];
6 tmp += y[5];
7 tmp += y[6];
8 tmp += y[7];
```

(a) Reduction on `tmp`.

```
1  __vop0 = _mm256_loadu_ps(&y[0]);
2  __mv_lo128 = _mm256_castps256_ps128(__vop0);
3  __mv_hi128 = _mm256_extractf128_ps(__vop0, 0x1);
4  __mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
5  __mv_hi128 = _mm_movehl_ps(__mv_lo128, __mv_lo128);
6  __mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
7  __mv_hi128 = _mm_shuffle_ps(__mv_lo128, __mv_lo128,
8                              0b00000001);
9  __mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
10 tmp = tmp + __mv_lo128[0];
```

(b) MACVETH output.

Listing 5.11: Example of synthesis in MACVETH for 8 reductions on a float in AVX2.

Random vector packing templates

When the operands of an operation are not contiguous in memory, the packing cost model selects the random vector packing template to use based on the already pre-computed cost model described in Section 5.2. This model gets the reference for the concrete template based on the architecture and the ISA, and the contiguity of the elements to pack together. These template files are indexed by the grammar specified in Section 5.2.4, and they correspond to the candidates generated by the MRKVS system. With these templates, MACVETH fills their input values with the corresponding memory addresses and registers from the code.

Fusing reductions

In the middle-end, the packing cost model tries to maximize the vector occupancy for reductions. MACVETH considers two forms of fusing independent reductions: using the same vector register (intra-register), and using multiple vector registers (inter-register). For the first case, the back-end just performs a partial reduction on the register to be reduced, as shown in Listing 5.12. In this case, the compiler uses the same operations to simultaneously compute both independent reductions (`tmp0` and `tmp1`). This idea is also graphically depicted in Figure 5.14. This example is for vectors of 256 bits, but it would be the same for vectors of 128 bits, only changing the instructions used and the number of steps. This approach has a limitation: the number of values in each independent reduction must be the same, and the values must be placed contiguously. This is why the packing cost model must pack, typically, a multiple of 2 reductions together. Following the example, packing 5 reductions on `tmp0` and 3 on `tmp1` cannot be done with the approach proposed here. However, MACVETH does not even consider this kind of patterns in the packing cost model, since we have tested them empirically observing no benefit to their vectorization.

The second case, fusing reductions in different vector registers, is shown in Listing 5.13 and graphically depicted in Figure 5.15. In this case, we leverage the horizontal addition instruction in the ISA (`vhaddps`, in Intrinsics `_mm256_hadd_ps()`) for fusing the shuffling and addition at the same time of two vectors using a single instruction (decoded into more micro-operations in the pipeline). This instruction horizontally adds adjacent pairs of elements in both input operands, leaving the most significant 128 bits and the least significant 128 bits of the output register perfectly aligned to be added vector wise. The rest of the operation has already been covered in the previous example, when reducing multiple values within the same vector. In this example, for simplicity, we just illustrate the joint reduction of two vectors, but this approach is extensible and was implemented in MACVETH for up to four independent reductions in different vector registers (256 bits). The main goal of both types of fusions is to maximize vector occupancy since reductions, by nature, are not 'strictly' vectorizable.

```
1 tmp0 += y[0];
2 tmp0 += y[1];
3 tmp0 += y[2];
4 tmp0 += y[3];
5 tmp1 += y[4];
6 tmp1 += y[5];
7 tmp1 += y[6];
8 tmp1 += y[7];
```

(a) Two independent reductions on `tmp0` and `tmp1`.

```
1 __vop0 = _mm256_loadu_ps(&y[0]);
2 __tmp0_256 = _mm256_permute_ps(__vop0, 0b00001110);
3 __tmp1_256 = _mm256_add_ps(__vop0, __tmp0_256);
4 __tmp2_256 = _mm256_shuffle_ps(__tmp1_256, __tmp1_256,
5                                 0b00000001);
6 __mv_lo256 = _mm256_add_ps(__tmp1_256, __tmp2_256);
7 tmp0 = tmp0 + __mv_lo256[0];
8 tmp1 = tmp1 + __mv_lo256[4];
```

(b) MACVETH output for the fusion of these reductions.

Listing 5.12: Example of synthesis in MACVETH for the fusion of two independent reductions of 4 elements each within the same vector.

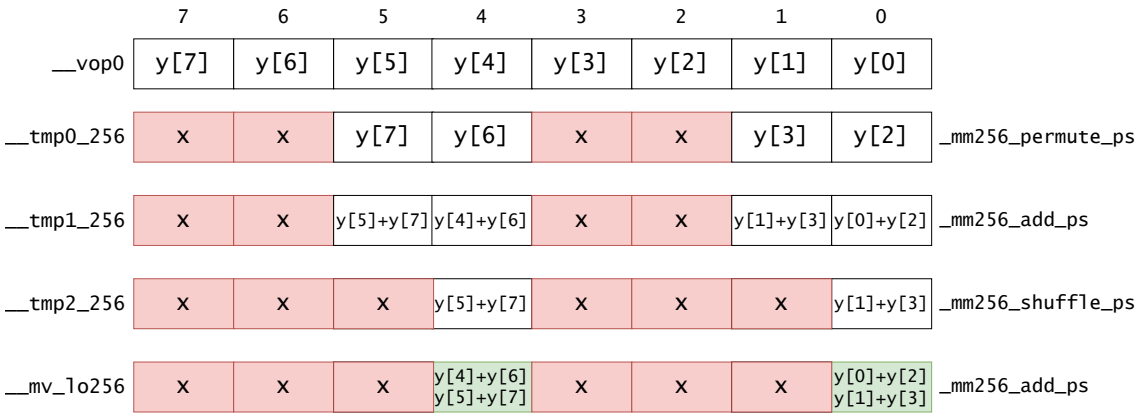


Figure 5.14: Graphic description of vectors' content for the code in Listing 5.12b. Values marked with a cross (x) are not displayed as they do not contribute to the final reduction. The position of the result of each reduction is in green.

```
1  tmp0 += y[0];
2  tmp0 += y[1];
3  tmp0 += y[2];
4  tmp0 += y[3];
5  tmp0 += y[4];
6  tmp0 += y[5];
7  tmp0 += y[6];
8  tmp0 += y[7];
9  tmp1 += z[0];
10 tmp1 += z[1];
11 tmp1 += z[2];
12 tmp1 += z[3];
13 tmp1 += z[4];
14 tmp1 += z[5];
15 tmp1 += z[6];
16 tmp1 += z[7];
```

(a) Two independent reductions on `tmp0` and `tmp1`.

```
1  __vop0 = _mm256_loadu_ps(&y[0]);
2  __vop2 = _mm256_loadu_ps(&z[0]);
3  __vop0 = _mm256_hadd_ps(__vop0, __vop2);
4  __mv_lo128 = _mm256_castps256_ps128(__vop0);
5  __mv_hi128 = _mm256_extractf128_ps(__vop0, 0x1);
6  __mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
7  __mv_hi128 = _mm_shuffle_ps(__mv_lo128, __mv_lo128,
8                              0b00110001);
9  __mv_lo128 = _mm_add_ps(__mv_lo128, __mv_hi128);
10 tmp0 = tmp0 + __mv_lo128[0];
11 tmp1 = tmp1 + __mv_lo128[2];
```

(b) MACVETH output for the fusion of these reductions.

Listing 5.13: Example of synthesis in MACVETH for the fusion of two independent reductions of 8 elements each in two different vectors.

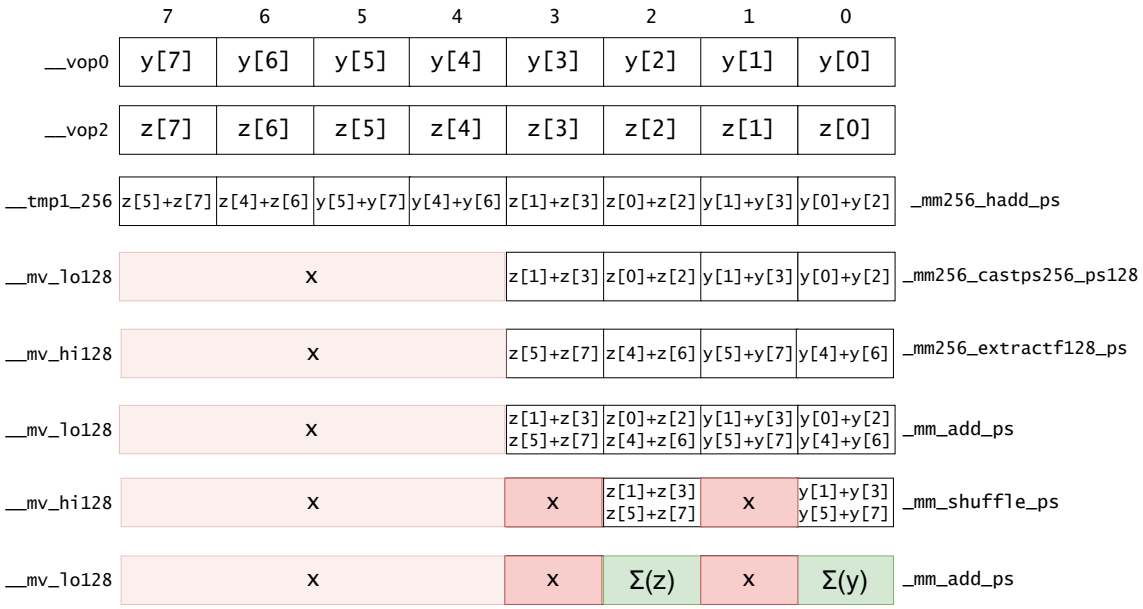


Figure 5.15: Graphic description of vectors' content for the code in Listing 5.13b. Values marked with a cross (`x`) are not displayed as they do not contribute to the final reduction. The position of the result of each reduction is in green.

Peephole optimizations: FMAs and dead code elimination

MACVETH also performs two types of peephole optimizations in the code: the substitution of additions and multiplications by Fused Multiply-Add (FMA) instructions, and a simple dead code elimination (DCE) approach. FMAs are of the form shown in Listing 5.6: `D = (A + B) * C`. We can fuse these two operations into a single FMA if and only if the result of `A + B` is not consumed by any other statement in the code. Not all architectures support this feature, and this is the reason why this optimization is only considered in the back-end. MACVETH also inspects the liveness of the variables it generates in order to remove them if they are not used after the end of the region of interest.

5.3.6 Current limitations of the tool

MACVETH is meant to be built using LLVM library from version 11.x. It is written using C++17 features, and it is recommended to be compiled using GNU GCC from version 9.3.0. Currently it only fully supports Linux and UNIX-like systems. Here is the list of restrictions and assumptions that the user must consider when using MACVETH:

- Only C/C++ code is allowed, and it must be delimited by pragmas (see Appendix C.2).
- There is only support for x86 architectures featuring AVX2 SIMD extensions (including FMA) or prior. There is no support for AVX-512 extensions yet.
- Pointers and indirections are not allowed in the code to be vectorized.
- Variable declarations are not allowed within the body of a loop or a region. Only variable declarations within a `for` statement are allowed, e.g., `for (int i = ...)`.
- No conditional statements are allowed within the regions to be considered for vectorization.

- All statements contained in the region of interest must adhere to the following grammar in extended Backus-Naur Form:

$$
\begin{aligned}
\langle \text{syntax} \rangle &\models \langle \text{stmt} \rangle \mid \langle \text{stmt} \rangle \langle \text{syntax} \rangle \\
\langle \text{stmt} \rangle &\models \langle \text{expr} \rangle \langle \text{assign} \rangle \langle \text{op} \rangle ; \mid \langle \text{for\_stmt} \rangle \mid \lambda \\
\langle \text{for\_stmt} \rangle &\models for(\texttt{var} = \langle \text{expr} \rangle ; \texttt{var}\ \langle \text{cond} \rangle\ \langle \text{expr} \rangle ; \langle \text{incr\_op} \rangle) \{ \langle \text{syntax} \rangle \} \\
\langle \text{op} \rangle &\models \langle \text{op} \rangle\ \langle \text{bin\_op} \rangle\ \langle \text{op} \rangle \mid f(\langle \text{op} \rangle, \langle \text{op} \rangle) \mid f(\langle \text{op} \rangle) \mid \langle \text{expr} \rangle \\
\langle \text{incr\_op} \rangle &\models \texttt{++var} \mid \texttt{var++} \mid \texttt{var} \langle \text{assign} \rangle \texttt{digit} \\
\langle \text{cond} \rangle &\models < \mid <= \mid >= \mid > \\
\langle \text{bin\_op} \rangle &\models \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \\
\langle \text{assign} \rangle &\models \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \\
\langle \text{expr} \rangle &\models \texttt{array} \mid \texttt{digit} \mid \texttt{var}
\end{aligned}
$$

5.4 Experimental Results

In this section we describe the experiments performed to assess the performance and correctness of the codes synthesized by MACVETH. This tool is targeted towards irregular codes, such as SpMV computations. Sparse matrices are typically represented using the Compressed Sparse Row (CSR) format, or other similar compressed schemes, e.g., Coordinate List (COO), Compressed Sparse Column (CSC), etc. These formats are designed to save space and computations by only storing the positions of nonzero values of the matrix, but indirection arrays need to be used to scan the nonzero elements. A common SpMV kernel using the CSR format is illustrated in Listing 5.14.

```
1 for (int i = 0; i < N ; ++i) {
2     y[i] = 0.0;
3     for(j = row_ptr[i]; j < row_ptr[i + 1]; ++j)
4         y[i] += A[j] * x[cols[j]];
5 }
```

Listing 5.14: Classic SpMV kernel using CSR format.

```
1  void kernel_spmv_fragment_0(float *__restrict A, float *__restrict x,
2                              float *__restrict y) {
3      register int i0;
4      for (i0 = 0; i0 <= 1; ++i0) {
5          y[0 * i0 + 1] += A[1 * i0 + 0] * x[1 * i0 + 0];
6      }
7      for (i0 = 0; i0 <= 2; ++i0) {
8          y[0 * i0 + 2] += A[1 * i0 + 2] * x[1 * i0 + 0];
9      }
10     for (i0 = 0; i0 <= 1; ++i0) {
11         y[0 * i0 + 3] += A[1 * i0 + 5] * x[1 * i0 + 1];
12     }
13     for (i0 = 0; i0 <= 1; ++i0) {
14         y[0 * i0 + 4] += A[1 * i0 + 7] * x[1 * i0 + 1];
15     }
16     y[5] += A[9] * x[1];
17     for (i0 = 0; i0 <= 1; ++i0) {
18         y[1 * i0 + 5] += A[1 * i0 + 10] * x[0 * i0 + 2];
19     }
20 }
21
22 void kernel_spmv(float *__restrict A, float *__restrict x,
23                  float *__restrict y) {
24     kernel_spmv_fragment_0(A, x, y);
25 }
```

Listing 5.15: Computations generated with the system developed by Augustine et al. [7] for the input matrix JGD_Kocay/Trec5 from the SuiteSparse collection [21].

In order to avoid the use of indirection arrays, Augustine et al. [7] developed a system for automatically building sets of regular subcomputations by mining regular subregions on the irregular data structure, i.e., on a sparse matrix. This approach generates specialized code for each input sparse matrix improving the opportunities to generate vector code, such as in those included in the SuiteSparse collection [21]. An example of the output of this system for the input sparse matrix JGD_Kocay/Trec5 is depicted in Listing 5.15. This type of codes features loops with very small trip count and known upper bound, which can be fully unrolled. It also presents regular subcomputations in the form of n-sized reductions that are easily vectorizable (even for auto-vectorizers) and are eligible candidates to be fused in MACVETH. On the other hand, the code also presents independent basic blocks

that can be grouped together using SLP-like techniques (as in line 16 of the listing).

We have designed a set of experiments to assess the correctness of the code synthesized by the compiler, varying from tens to thousands of sets of reductions on the same values. These experiments have been executed using MARTA (see Chapter 4). We have also developed an extension of MARTA to integrate the MACVETH compilation step, and the dumping of the output values in the resulting arrays for both code versions: the original one and the MACVETH-vectorized version. This dumping is performed using macros and functions available in PolyBench [103]. In this way, MARTA is able to check the correctness of MACVETH-generated codes by comparing their outputs with those of the original codes. All the experiments presented in this section passed this sanity check successfully. Both the compilation and execution steps (including the MACVETH compilation) were completely automated, requiring only a very small configuration file for running these tests. All the experiments were conducted on an Intel Xeon Scalable 4216 processor (Cascade Lake architecture) at a fixed frequency of 2.1 GHz. This architecture features AVX-512, but in our case we limited the vectorization opportunities for the MACVETH-generated codes to AVX2. All binaries were compiled with GNU GCC 11.2.0 using `-Ofast -march=native -mtune=native`, as the SLP-like vectorizer features improvements over previous versions by considering opportunities for vectorizing across basic blocks of code and loops [37]. Each version of the program has been executed 50 times, reporting the arithmetic mean of all measurements after removing the largest and smallest ones, and discarding the outliers with an absolute difference with the arithmetic mean greater than 3 times the standard deviation.

Next, Section 5.4.1 presents the analysis of the results obtained for the synthetic patterns described there, and Section 5.4.2 presents the analysis of a subset of the matrices built in the work by Augustine et al. [7].

5.4.1 Synthetic patterns

The synthetic patterns in the experimental set consist of different instances of the parametric loop in Listing 5.16. The `UB` variable in the loop declaration represents the upper bound, which is known at compile time, enabling the full unrolling of the loop. In this way, each synthetic code contains $\frac{\#REDS}{UB}$ loops, where $\#REDS$ is the

```
1    for (int i = 0; i < UB; ++i) {
2        y[Y_OFFSET] += A[i + A_OFFSET] * x[X_STRIDE * i + X_RANDOM_OFFSET];
3    }
```

Listing 5.16: Loop present in the synthetic codes generated.

Table 5.1: MACVETH configurations for the synthetic patterns.

Name	`--min-redux-size`	`--nofuse`
`4redux`	4	N.A.
`8redux_fuse`	8	Disabled
`8redux_nofuse`	8	Enabled

number of reductions considered for the code. The `Y_OFFSET` is incremental for each loop generated in the code, so that each loop reduces on the same element. To this extent, each loop represents `UB` reductions on `y[Y_OFFSET]`. The indices for the `x` array are random across its domain. The access pattern for matrix `A` is sequential regardless of the loop trip count. These computations resemble the SpMV codes generated in [7], but they do not represent any real matrix.

We used different configurations for MACVETH, as disclosed in Table 5.1. In this case, we consider two values for the minimum reduction size, 4 and 8, and the option of enabling or disabling the fusion of independent reductions (we refer to Sections 5.3.4 and 5.3.5 for more details). These values also determine the loop trip count, i.e., for a minimum reduction size of 4 elements, the synthetic patterns to compile all have 4 iterations. The same reasoning applies for a minimum reduction size of 8 elements. Note that all these patterns are regular since all loops have the same upper bound, and the packing algorithm produces zero orphans since the operations involved in each reduction can be packed together. As such, the consideration for vectorizing orphan reductions does not apply (see Section 5.3.4). The `4redux` configuration does not consider the inter-register fusion of independent reductions, so the setup of this option does not affect the output of the synthesized code. With this configuration we want to measure the impact of the intra-register fusion of reductions. In the same way, the `8redux_fuse` configuration measures the impact of the inter-register fusion of reductions. The `8redux_nofuse` configuration does not apply inter-register fusion, and because of the nature of the input loops for

Table 5.2: Comparison of MACVETH configurations for the synthetic patterns in terms of speedup in cycles. Blue values indicate the maximum speedup for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
`4redux_fuse`	1.062	0.187	0.885	0.916	1.005	1.190	1.441
`8redux_fuse`	1.154	0.124	0.970	1.060	1.197	1.219	1.326
`8redux_nofuse`	0.937	0.319	0.228	0.803	0.988	1.114	1.335

Table 5.3: Comparison of MACVETH configurations for the synthetic patterns in terms of speedup in the reduction of the number of micro-operations retired. Blue values indicate the maximum speedup for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
`4redux_fuse`	1.108	0.072	1.011	1.043	1.127	1.141	1.207
`8redux_fuse`	1.174	0.122	0.996	1.089	1.186	1.284	1.331
`8redux_nofuse`	0.990	0.017	0.967	0.975	0.996	1.001	1.016

this concrete configuration, it neither applies intra-register fusion.

We generated a sequence of codes varying from 8 to 4096 reductions, doubling the number of reductions in each step. Tables 5.2 and 5.3 show descriptive statistics for the speedups measured in terms of cycles and reduction of micro-operations, respectively. The best mena and statistical dispersion (standard deviation and quartiles) in both cycles and micro-operations retired is obtained for the `8redux_fuse` configuration (even though the best standard deviation in micro-operations is obtained for `8redux_nofuse`). On the other hand, the maximum speedup in cycles is obtained for the `4redux` configuration with a 44.1% speedup (value 1.441 in Table 5.2). Regarding reduction of micro-operations, the `8redux_fuse` configuration obtains a maximum speedup of 33.1%. Both maximum speedups are obtained for the pattern with 4096 reductions. This effect is more remarkable for the reductions of 8 elements, where the speedup in cycles is on average 15.4% for the best configuration. In terms of micro-operations, there is a significant average speedup for all the configurations but, again, `8redux_fuse` configuration achieves the best overall results, with a promising average speedup of 17.4%. Both the speedup in cycles and in the reduction of the number of micro-operations retired greatly contribute to the energy efficiency of the application.

Figure 5.16 shows a barplot of the speedups obtained all MACVETH configurations described in terms of cycles and reduction of micro-operations. It is remarkable that in almost all cases there is a significant speedup for `4redux` and `8redux_fuse`, except when the number of reductions to pack is very small (8 and 16 reductions). These are cases where the compiler is able to vectorize these loops in a similar manner as MACVETH. The potential benefits of our approach are visible when increasing the number of independent reductions. Here the fusion of reductions plays an important role, by decreasing the number of instructions retired. For instance, this decrease and the number of reductions in the code have a Pearson correlation coefficient of R=0.70 for the `8redux_fuse` configuration (inter-register fusion enabled).

Table 5.4 shows the descriptive metrics for all MACVETH configurations regarding the increment of vector FLOPs compared with the GCC auto-vectorized version. It is remarkable that GCC reports successful vectorization of all loops for all these codes, adding more value to our results. The metric shown in Table 5.4 is computed as described in Equation 5.7, where $SCALAR$ corresponds to scalar FP instructions retired, 128_PACKED to 128-bit FP instructions (4 floats), and 256_PACKED to 256-bit FP instructions (8 floats). The `4redux` configuration shows, on average, more than 5x increase in the vector FLOPs, the same as for `8redux_fuse`. These results are not correlated with the speedups in cycles, but they clearly indicate that MACVETH synthesizes more efficient SIMD code (more SIMD packed instructions than scalar instructions), as at the same time the number of micro-operations is reduced (see Table 5.3). Even though the best configuration for this metric (by an almost insignificant margin) is `4redux`, Figure 5.17 illustrates the percentage of scalar and vector FLOPs over the total FLOPs for the `8redux_fuse` configuration, which has the best speedups in cycles. These results show the trends observed in Table 5.4, where we can see how the MACVETH version of the code issues more vector FLOPs than the auto-vectorized version by GCC. In this figure we also include the metric in Equation 5.8, which computes the percentage of scalar FLOPs. For all MACVETH configurations these values decrease significantly.

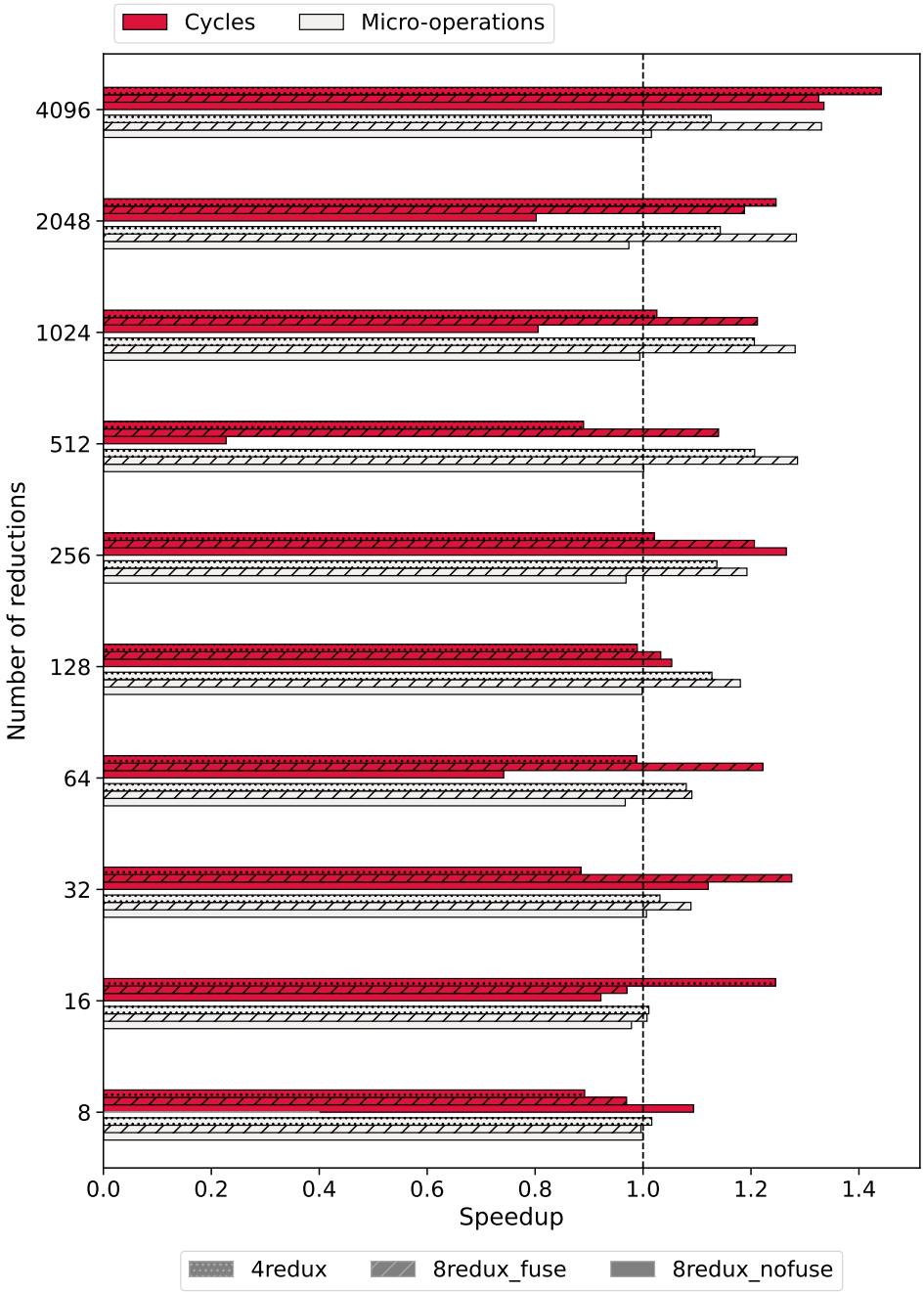


Figure 5.16: Speedups obtained in cycles and in the reduction of the number of micro-operations for the synthetic patterns for all MACVETH configurations.

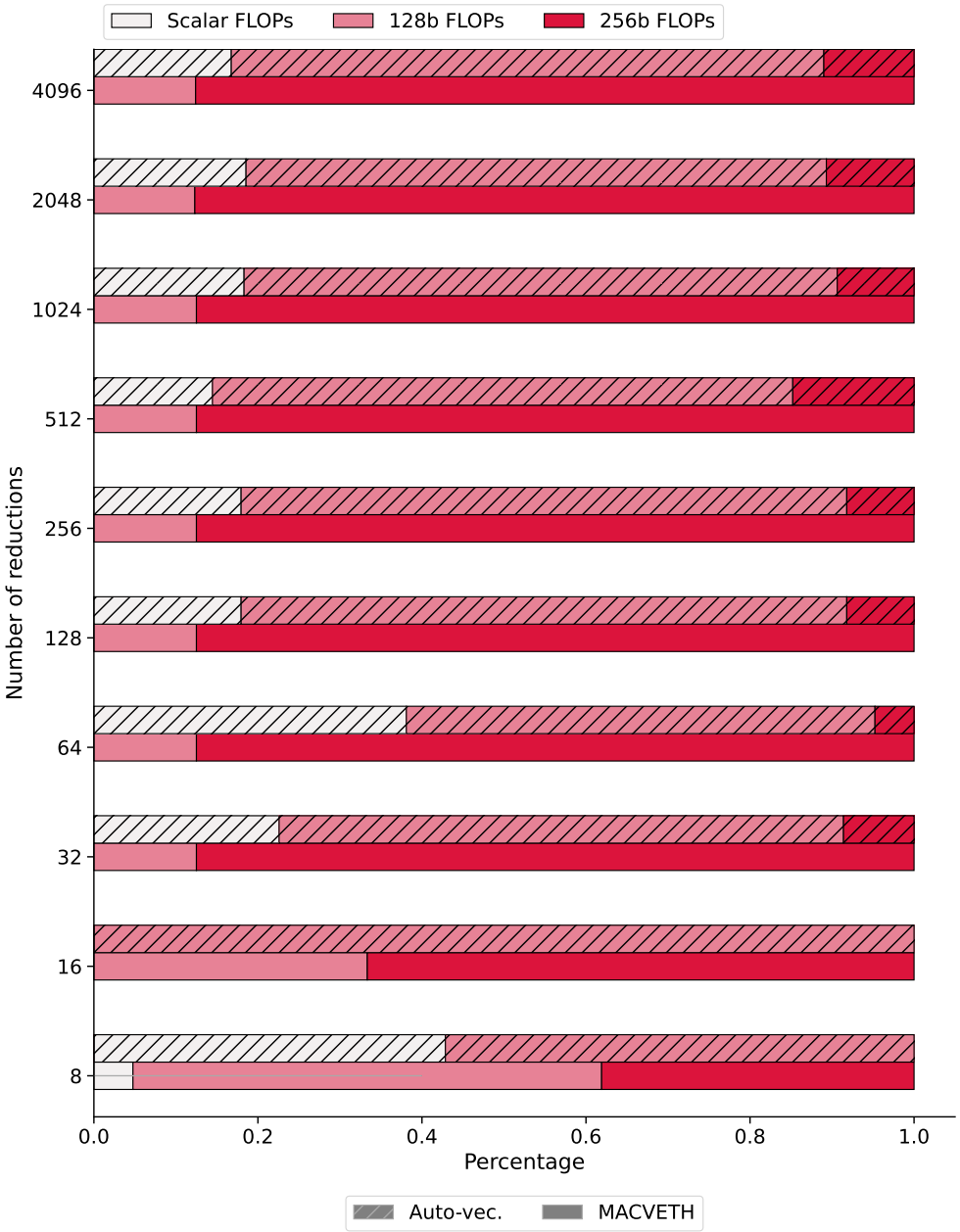


Figure 5.17: Percentage of scalar and vector FLOPs for the synthetic patterns with the GCC auto-vectorized version and the `8redux_fuse` MACVETH configuration.

Table 5.4: Comparison of MACVETH configurations for the synthetic patterns in terms of increment of vector FLOPs issued over the auto-vectorized version. Blue values indicate the maximum increment for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
`4redux_fuse`	5.099	0.556	4.000	4.907	5.220	5.441	5.895
`4redux_nofuse`	5.099	0.556	4.000	4.907	5.220	5.441	5.895
`8redux_fuse`	5.050	1.460	1.000	5.362	5.661	5.825	5.971
`8redux_nofuse`	1.197	0.189	0.971	1.117	1.171	1.238	1.750

$$VectorFLOPs = \frac{4 * 128\_\text{PACKED} + 8 * 256\_\text{PACKED}}{SCALAR + 4 * 128\_\text{PACKED} + 8 * 256\_\text{PACKED}} \tag{5.7}$$

$$ScalarFLOPs = \frac{SCALAR}{SCALAR + 4 * 128\_\text{PACKED} + 8 * 256\_\text{PACKED}} \tag{5.8}$$

5.4.2 Sparse matrices: SuiteSparse repository

We chose a subset of 150 matrices presented in [7] for these experiments (full list in Table C.1, Appendix C.3). The number of nonzero values (NNZ) for all these matrices is under 62,000. For completeness, we also selected 12 large sparse matrices from the collection with more than 1 million NNZ values. We used multiple configurations for MACVETH, as described in Table 5.5. Some of them include the option to consider for vectorization those orphan reductions which cannot be packed together (when `--novec-orphan-redux` is 'Disabled', we refer to Section 5.3.4 for further information and to Appendix C.1 for the compiler options).

Tables 5.6 and 5.7 show some descriptive statistics for all the configurations presented. These values are computed over all the matrices for each MACVETH configuration. In general, considering both cycles and micro-operations, we could infer that the best configuration is `4redux_noorphan_fuse`. In terms of cycles it gets the best average for all the matrices (12.4% speedup), with a central tendency above 9%, and a maximum of almost 200% (or 3x). It is remarkable that all MACVETH

Table 5.5: MACVETH configurations for the SuiteSparse matrices selected.

Name	`--min-redux-size`	`--nofuse`	`--novec-orphan-redux`
`4redux_noorphan_fuse`	4	Disabled	Enabled
`4redux_noorphan_nofuse`	4	Enabled	Enabled
`4redux_orphan_fuse`	4	Disabled	Disabled
`4redux_orphan_nofuse`	4	Enabled	Disabled
`8redux_noorphan_fuse`	8	Disabled	Enabled
`8redux_noorphan_nofuse`	8	Enabled	Enabled
`8redux_orphan_fuse`	8	Disabled	Disabled
`8redux_orphan_nofuse`	8	Enabled	Disabled

configurations improve, on average, the performance in cycles. In terms of micro-operations, the average reduction in almost all configurations is positive, varying from -1.2% to 8.2%, being the `4redux_noorphan_fuse` the best configuration again. From these values we can also conclude that the fusion of independent reductions (see Sections 5.3.4 and 5.3.5) provides benefits in terms of cycles and micro-operations issued. However, our implementation for the vectorization of orphan reductions does not provide any benefit on average, even though it achieves the best speedup in cycles (440% or 5.4x for `4redux_orphan_fuse`). We observed a degradation in cache performance by increasing almost 2x and 1.25x on average the number of cache misses in L2 and L3, respectively, on average, with regard to the same MACVETH configuration without this optimization. Note that out approach is quite naïve and does not consider the distance between the orphan reduction nodes, which could be causing a detrimental effect in cache locality, thereby negating any potential benefit of their vectorization. As evidenced in the results, relaxing the minimum size of reductions to pack to 4 for this set of matrices performs better than restricting the minimum size to 8. This is reasonable as these real sparse matrices present very irregular patterns, and for most of them restricting the size of reductions to pack to 8 elements prevents the vectorization of many computations.

Figures 5.18 and 5.19 detail the speedups in both cycles and micro-operations for the `4redux_noorphan_fuse` configuration for each matrix in the experimental set. 95 out of 150 matrices (63.3%) show positive speedup in cycles for MACVETH, being the configuration with the best percentage. Only the speedup in cycles of 8 out of 150 matrices (5.3%) is below 0.7%, in 10% of the cases is between 0.7% and 0.8%, in 9.3% is between 0.8% and 0.9%, and in 12% is between 0.9% and 1%.

Table 5.6: Comparison of MACVETH configurations for the set of matrices in terms of speedup in cycles. Blue values indicate the maximum speedup for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
4redux_noorphan_fuse	1.124	0.339	0.250	0.906	1.093	1.305	2.851
4redux_noorphan_nofuse	1.103	0.458	0.247	0.878	1.039	1.217	4.408
4redux_orphan_fuse	1.114	0.483	0.197	0.895	1.046	1.275	5.445
4redux_orphan_nofuse	1.103	0.435	0.265	0.811	1.058	1.271	4.056
8redux_noorphan_fuse	1.054	0.334	0.289	0.822	1.033	1.205	2.745
8redux_noorphan_nofuse	1.074	0.391	0.302	0.807	1.006	1.257	3.325
8redux_orphan_fuse	1.056	0.313	0.282	0.876	1.034	1.198	2.948
8redux_orphan_nofuse	1.031	0.412	0.202	0.820	0.976	1.205	4.515

Table 5.7: Comparison of MACVETH configurations for the set of matrices in terms of speedup in the reduction of the number of micro-operations retired. Blue values indicate the maximum speedup for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
4redux_noorphan_fuse	1.082	0.194	0.711	0.981	1.034	1.133	1.926
4redux_noorphan_nofuse	1.054	0.170	0.711	0.966	1.019	1.100	1.689
4redux_orphan_fuse	1.070	0.191	0.792	0.945	1.026	1.120	1.908
4redux_orphan_nofuse	1.037	0.165	0.734	0.943	0.998	1.082	1.711
8redux_noorphan_fuse	1.055	0.154	0.715	0.973	1.014	1.122	1.671
8redux_noorphan_nofuse	1.034	0.143	0.716	0.964	1.009	1.087	1.652
8redux_orphan_fuse	1.020	0.157	0.799	0.922	0.981	1.074	1.660
8redux_orphan_nofuse	0.988	0.142	0.718	0.898	0.959	1.039	1.632

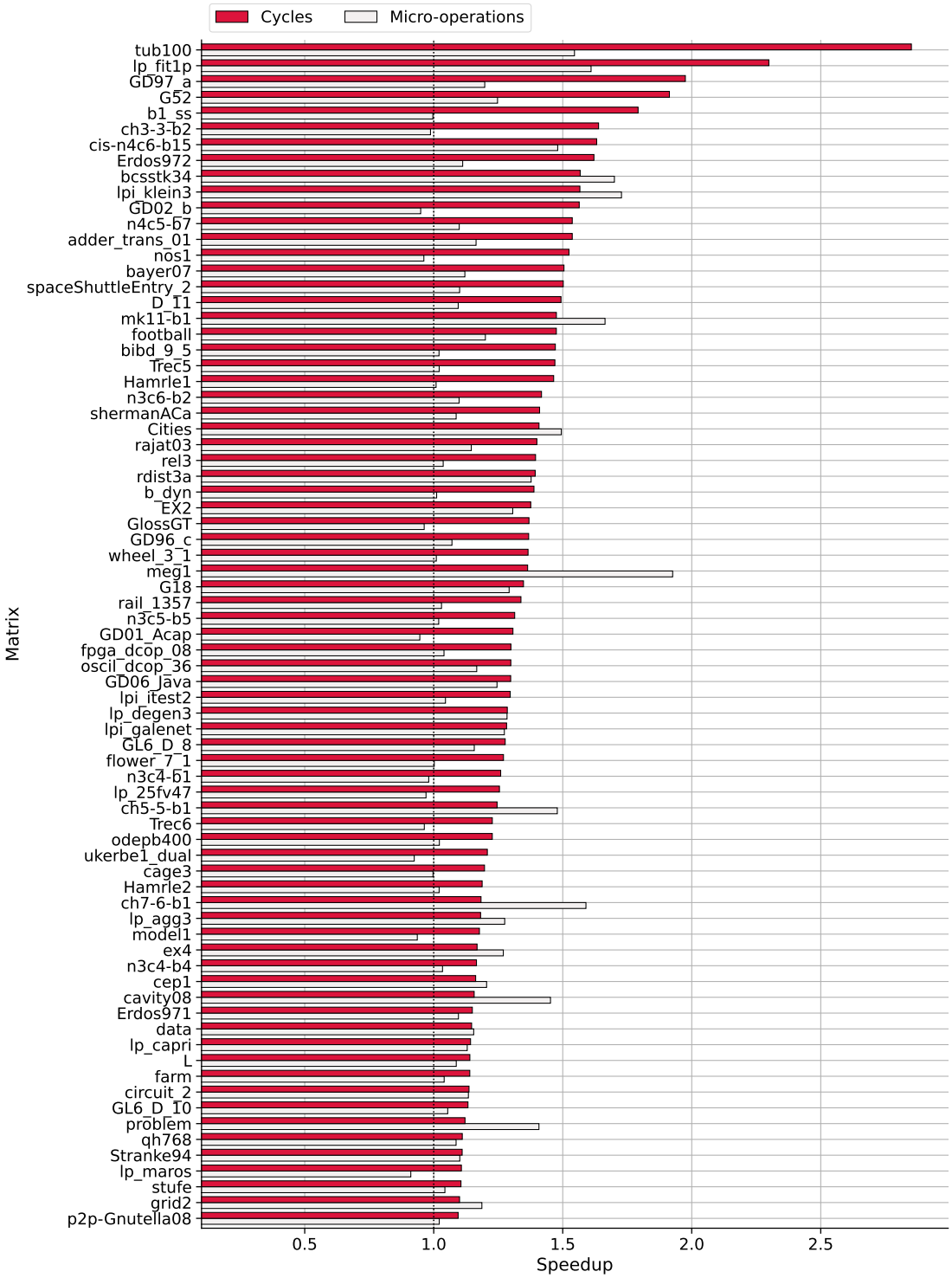


Figure 5.18: Speedups obtained for the `4redux_noorphan_fuse` configuration for the 150 matrices (under 62K NNZ) selected from [7]. Sorted in descending order according to the speedup in cycles (Part I).

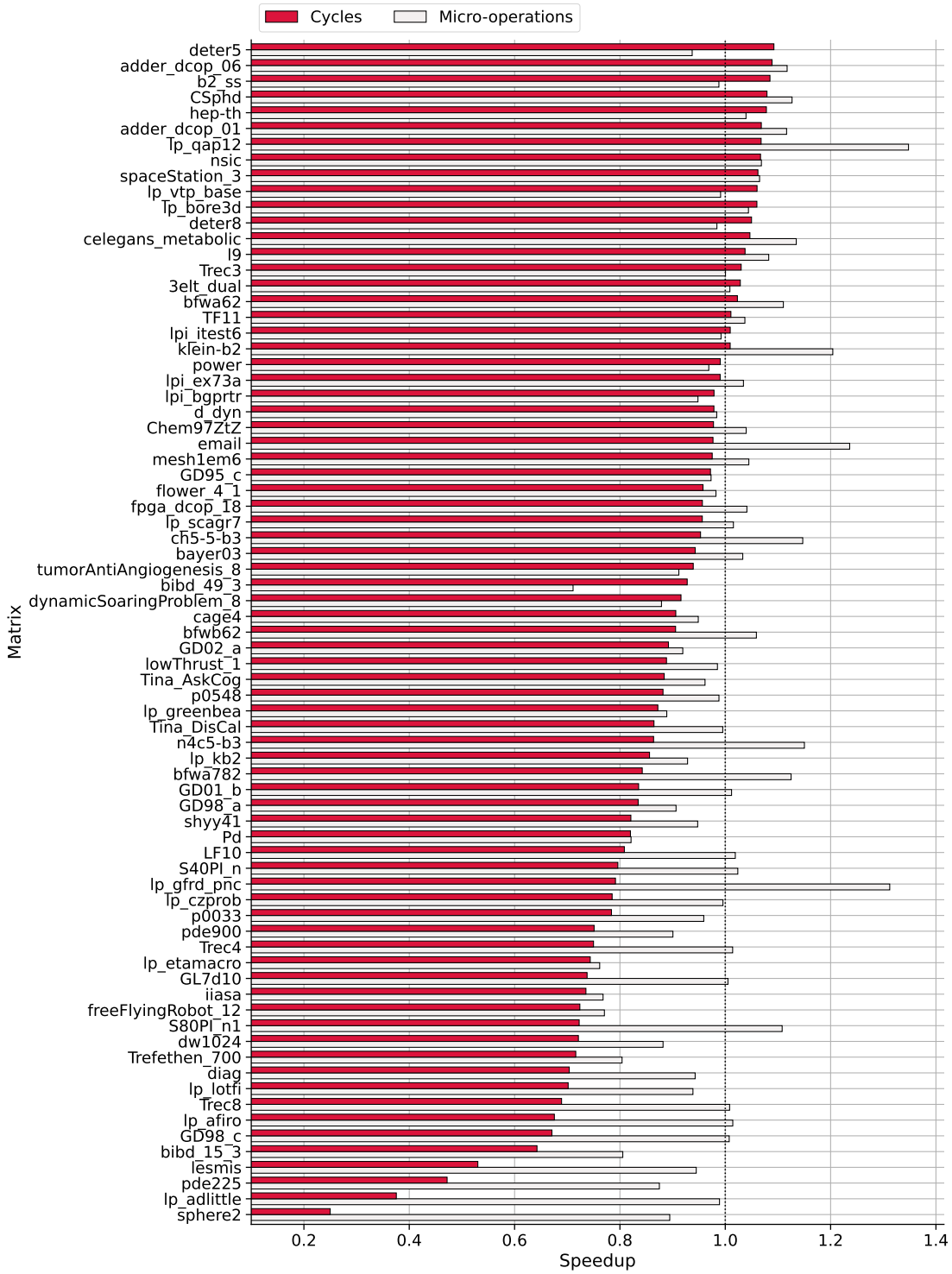


Figure 5.19: Speedups obtained for the `4redux_noorphan_fuse` configuration for the 150 matrices (under 62K NNZ) selected from [7]. Sorted in descending order according to the speedup in cycles (Part II).

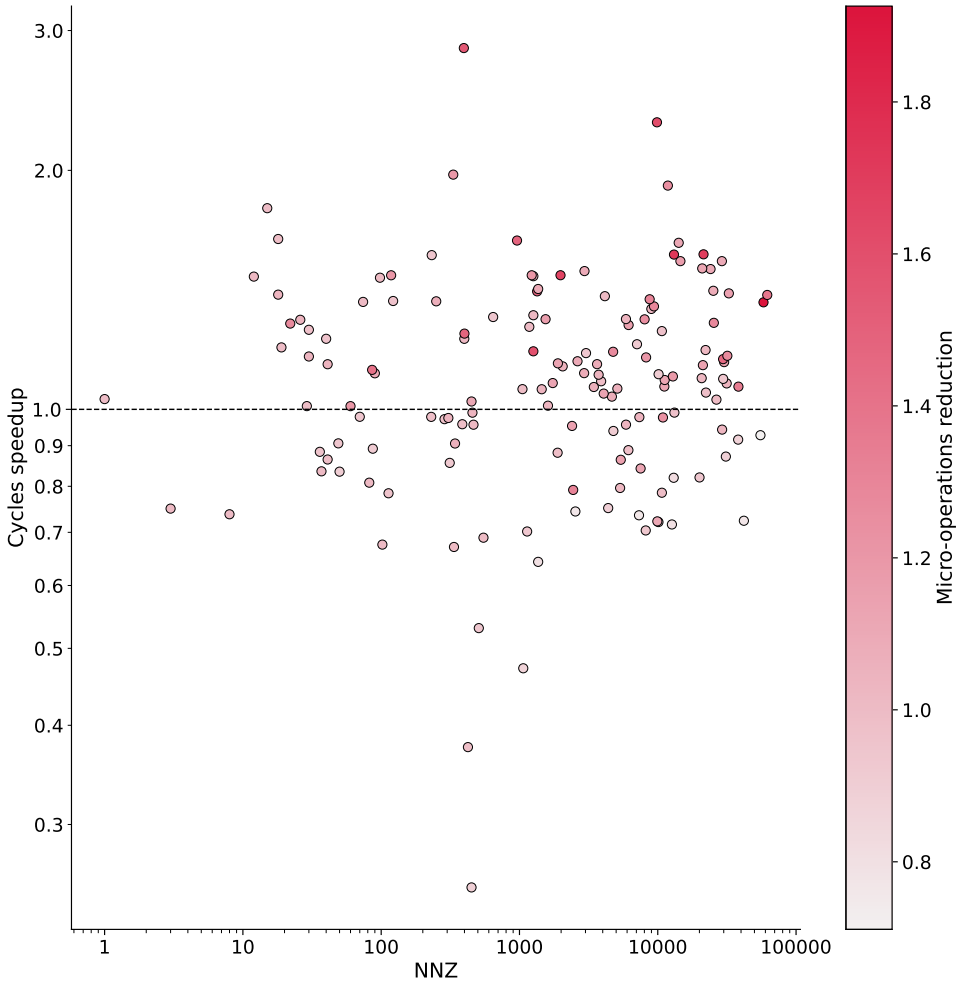


Figure 5.20: Speedups obtained for the `4redux_noorphan_fuse` configuration for the 150 matrices (under 62K NNZ) selected from [7] according to the NNZ values of the matrix and the micro-operations reduction. Note that the color of the points above the 1.0 threshold of cycles speedup tends to be darker, confirming the correlation between that speedup and the reduction of micro-operations retired.

There is a moderate connection between the number of instructions retired and the cycles consumed reflected in a Pearson correlation coefficient of R=0.52. This is graphically depicted in Figure 5.20. It shows an insignificant correlation between the number of NNZ values in the matrix and the speedup in cycles observed. 98 out of 150 matrices decrease the number of micro-operations retired, and 77 out matrices show positive speedup on both dimensions. In this way, obtaining speedups in these

Table 5.8: Comparison of MACVETH configurations for the set of matrices in terms of increment of vector FLOPs issued over the auto-vectorized version. Blue values indicate the maximum increment for each metric, but for the standard deviation (Std.), which shows the minimum.

Name	Mean	Std.	Min.	25%	50%	75%	Max.
`4redux_noorphan_fuse`	3.372	5.093	0.558	1.200	1.793	3.112	41.251
`4redux_noorphan_nofuse`	3.549	6.088	0.558	1.202	1.788	3.118	48.104
`4redux_orphan_fuse`	7.231	33.669	0.945	1.525	2.311	3.786	390.862
`4redux_orphan_nofuse`	7.228	33.669	0.945	1.519	2.286	3.785	390.823
`8redux_noorphan_fuse`	2.574	4.218	0.177	0.935	1.315	2.408	31.839
`8redux_noorphan_nofuse`	2.724	5.368	0.177	0.943	1.296	2.370	48.104
`8redux_orphan_fuse`	7.127	33.620	0.824	1.514	2.306	3.612	390.862
`8redux_orphan_nofuse`	7.108	33.620	0.824	1.512	2.247	3.568	390.823

two dimensions contribute to the energy efficiency of these kernels.

Table 5.8 shows the increment in the percentage of vector FLOPs over the auto-vectorized version for all MACVETH configurations, using the formula of Equation 5.7. In this case, the best average is obtained for `4redux_orphan_fuse`. The vectorization of orphan reductions provides an increment in the vector FLOPs synthesized. For the other configurations with this optimization enabled the results for this metric are very similar. Focusing on the configuration with the most promising speedups in cycles and micro-operations, Figures 5.21, 5.22 and 5.23 illustrate these data graphically for `4redux_noorphan_fuse`. The increment in scalar FLOPs (see Equation 5.8) is also included. There are only 15 out of 150 matrices where there is no increment in vector FLOPs but in the scalar FLOPs issued. Furthermore, there are only 4 cases where no packed vector instructions are synthesized. We have found no correlation (R=0.05) between the speedup in cycles and the increment of vector FLOPs in the data presented here. On the other hand, the vectorization of orphan reductions (`4redux_orphan_fuse` configuration), always reports an increment in vector FLOPs. This is a good sign, as the final goal of our approach is to vectorize irregular codes, but we should also consider not vectorizing a region if it is not profitable, i.e., MACVETH should include a more realistic cost model for issuing vector code.

Figure 5.24 shows the results obtained for the 12 large sparse matrices (NNZ > 1 million) using the `4redux_noorphan_fuse` MACVETH configuration. Table 5.9

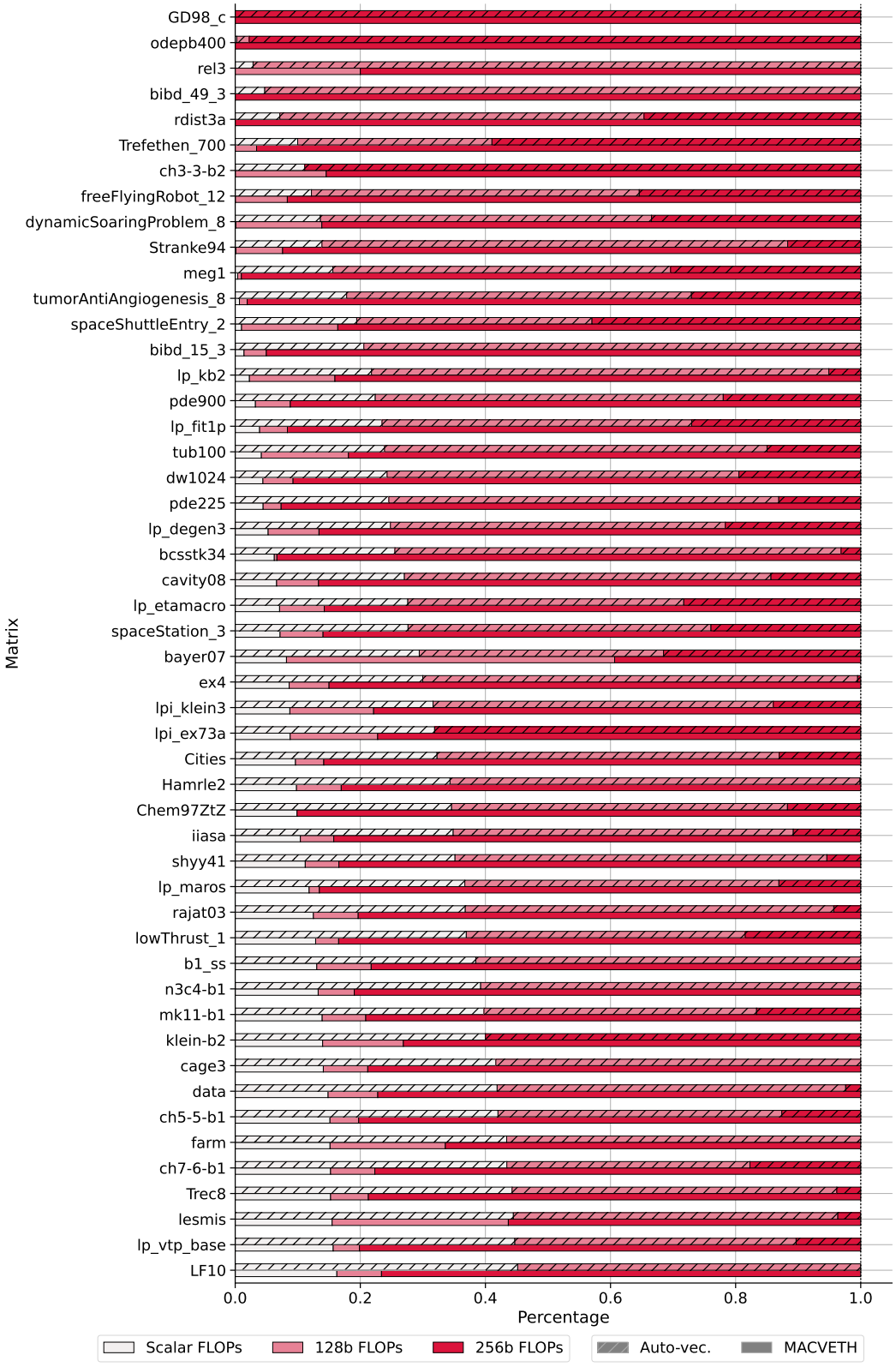


Figure 5.21: Percentage of scalar and vector FLOPs for the 150 matrices (under 62K NNZ) selected from [7] with the GCC auto-vectorized version and the `4redux_noorphan_fuse` MACVETH configuration (Part I).

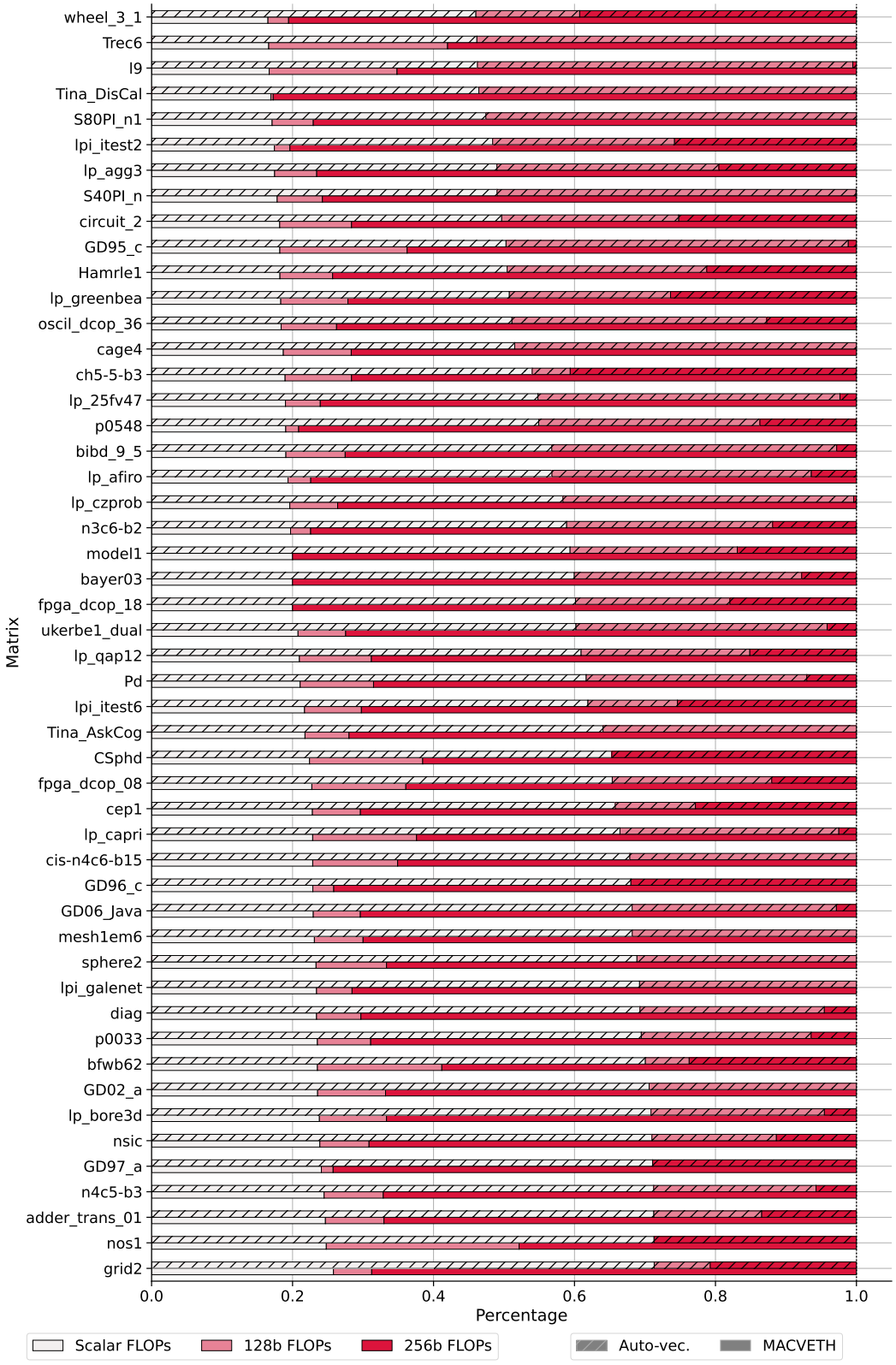


Figure 5.22: Percentage of scalar and vector FLOPs for the 150 matrices (under 62K NNZ) selected from [7] with the GCC auto-vectorized version and the `4redux_noorphan_fuse` MACVETH configuration (Part II).

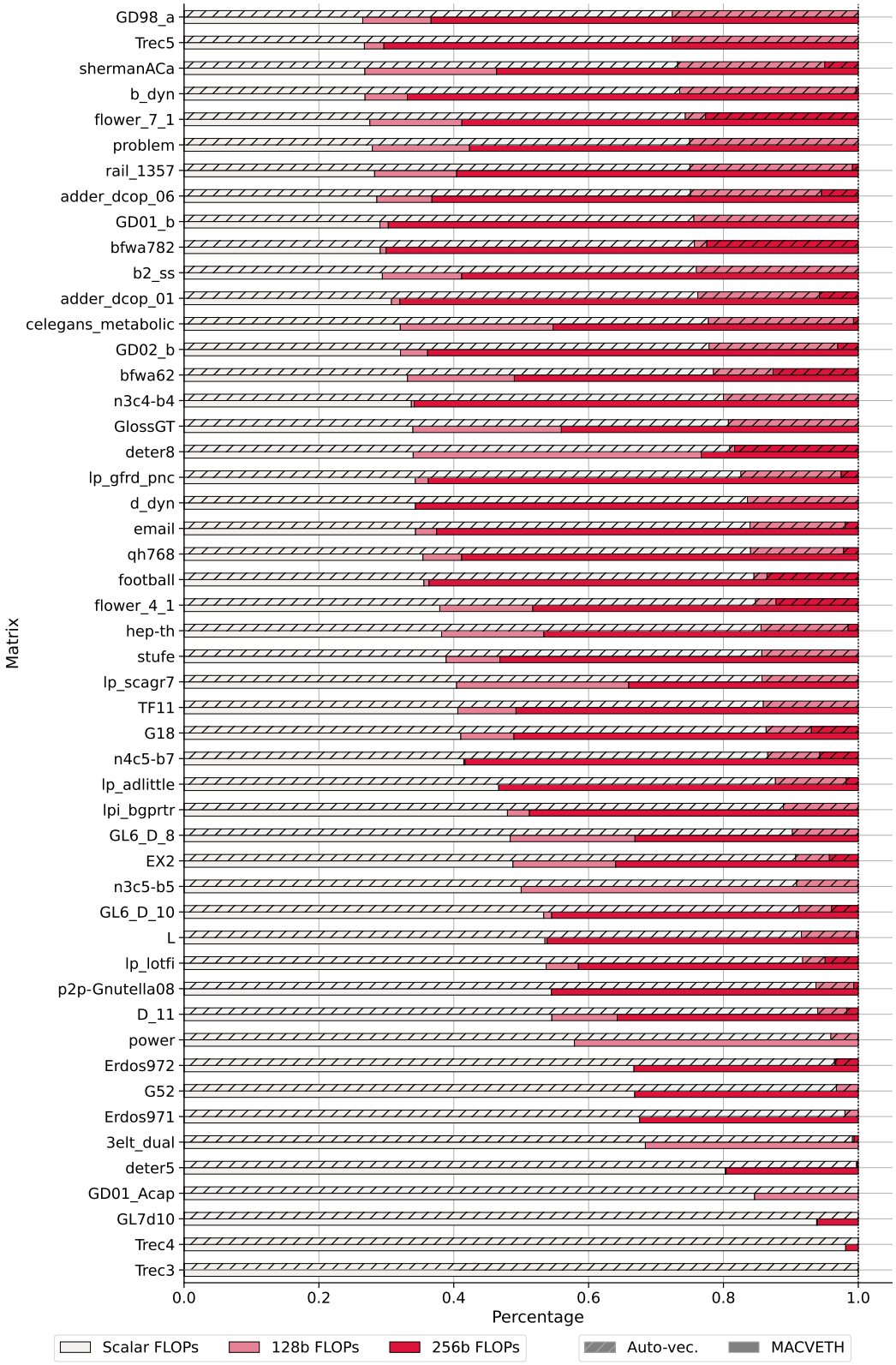
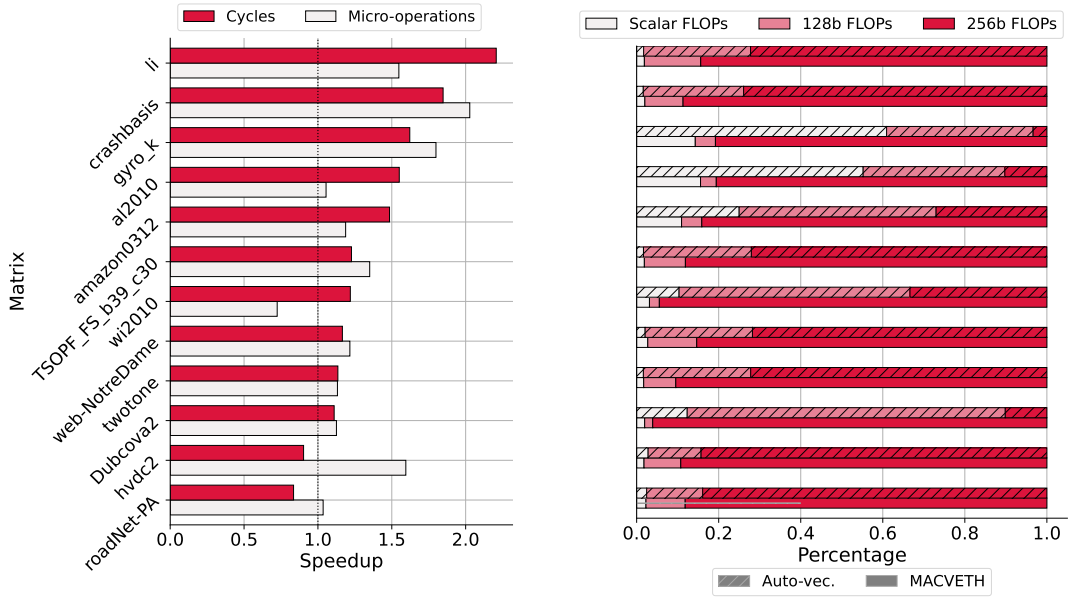


Figure 5.23: Percentage of scalar and vector FLOPs for the 150 matrices (under 62K NNZ) selected from [7] with the GCC auto-vectorized version and the `4redux_noorphan_fuse` MACVETH configuration (Part III).



(a) Speedup in cycles and micro-operations.

(b) Percentage of scalar and vector FLOPs.

Figure 5.24: Results obtained for the `4redux_noorphan_fuse` MACVETH configuration for large sparse matrices (>1M NNZ) sorted by speedup in terms of cycles. Both figures share the same y-label.

also presents the numerical results of Figure 5.24a. We show this configuration as it provides the best positive speedups. The compilation process of these matrices has been very time consuming, not only for MACVETH but also for GCC. In our machine (Intel Xeon Silver 4216), the compilation of these matrices in parallel and for each MACVETH configuration took more than 14 hours, while the execution took only several minutes. The main overhead of MACVETH's compilation is the generation and parsing of the Clang AST. Anyway, the results obtained outperform in 10 out of 12 cases the auto-vectorized version by GCC for this concrete configuration. Speedups are significant in both cycles and in the reduction of the number of micro-operations, as depicted in Figure 5.24a. We achieve the best results for the `li` matrix in terms of cycles (speedup of 120.7%), and for the `crashbasis` matrix in terms of micro-operations (reduction of 102.7%). The correlation coefficient for these two dimensions is almost 1, but the sample in this case is not very significant (12 matrices). Figure 5.24b also shows an increment in the percentage of vector FLOPs for all the matrices, while they all decrease the number of micro-operations retired.

Table 5.9: Speedups for large matrices using the `4redux_noorphan_fuse` MACVETH configuration sorted by NNZ values.

Matrix	NNZ	Cycles	micro-operations
gyro_k	1,021,159	1.621	1.799
Dubcova2	1,030,225	1.110	1.125
twotone	1,206,265	1.135	1.132
wi2010	1,209,404	1.219	0.724
li	1,215,181	2.207	1.547
al2010	1,230,482	1.550	1.055
hvdc2	1,339,638	0.902	1.595
web-NotreDame	1,497,134	1.166	1.216
crashbasis	1,750,416	1.847	2.027
roadNet-PA	3,083,796	0.835	1.035
TSOPF_FS_b39_c30	3,121,160	1.227	1.351
amazon0312	3,200,440	1.484	1.187

In this case, we have not found any real correlation between these dimensions or others. These results are in general very promising, and they show the potential benefits of our approach for large working sets. Note that these codes feature many irregular and different patterns, and our approach is able to vectorize all of them since we have implemented custom ad hoc vector packing techniques for random memory positions. We are able to pack together independent reductions within the same vector register, and fuse these computations maximizing vector occupancy. Again, for these matrices, the vectorization of orphan reductions has a detrimental impact on performance due to the same cache locality reasons described previously for the smaller matrices.

As a final and important note, the experiments shown in this section assess the correctness of the approach proposed, including all the different configurations for MACVETH. All experiments were fully automated using MARTA, only requiring semi-manual post-processing tasks for generating graphs and tables.

5.5 Related Work

Vectorization is key for exploiting hardware capabilities in modern architectures, improving the IPC, and for energy efficient-aware applications. For these reasons,

any compiler features auto-vectorization capabilities that typically target and exploit both blocks of instructions (SLP) [71] and loops (LLV). Mendis and Amarasinghe [90] propose an improvement to SLP vectorizers using an ILP-based (Integer Linear Programming) framework for packing elements in a pairwise optimal manner, but at an impractical cost in compilation time. Porpodas et al. [100, 101, 102] introduce several optimizations to the SLP vectorizer in LLVM, including the vectorization of arbitrarily long chains of commutative operations, and an algorithm capable of adjusting the vector width at an instruction granularity.

Regarding x86 architectures, the most popular open source compilers are GNU GCC [38] and Clang/LLVM [72]. Intel and AMD currently have their own in-house Clang/LLVM-based compilers (ICC [59] and AOCC [5] respectively), inheriting, at least, all auto-vectorization capabilities available in LLVM. They are meant to optimize code in their own architectures by rewriting the auto-vectorization cost models, among other optimizations (most of them are not open source). Pohl et al. [99] study the overestimation of these cost models implemented by GCC and Clang. Their findings show an overestimation in the speedup, resulting in mispredictions and a weak to medium correlation between predicted and actual performance gain. They propose a novel cost model based on a code IR with refined memory access pattern features.

Other works also target specific vectorization optimizations. Chen et al. [19] present VeGen, a vectorizer targeting Lane Level Parallelism, which captures the model of parallelism on SIMD and non-SIMD vector instructions. Non-SIMD instructions violate the two fundamental SIMD principles: element-wise operations, and being isomorphic across all lines. In this way, this vectorizer is able to find opportunities for these instructions in the codes without requiring ad hoc formulas in the peephole optimization stage of the compiler's back-end. Kjolstad et al. [66] present the Tensor Algebra Compiler (TACO), a framework for generating optimized kernels according to the computations to perform. Willsey et al. [132] present a library that exploits the equality saturation technique using e-graphs[6] for implementing rewrite-driven compiler optimizations and program synthesizers [129, 135]. The equality saturation (which can be seen as a specialization of the general SMT solvers) was first explored by the Peggy tool [122], by transforming the optimizations in the form

[6] An e-graph holds equivalence relations according to the semantics of the language.

of equality analysis into a common IR that encodes multiple optimized versions of the input program. Then, the heuristics choose the best candidate among the various programs represented [120]. Franchetti et al. [33] developed a framework using a mathematical formalism (the Operator Language, OL) for capturing computational algorithms and program transformations, in order to synthesize optimized code for each kernel for many target platforms. The ROSE compiler [104] provides a framework for building tools, similar to LLVM, for analysis and code transformations. Currently, as a new project of the LLVM framework, it was released MLIR [73], a new alternative for reducing the cost of building domain specific compilers, which aids in connecting existing compilers together.

There are also C++-centric template-based works in order to generate vector code. The Vector Class Library [32] provides a C++ library for writing SIMD code neither using Intrinsics nor assembly. The support is limited to x86 architectures and it uses Intel Intrinsics directives beneath. Eigen [41] is a C++ template library for linear algebra: matrices, vectors, numerical solvers and related algorithms. The Intel oneAPI Math Kernel Library [55] was designed for Intel processors to optimize math kernels, including BLAS routines such as vector-vector, vector-matrix and matrix-matrix multiplications, Fast Fourier Transforms (FFT), etc.

5.6 Concluding Remarks and Discussion

In this chapter we have presented two different SIMD optimizations for x86 architectures: random vector packing and fusion of reductions (intra- and inter-vector). The first one emulates the behavior and output of a gather instruction, but using other instructions available in the ISA. We have represented the exploration space for packing random operands, defining equivalence classes to describe all the possible cases to consider. Using and extending the Intel Intrinsics SIMD directives, we have developed MRKVS, an SMT-based system for generating and validating possible candidates for packing random operands according to the equivalence classes defined. The system is able to generate multiple candidates for each equivalence class. In this way we can build a cost model for each platform based on the performance of each candidate for each class. Our experiments show the potential benefits of our solutions compared to the gather instruction in both modern Intel and AMD

processors.

Irregular structures such as sparse matrices typically use formats (e.g., CSR) that are accessed through indirection arrays. Computations using these formats are not easily vectorizable. Augustine et al. [7] developed a system for mining regular sub-computations from irregular data structures, leading to specialized codes that avoid any type of indirection and improve the opportunities to generate vector code. We have developed MACVETH, a source-to-source compiler targeting the vectorization of irregular patterns such as reductions in SpMV codes. It includes in its cost model the random vector packing system we have developed. MACVETH performs the fusion of independent reductions within the same and also in different vector registers, leveraging the opportunities to maximize vector occupancy when performing these operations. We have conducted an exhaustive evaluation of our model for SpMV codes using synthetic patterns, and also for 150 sparse matrices from the SuiteSparse repository [21]. Our experiments show promising results for the optimizations presented in this chapter, improving the auto-vectorized code synthesized by the latest version of GCC. On the other hand, some promising optimizations, such as the vectorization of orphan reduction nodes, do not seem to perform as expected, and further investigation is required. We found that in most cases this optimization is negatively affecting cache locality, causing a degradation in L2 and last-level caches. This issue could be addressed by considering in the cost model the cache distance of the points to vectorize. We have also explored the performance on 12 large matrices (with between 1M and 3M NNZ values), but it would be necessary to increase the number of matrices in the analysis. Still, the results shown are solid, showing significant speedups in cycles (up to 120.7%). As a final note, the results presented in this chapter contribute to the energy efficiency of the application as we are reducing the number of micro-operations and thus energy consumption. Furthermore, our approach shows a significant speedup in terms of cycles, increasing the performance per watt of the system. Nevertheless, further measurements on energy consumption should be performed to assess the real impact on energy efficiency.

The MRKVS system presented in this chapter can be extended to any ISA. It is only required to provide the semantics of the instructions to consider. For a better performance, it would also be necessary to provide new heuristics adapted to the desired ISA, in order to better prune the exploration space. MRKVS leverages the

capabilities of the Z3 SMT solver to generate a set of valid candidates for packing operands in vectors. Other alternatives could be used instead of Z3, for instance, the recent egg library [132] (already described in Section 5.5), but we have not explored any other implementations as performance was not our major concern. MACVETH is a promising specific source-to-source compiler in an early stage. Its cost model can be extended to other modern x86 ISAs, such as AVX-512. According to the architecture described in this chapter, this compiler can be easily adapted to other modern ISAs such as ARM Neon, only requiring a new back-end (and probably minor modifications to the middle-end). The major advantage of MACVETH being a source-to-source compiler is the portability between architectures. At the same time, the compiler is platform-aware, generating specialized code for each architecture. In that case, for a better performance, the compiler should target explicitly the desired platform. Besides, since the output is also C code, the user (an expert user) can inspect it and even perform its own modifications. The main disadvantage of not generating machine code is the additional compilation time required to generate an executable binary. MACVETH currently supports multiple options in order to let the user choose the best configuration for each case. It would be a good feature to include different cost models that could automatically configure the best combination of parameters of the compiler for a given code. Currently MACVETH only implements a very simple heuristic to discard vectorization if the cost of packing the operands and operations is higher than just issuing scalar code.

"There is nothing permanent except change."

–Heraclitus

6 Concluding Remarks and Future Work

In this chapter, we summarize the main contributions proposed in this Thesis, as well as potential future research lines.

6.1 Conclusions and Discussion

Current trends in high performance computing exhibit parallelism at different levels: from instruction decoding and execution (Instruction-Level Parallelism, ILP) to the number of interconnected nodes in a cluster or worldwide. From an architectural perspective, parallelism is exhibited in the number of cores etched on a single die. This increment in the degree of components interconnected brings forward structural scalability complications. In addition, the design of modern cores are, nowadays, extremely sophisticated by implementing complex pipelines featuring wide vector length capabilities. For these reasons, in this Thesis we addressed these two orthogonal dimensions by analyzing modern manycore architectures and focusing on discovering potential design improvements, and providing techniques for synthesizing efficient platform-aware SIMD code.

In the first part of the Thesis we investigated the interconnection networks present in modern manycore architectures, and the performance impact of the traffic generated by the distributed cache coherence directories. We focused on the Intel Mesh Interconnect, first introduced in the Xeon Phi x200 Knights Landing,

and later featured in the subsequent Xeon Scalable generations. When we initially explored coherence traffic on the Knights Landing NoC, we observed a clear effect on application performance due to affinity relationships between cores and their cache directories (or CHAs), i.e., we observed a NUMA behavior. For this reason, we mapped the block-to-CHA correspondence for each cache line to physical parts of the NoC, depending on their physical address, in order to disclose the physical mapping of coherence data for memory blocks. Leveraging this mapping, we developed a dynamic scheduling in an inspector-executor fashion to characterize and optimize the impact of the core-to-CHA affinity. Based on these promising results, we extended this work by reverse engineering the functions responsible for this mapping. At this stage, we expected these functions to be useful in order to alleviate the runtime overhead of our inspector-executor approach, in addition to being usable by architecture-specific compilers that could perform low-level optimizations of coherence traffic. However, these expectations were toned down by the actual shape of the mapping functions. Although the XOR-based functions are cheap to implement in hardware and widely used for other non-regular mappings, such as the assignment between memory blocks and LLC slices in Intel Core processors, they are costly to compute in software. This cost could be overcome if the mapping presented some kind of regularity that could be exploited by carefully optimizing the code and scheduling, but that was not the case for KNL as the number of tiles was not a power of 2, producing nonlinear mapping functions. Our evaluation assessed the importance of data affinity for systems featuring distributed cache directories, and the significant performance impact of the coherence traffic. With manycore architectures firmly considered as the future of computer architecture, it would be desirable to improve these designs, coupling the directory distribution that avoids bottlenecks in the NoC with a more regular and predictable mapping of the memory blocks to enable programmers, particularly in the high performance computing domain, to have full control over coherence traffic. The approach followed in this part of the Thesis focused on the Intel Xeon Phi 7210, but it is potentially applicable to Xeon Scalable processors, which feature the same interconnection network.

Along the same lines, we built a model of the KNL architecture on Tejas, a state-of-the-art cycle-accurate architectural simulator. With this model, we were able to perform in-depth analysis of the behavior of the complex interconnection network. First we validated our model against real hardware, considering the instruction

limitations of the simulator (as Tejas does not support the full x86 ISA). Then, we also presented a case study analyzing the low-level behavior of the interconnection network for the previously proposed optimizations taking advantage of the physical location of cores and cache block holders, i.e., core-to-CHA distance, and also the thread mapping for parallel applications. Our evaluation confirmed the reduction of coherence traffic in the network and the reduction of collisions when tuning our applications to consider data affinity.

In an orthogonal dimension, the second part of the Thesis investigated SIMD optimizations for computations using irregular data structures such as sparse matrices. We focused on two concrete types of optimizations: the packing of random operands into the same vector register, and the fusion of independent reductions. For this part of the work we needed a framework for automating a high volume of experiments, in order to compile them using different configurations, profiling those programs using many different hardware counters, and gathering and analyzing all these data. For these reasons, we developed MARTA (Multi-configuration Assembly pRofiler and Toolkit for performance Analysis), a toolkit designed to increase the productivity of this type of micro-benchmarking that requires the configuration of many different parameters. MARTA, besides profiling, is also meant to extract knowledge from the generated data by applying data mining and machine learning techniques. In this way, MARTA is able to profile and characterize the performance according to an input set of dimensions of interest. We assessed MARTA with different cases of study as a good alternative for any type of profiling experiment, although it was originally conceived for building the cost models for our SIMD optimizations.

The first optimization we targeted was the efficient packing of random operands in vector registers for x86 architectures. The goal was to generate an efficient implementation of the gather macro-instruction (decoded into multiple micro-operations) using other single instructions from a concrete ISA. We first defined an exploration space according to the pool of valid instructions to use, and equivalence classes according to the contiguity and data type of the memory addresses to pack. With this information, we built MRKVS (Mega-Random Kernel Vector SMT), an SMT-based model for generating sets of instructions or candidates for each equivalence class. Using MARTA in each target architecture we built a platform-aware cost model. The candidates used for the equivalence classes defined depend on the target archi-

tecture. We assessed the performance of the candidates generated by the system against the latency and throughput of the equivalent gather instruction. For most cases these new candidates outperform gather by a wide margin.

Since we wanted to vectorize computations on irregular data structures, such as SpMV codes, we developed MACVETH (Multi-Architectural C-VEcTorizer for HPC applications), a source-to-source compiler based on Clang which includes the packing of random vector operands, as described above, and also the fusion of independent reductions. For that purpose we developed different algorithms for packing independent reductions within a program, synthesizing efficient SIMD code. We also integrated a strategy to vectorize those reduction operations which cannot be packed together by the algorithm (named orphan reductions). We evaluated the performance of our approach using different patterns and loop shapes. The results are promising, discovering potential optimization paths, and confirming the benefits of vectorizing those regions of code even when using irregular data structures in memory.

6.2 Future Work

Potential future work topics are discussed next:

- The Intel KNL architecture was discontinued, but its NoC legacy lives on the newer Intel Xeon Scalable processors, which also implement distributed cache directories in the form of snoop filters (the equivalent to the CHA in KNL) within a 2D mesh interconnection network. Our approach to improve data locality presented in Chapter 2 could be translated to these new architectures. Findings from McCalpin [88] reveal a layout similar to the KNL.
- Following the same research lines, a new extension for the Xeon Scalable processors could be implemented in Tejas, by modifying our existing model. In any case, having a model in software can help propose hypothesis about the theoretical performance of an application, and assess its reliability in a real platform, as we have described in Chapter 3.
- MARTA (Chapter 4) is a novel yet powerful tool. The profiling component

could be improved by further automating the generation of benchmark templates. New options could be added to the front-end for better parsing the configuration file, or for allowing a more flexible format. The Analyzer module can be improved by relaxing the configuration parameters required for the analysis. The system could iteratively choose the best combinations, avoiding user intervention. In the same way, the system could also support other types of analyzers and algorithms such as calibration analysis (using isotonic or logistic regression), which could be interesting to complement the knowledge extracted from a decision tree classifier.

- MACVETH (Chapter 5) was developed as a source-to-source compiler, targeting x86 architectures for random vector packing and fusion of reductions. We have assessed the performance optimizations implemented against a variety of irregular codes obtaining promising results, but further evaluation is required including other architectures (e.g., AMD, ARM) and new SIMD extensions in addition to AVX2 (e.g., AVX-512). In the same way, some of the potential optimizations included in the compiler, such as the vectorization of orphan reductions, have not contributed to the speedup as expected. This was a promising candidate to improve performance, but it caused certain degradation for the piecewise-regular SpMV codes used in our experiments. The reason for this behavior is the distance between these orphan reductions, which causes locality degradation and a consequent decrease in cache performance. It would be a nice research topic to determine the maximum distance allowed depending on the host platform to efficiently vectorize these nodes. As such, MACVETH could be improved by automatically determining the best configuration parameters for each platform by building a more sophisticated and platform-aware cost model.

Bibliography

[1] A. Abel and J. Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 673–686, Providence, RI, USA, 2019. doi: 10.1145/3297858.3304062. (pages: 103, 133, and 149)

[2] A. Abel and J. Reineke. nanoBench: A Low-Overhead Tool for Running Microbenchmarks on x86 Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46, Boston, MA, USA (Virtual), 2020. doi: 10.1109/ISPASS48437.2020.00014. (page: 125)

[3] A. Abel and J. Reineke. A Parametric Microarchitecture Model for Accurate Basic Block Throughput Prediction on Recent Intel CPUs. arXiv: 2107.14210, 2022. (page: 146)

[4] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010. doi: 10.1002/cpe.1553. (pages: 86, 125)

[5] Advanced Micro Devices, Inc. (AMD), *AOCC User Guide*, 2021. URL

https://developer.amd.com/wp-content/resources/57222_AOCC_UG_Rev_3.2.pdf. [Accessed: 01-03-2022]. (page: 193)

[6] J. H. Ahn, S. Li, S. O, and N. P. Jouppi. McSimA+: A Manycore Simulator with Application-Level Simulation and Detailed Microarchitecture Modeling. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85, Austin, TX, USA, 2013. doi: 10.1109/ISPASS.2013.6557148. (page: 82)

[7] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez. Generating Piecewise-Regular Code from Irregular Structures. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 625–639, Phoenix, AZ, USA, 2019. doi: 10.1145/3314221.3314615. (pages: 40, 46, 174, 175, 176, 181, 184, 185, 186, 188, 189, 190, and 195)

[8] A. Azad and A. Buluç. A Work-Efficient Parallel Sparse Matrix-Sparse Vector Multiplication Algorithm. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 688–697, Orlando, FL, USA, 2017. doi: 10.1109/IPDPS.2017.76. (pages: 8, 53)

[9] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical Report TR-2008-13, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008. URL http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf. (page: 3)

[10] O. Bilaniuk. libpfc, https://github.com/obilaniu/libpfc, [Accessed: 01-03-2022]. (page: 126)

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011. doi: 10.1145/2024716.2024718. (pages: 57, 82)

[12] S. M. Blackburn, A. Diwan, M. Hauswirth, P. F. Sweeney, J. N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, D. Frampton, L. J. Hendren, M. Hind, A. L. Hosking, R. E. Jones, T. Kalibera, N. Keynes, N. Nystrom, and A. Zeller. The Truth, the Whole Truth, and Nothing but the Truth: A Pragmatic Guide to Assessing Empirical Evaluations. *ACM Transactions on Programming Languages and Systems*, 38(4): 1–20, 2016. doi: 10.1145/2983574. (page: 94)

[13] Z. I. Botev, J. F. Grotowski, and D. P. Kroese. Kernel Density Estimation Via Diffusion. *The Annals of Statistics*, 38(5):2916–2957, 2010. doi: 10.1214/10-AOS799. (page: 92)

[14] BSC Performance Tools. Extrae, https://tools.bsc.es/extrae, [Accessed: 01-03-2022]. (page: 125)

[15] C. Byun, J. Kepner, W. Arcand, D. Bestor, B. Bergeron, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Prout, A. Rosa, S. Samsi, C. Yee, and A. Reuther. Benchmarking Data Analysis and Machine Learning Applications on the Intel KNL Many-Core Processor. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Waltham, MA, USA, 2017. doi: 10.1109/HPEC.2017.8091067. (pages: 53, 80)

[16] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–12, Seattle, WA, USA, 2011. doi: 10.1145/2063384.2063454. (page: 82)

[17] S. Charles, C. Patil, U. Ogras, and P. Mishra. Exploration of Memory and Cluster Modes in Directory-Based Many-Core CMPs. In *Proceedings of the 12th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 2:1–8, Torino, Italy, 2018. doi: 10.1109/NOCS.2018.8512154. (pages: 53, 57)

[18] L. Chen, B. Peng, B. Zhang, T. Liu, Y. Zou, L. Jiang, R. Henschel, C. Stewart, Z. Zhang, E. McCallum, Z. Tom, J. Omer, and J. Qiu. Benchmarking Harp-

DAAL: High Performance Hadoop on KNL Clusters. In *Proceedings of the 10th IEEE International Conference on Cloud Computing (CLOUD)*, pages 82–89, Honolulu, HI, USA, 2017. doi: 10.1109/CLOUD.2017.19. (pages: 53, 80)

[19] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe. VeGen: A Vectorizer Generator for SIMD and Beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 902—914, Virtual, 2021. doi: 10.1145/3445814.3446692. (page: 193)

[20] Chips and Cheese. How Zen 2's Op Cache Affects Performance, https://chipsandcheese.com/2021/07/03/how-zen-2s-op-cache-affects-performance/, [Accessed: 01-03-2022]. (page: 134)

[21] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38:1–25, 2011. doi: 10.1145/2049662.2049663. (pages: 40, 174, 195, and 234)

[22] B. Daya, C.-H. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. Chandrakasan, and L.-S. Peh. SCORPIO: A 36-Core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-Network Ordering. In *Proceedings of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 25–36, Minneapolis, MN, USA, 2014. doi: 10.1109/ISCA.2014.6853232. (page: 52)

[23] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, Budapest, Hungary, 2008. doi: 10.1007/978-3-540-78800-3_24. (pages: XXIII, 141, and 248)

[24] T. Downs. avx-turbo, https://github.com/travisdowns/avx-turbo, [Accessed: 01-03-2022]. (page: 4)

[25] T. Downs. uarch-bench, https://github.com/travisdowns/uarch-bench, [Accessed: 01-03-2022]. (page: 126)

[26] T. Downs. Gathering Intel on Intel AVX-512 Transitions, https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html, [Accessed: 01-03-2022]. (page: 4)

[27] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the ACM/IEEE 38th International Symposium on Computer Architecture (ISCA)*, pages 365–376, San Jose, CA, USA, 2011. doi: 10.1145/2000064.2000108. (page: 10)

[28] ExtremeTech. There's No Such Thing as "Huang's Law," Despite Nvidia's AI Lead, https://www.extremetech.com/computing/315277-theres-no-such-thing-as-huangs-law. (page: 3)

[29] A. Farshin, A. Roozbeh, G. Maguire, and D. Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the 14th EuroSys Conference*, pages 8:1–17, Dresden, Germany, 2019. doi: 10.1145/3302424.3303977. (page: 50)

[30] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo Directory: A Scalable Directory for Many-Core Systems. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture (HPCA)*, pages 169–180, San Antonio, TX, USA, 2011. doi: 10.1109/HPCA.2011.5749726. (page: 52)

[31] A. Fog. 4. Instruction Tables. Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD, and VIA CPUs, https://www.agner.org/optimize/instruction_tables.pdf, [Accessed: 01-03-2022]. (pages: 133, 149)

[32] A. Fog. Vector Class Library, https://github.com/vectorclass/version2, [Accessed: 01-03-2022]. (page: 194)

[33] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. SPIRAL: Extreme Performance Portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018. doi: 10.1109/JPROC.2018.2873289. (page: 194)

[34] Y. Fu and D. Wentzlaff. PriME: A Parallel and Distributed Simulator for Thousand-Core Chips. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, Monterey, CA, USA, 2014. doi: 10.1109/ISPASS.2014.6844467. (page: 82)

[35] N. Gawande, J. Landwehr, J. Daily, N. Tallent, A. Vishnu, and D. Kerbyson. Scaling Deep Learning Workloads: NVIDIA DGX-1/Pascal and Intel Knights Landing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 399–408, Orlando, FL, USA, 2017. doi: 10.1109/IPDPSW.2017.36. (page: 80)

[36] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010. doi: 10.1002/cpe.1556. (page: 125)

[37] GNU GCC. GCC 11 Release Series Changes, New Features, and Fixes, https://gcc.gnu.org/gcc-11/changes.html, [Accessed: 01-03-2022]. (page: 175)

[38] GNU GCC. Auto-Vectorization in GCC: Using the Vectorizer, https://gcc.gnu.org/projects/tree-ssa/vectorization.html, [Accessed: 01-03-2022]. (pages: 115, 116, 130, and 193)

[39] GNU GCC. 3.11 Options That Control Optimization, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [Accessed: 01-03-2022]. (page: 121)

[40] J. Goodman and H. Hum. MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects. Technical Report, University of Auckland, New Zealand, 2009. URL https://www.cs.auckland.ac.nz/~goodman/TechnicalReports/MESIF-2009.pdf. (page: 13)

[41] G. Guennebaud and B. Jacob. Eigen: A C++ Linear Algebra Library, http://eigen.tuxfamily.org, [Accessed: 01-03-2022]. (page: 194)

[42] J. Hammer, J. Eitzinger, G. Hager, and G. Wellein. Kerncraft: A Tool for Analytic Performance Modeling of Loop Kernels. In *Proceedings of the 10th*

International Workshop on Parallel Tools for High Performance Computing, pages 1–22, 2016. doi: 10.1007/978-3-319-56702-0_1. (page: 126)

[43] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *Proceedings of the Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, pages 57–64, Orlando, Florida, USA, 2014. doi: 10.1145/2568058.2568068. (page: 103)

[44] M. Horro, G. Rodríguez, J. Touriño, and M. T. Kandemir. Study of the Intel Knights Landing (KNL) Memory System Tradeoffs. In *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, pages 1–4, Fiuggi, Italy, 2017. (pages: XXI, XXIV, 245, and 249)

[45] M. Horro, M. T. Kandemir, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Effect of Distributed Directories in Mesh Interconnects. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 51:1–6, Las Vegas, NV, USA, 2019. doi: 10.1145/3316781.3317808. (pages: XXII, XXIV, 245, and 249)

[46] M. Horro, G. Rodríguez, and J. Touriño. Simulating the Network Activity of Modern Manycores. *IEEE Access*, 7:81195–81210, 2019. doi: 10.1109/ACCESS.2019.2923855. (pages: XXI, XXIII, 245, and 248)

[47] M. Horro, G. Rodríguez, and J. Touriño. papi_wrapper. https://github.com/UDC-GAC/papi_wrapper, 2019. (page: 70)

[48] M. Horro, G. Rodríguez, and J. Touriño. Tejas KNL Simulator. https://github.com/UDC-GAC/tejas_knl, 2019. (pages: 56, 64)

[49] M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Exploring SIMD Instructions for Packing Random Vector Operands in Modern x86 CPUs. In *Proceedings of the 17th International Summer School on Advanced Computer Architecture and Compilation for High-Performance Embedded Systems (ACACES)*, pages 143–146, Fiuggi, Italy, 2021. (pages: XXIV, 249)

[50] M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Toolkit para (Micro-) Benchmarking y Análisis de Características de Rendimiento en Kernels. In *Actas XXXI Jornadas Paralelismo (SARTECO)*, pages 303–312, Málaga, Spain, 2021. (pages: XXII, XXIV, 245, and 249)

[51] M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MACVETH: Multi-Architectural C-VEcTorizer for HPC applications. Submitted for publication, 2022. (pages: XXII, XXIV, 246, and 248)

[52] M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. Submitted for publication, 2022. (pages: XXII, XXIII, 245, and 248)

[53] M.-Y. Hsu. *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries*. Packt Publishing Limited, 2021. (page: 152)

[54] Intel Corporation. Excerpts from a Conversation with Gordon Moore: Moore's Law. http://large.stanford.edu/courses/2012/ph250/lee1/docs/Excepts_A_Conversation_with_Gordon_Moore.pdf, . [Accessed: 01-03-2022]. (page: 2)

[55] Intel Corporation. Intel® oneAPI Math Kernel Library, https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl.html, [Accessed: 01-03-2022]. (page: 194)

[56] Intel Corporation. Intel® oneAPI Toolkits, https://software.intel.com/content/www/us/en/develop/tools/oneapi.html#gs.1ptgr0, [Accessed: 01-03-2022]. (pages: 86, 125)

[57] Intel Corporation, *Intel® Xeon® Phi™ Processor Performance Monitoring Reference Manual—Volume 1: Registers (Rev. 002)*, 2017. URL https://www.intel.com/content/dam/develop/external/us/en/documents/intel-c2-ae-xeon-phi-e2-84-a2-processor-performance-monitoring-reference-manual-vol1-mar2017.pdf. [Accessed: 01-03-2022]. (page: 17)

[58] Intel Corporation, *Intel® Xeon® Phi™ Processor Performance Monitoring Reference Manual—Volume 2: Events*, 2017. URL https://www.intel.com/content/dam/develop/external/us/en/documents/intel-c2-ae-xeon-phi-e2-84-a2-processor-performance-monitoring-reference-manual-vol2-mar2017.pdf. [Accessed: 01-03-2022]. (pages: 13, 17)

[59] Intel Corporation, *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*, 2021. URL https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi_dpcpp_cpp_compiler_2021.4.pdf. [Accessed: 01-03-2022]. (page: 193)

[60] G. Irazoqui, T. Eisenbarth, and B. Sunar. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *Proceedings of the Euromicro Conference on Digital System Design (DSD)*, pages 629–636, Madeira, Portugal, 2015. doi: 10.1109/DSD.2015.56. (page: 50)

[61] M. Jacquelin, W. De Jong, and E. Bylaska. Towards Highly Scalable Ab Initio Molecular Dynamics (AIMD) Simulations on the Intel Knights Landing Manycore Processor. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 234–243, Orlando, FL, USA, 2017. doi: 10.1109/IPDPS.2017.26. (page: 53)

[62] J. Jeffers, J. Reinders, and A. Sodani. *Intel® Xeon® Phi™ Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufman, 2016. (pages: 12, 44, 57, and 65)

[63] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 423–428, Nice, France, 2009. doi: 10.1109/DATE.2009.5090700. (page: 59)

[64] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *43rd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 65–76, Atlanta, GA, USA, 2010. doi: 10.1109/MICRO.2010.51. (page: 53)

[65] D. Kirk. NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *Proceedings of the 6th International Symposium on Memory Management (ISMM)*, pages 103–104, Montreal, QC, Canada, 2007. doi: 10.1145/1296907.1296909. (page: 11)

[66] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The Tensor Algebra Compiler. *Proceedings of the ACM on Programming Languages*, 1 (OOPSLA):77:1–29, 2017. doi: 10.1145/3133901. (page: 193)

[67] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The Vampir Performance Analysis Tool-Set. In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 139–155. 2008. doi: 10.1007/978-3-540-68564-7_9. (page: 125)

[68] S. Kommrusch, M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Optimizing Coherence Traffic in Manycore Processors Using Closed-Form Caching/Home Agent Mappings. *IEEE Access*, 9:28930–28945, 2021. doi: 10.1109/ACCESS.2021.3058280. (pages: xxii, xxiii, 245, and 248)

[69] H. Krawczyk. LFSR-based Hashing and Authentication. In *Proceedings of the 14th Annual International Cryptology Conference (CRYPTO)*, pages 129–139, Santa Barbara, CA, USA, 1994. doi: 10.1007/3-540-48658-5_15. (page: 28)

[70] R. Kumar, T. G. Mattson, G. Pokam, and R. Van Der Wijngaart. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, chapter The Case for Message Passing on Many-Core Chips, pages 115–123. Springer, 2011. doi: 10.1007/978-1-4419-6460-1_5. (page: 8)

[71] S. Larsen and S. Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the 21st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 145–156, Vancouver, BC, Canada, 2000. doi: 10.1145/349299.349320. (pages: 156, 193)

[72] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the IEEE/ACM Interna-*

tional Symposium on Code Generation and Optimization (CGO), pages 75–88, San Jose, CA, USA, 2004. doi: 10.1109/CGO.2004.1281665. (pages: 151, 193)

[73] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, Virtual, 2021. doi: 10.1109/CGO51591.2021.9370308. (page: 194)

[74] J. Levon. OProfile, https://oprofile.sourceforge.io/news/, [Accessed: 01-03-2022]. (page: 125)

[75] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, New York, NY, USA, 2009. doi: 10.1145/1669112.1669172. (page: 59)

[76] J. Liu, J. Kotra, W. Ding, and M. Kandemir. Network Footprint Reduction Through Data Access and Computation Placement in NoC-based Manycores. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, pages 181:1–6, San Francisco, CA, USA, 2015. doi: 10.1145/2744769.2744876. (pages: 53, 57)

[77] LLVM. Auto-Vectorization in LLVM, https://llvm.org/docs/Vectorizers.html, [Accessed: 01-03-2022]. (page: 130)

[78] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-F. Ngai. Data Layout Transformations for Enhancing Data Locality on NUCA Chip Multiprocessors. In *Proceedings of the 18th IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 348–357, Raleigh, NC, USA, 2009. doi: 10.1109/PACT.2009.36. (page: 53)

[79] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis

Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200, 2005. doi: 10.1145/1064978.1065034. (page: 60)

[80] J. R. Madsen, M. G. Awan, H. Brunie, J. Deslippe, R. Gayatri, L. Oliker, Y. Wang, C. Yang, and S. Williams. Timemory: Modular Performance Analysis for HPC. In *Proceedings of the International Supercomputing Conference (ISC)*, pages 434–452, Frankfurt, Germany, 2020. doi: 10.1007/978-3-030-50743-5_22. (page: 126)

[81] H. Markram. The Human Brain Project. *Scientific American*, 306(6):50–55, 2012. doi: 10.1038/scientificamerican0612-50. (pages: 3, 240)

[82] T. Mattson. The Future of Many Core Computing: A Tale of Two Processors, https://cseweb.ucsd.edu/classes/fa12/cse291-c/talks/SCC-80-core-cern.pdf. (pages: 8, 56)

[83] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 48–65, Kyoto, Japan, 2015. doi: 10.1007/978-3-319-26362-5_3. (page: 50)

[84] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, 1995. (page: 112)

[85] J. D. McCalpin. HPL and DGEMM Performance Variability on the Xeon Platinum 8160 Processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 18:1–13, 2018. doi: 10.1109/SC.2018.00021. (pages: 52, 81)

[86] J. D. McCalpin. Observations on Core Numbering and "Core ID's" in Intel Processors. Technical Report TR-2020-01, Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2020. doi: 10.26153/tsw/10858. (pages: 51, 52)

[87] J. D. McCalpin. Mapping Addresses to L3/CHA Slices in Intel Processors.

Technical Report TR-2021-03, Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2021. doi: 10.26153/tsw/14539. (page: 52)

[88] J. D. McCalpin. Mapping Core and L3 Slice Numbering to Die Locations in Intel Xeon Scalable Processors. Technical Report TR-2021-02, Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2021. doi: 10.26153/tsw/11256. (pages: 51, 52, 200, and 246)

[89] J. D. McCalpin. Mapping Core, CHA, and Memory Controller Numbers to Die Locations in Intel Xeon Phi x200 ("Knights Landing", "KNL") Processors. Technical Report TR-2021-01, Texas Advanced Computing Center (TACC), The University of Texas at Austin, 2021. doi: 10.26153/tsw/13120. (pages: 17, 51, and 52)

[90] C. Mendis and S. Amarasinghe. goSLP: Globally Optimized Superword Level Parallelism Framework. *Proceedings of the ACM Programming Languages.*, 2 (OOPSLA):110:1–28, 2018. doi: 10.1145/3276480. (page: 193)

[91] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, Bangalore, India, 2010. doi: 10.1109/HPCA.2010.5416635. (page: 82)

[92] N. Mishra, C. Imes, J. Lafferty, and H. Hoffman. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 184–198, Williamsburg, VA, USA, 2018. doi: 10.1145/3296957.3173184. (page: 82)

[93] M. A. H. Monil, S. Lee, J. S. Vetter, and A. D. Malony. Understanding the Impact of Memory Access Patterns in Intel Processors. In *Proceedings of the Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 52–61, 2020. doi: 10.1109/MCHPC51950.2020.00012. (page: 37)

[94] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):114–118, 1965. doi: 10.1109/N-SSC.2006.4785860. (page: 1)

[95] K. O'Leary, I. Gazizov, A. Shinsel, R. Belenov, Z. Matveev, and D. Petunin. Intel Advisor Roofline Analysis. https://www.codeproject.com/articles/1169323/intel-advisor-roofline-analysis. [Accessed: 01-03-2022]. (page: 37)

[96] I. E. Papazian. New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP). In *Proceedings of the IEEE Hot Chips Symposium (HCS)*, pages 1–22, Los Alamitos, CA, USA, 2020. doi: 10.1109/HCS49909.2020.9220434. (page: 10)

[97] T. Par, T. Lapusan, and P. Grover. dtreeviz : Decision Tree Visualization, https://github.com/parrt/dtreeviz, [Accessed: 01-03-2022]. (page: 93)

[98] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011. (page: 90)

[99] A. Pohl, B. Cosenza, and B. Juurlink. Portable Cost Modeling for Auto-Vectorizers. In *Proceedings of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 359–369, Rennes, France, 2019. doi: 10.1109/MASCOTS.2019.00046. (pages: 130, 193)

[100] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. Look-Ahead SLP: Auto-Vectorization in the Presence of Commutative Operations. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 163–174, Vienna, Austria, 2018. doi: 10.1145/3168807. (page: 193)

[101] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes. VW-SLP: Auto-Vectorization with Adaptive Vector Width. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 12:1–15, Limassol, Cyprus, 2018. doi: 10.1145/3243176.3243189. (page: 193)

[102] V. Porpodas, R. C. O. Rocha, E. Brevnov, L. F. W. Góes, and T. Mattson. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 206–216, Washington, DC, USA, 2019. doi: 10.1109/CGO.2019.8661192. (page: 193)

[103] L.-N. Pouchet. PolyBench: The Polyhedral Benchmark Suite. https://sourceforge.net/projects/polybench/. [Accessed: 01-03-2022]. (pages: 68, 90, 93, 127, and 175)

[104] D. Quinlan and C. Liao. The ROSE Source-to-Source Compiler Infrastructure. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (in conjunction with PACT)*, pages 1–3, Galveston Island, TX, USA, 2011. URL https://engineering.purdue.edu/Cetus/cetusworkshop/papers/4-1.pdf. (page: 194)

[105] S. Ramos and T. Hoefler. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, Orlando, FL, USA, 2017. doi: 10.1109/IPDPS.2017.30. (page: 53)

[106] S. Rho, G. Park, J.-S. Kim, S. Kim, and D. Nam. A Study on Optimal Scheduling Using High-Bandwidth Memory of Knights Landing Processor. In *Proceedings of the 2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 289–294, Tucson, AZ, USA, 2017. doi: 10.1109/FAS-W.2017.161. (page: 81)

[107] R. C. O. Rocha, V. Porpodas, P. Petoumenos, L. F. W. Góes, Z. Wang, M. Cole, and H. Leather. Vectorization-Aware Loop Unrolling with Seed Forwarding. In *Proceedings of the 29th International Conference on Compiler Construction (CC)*, pages 1–13, San Diego, CA, USA, 2020. doi: 10.1145/3377555.3377890. (page: 156)

[108] G. Rodríguez, M. Kandemir, and J. Touriño. Affine Modeling of Program Traces. *IEEE Transactions on Computers*, 68(2):294–300, 2019. doi: 10.1109/TC.2018.2853747. (page: 45)

[109] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 12–27, San Diego, CA, USA, 1988. doi: 10.1145/73560.73562. (page: 152)

[110] P. Salihundam, S. Jain, T. Jacob, S. Kumar, V. Erraguntla, Y. Hoskote, S. Vangal, G. Ruhl, P. Kundu, and N. Borkar. A 2Tb/s 6×4 Mesh Network with DVFS and 2.3Tb/s/W Router in 45nm CMOS. In *Proceedings of the Symposium on VLSI Circuits*, pages 79–80, Honolulu, HI, USA, 2010. doi: 10.1109/VLSIC.2010.5560277. (page: 67)

[111] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. *ACM SIGARCH Computer Architecture News*, 41(3):475–486, 2013. doi: 10.1145/2508148.2485963. (pages: 57, 82)

[112] S. R. Sarangi, R. Kalayappan, P. Kallurkar, and S. Goel. Tejas Simulator: Validation against Hardware. arXiv: 1501.07420v1, 2015. (page: 59)

[113] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. Tejas: A Java Based Versatile Micro-Architectural Simulator. In *Proceedings of the 25th International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, pages 47–54, Salvador, Brazil, 2015. doi: 10.1109/PATMOS.2015.7347586. (pages: 56, 57, 59, 61, and 63)

[114] scikit-learn. DecisionTreeClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html#sklearn.tree.DecisionTreeClassifier, [Accessed: 01-03-2022]. (page: 102)

[115] scikit-learn. RandomForestClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, [Accessed: 01-03-2022]. (page: 102)

[116] A. Scolari, D. Bartolini, and M. Santambrogio. A Software Cache Partitioning System for Hash-Based Caches. *ACM Transactions on Architecture and Code Optimization*, 13(4):1–24, 2016. doi: 10.1145/3018113. (page: 50)

[117] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *The International Journal of High Performance Computing Applications*, 20(2): 287–311, 2006. doi: 10.1177/1094342006064482. (page: 125)

[118] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, 1986. doi: 10.1002/bimj.4710300745. (page: 92)

[119] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog, and R. Ronen. Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 4–9, Huntington Beach, CA, USA, 2001. doi: 10.1109/LPE.2001.945363. (page: 133)

[120] M. Stepp, R. Tate, and S. Lerner. Equality-Based Translation Validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, pages 737–742, Snowbird, UT, USA, 2011. doi: 10.1007/978-3-642-22110-1_59. (page: 194)

[121] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, IL, USA, 2012. (page: 68)

[122] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 264–276, Savannah, GA, USA, 2009. doi: 10.1145/1480881.1480915. (page: 193)

[123] TechXplore. Trillion-Transistor Chip Breaks Speed Record, https://techxplore.com/news/2020-11-trillion-transistor-chip.html. (page: 2)

[124] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms*. John Wiley & Sons, 1992. doi: 10.1002/9781118033104. (page: 159)

[125] TOP500.org. TOP500 - November 2021, https://www.top500.org/lists/top500/list/2021/11/. (page: 3)

[126] L. Torvalds. Re: SCO: "thread creation is about a thousand times faster than on native Linux". https://lkml.org/lkml/2000/8/25/132, 2000. (page: 55)

[127] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW)*, pages 207–216, San Diego, CA, USA, 2010. doi: 10.1109/ICPPW.2010.38. (page: 125)

[128] S. Wang, C. Li, H. Hoffman, S. Lu, W. Sentosa, and A. Kistijantoro. Understanding and Auto-Adjusting Performance-Sensitive Configurations. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 154–168, Williamsburg, VA, USA, 2018. doi: 10.1145/3173162.3173206. (page: 82)

[129] Y. R. Wang, S. Hutchison, J. Leang, B. Howe, and D. Suciu. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proceedings of the VLDB Endowment*, 13(12):1919–1932, 2020. doi: 10.14778/3407790.3407799. (page: 193)

[130] Z. Wegner. SPRDPL: Simple Python Recursive-Descent Parsing Library, https://github.com/zwegner/sprdpl/tree/bf971e4ff5832fe2a1fd34deca57ba60a3687a73, [Accessed: 01-03-2022]. (page: 141)

[131] Z. Wegner. x86-sat, https://github.com/zwegner/x86-sat, [Accessed: 01-03-2022]. (page: 141)

[132] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):23:1–29, 2021. doi: 10.1145/3434304. (pages: 193, 196)

[133] Y. Xiao, Y. Xue, S. Nazarian, and P. Bogdan. A Load Balancing Inspired Optimization Framework for Exascale Multicore Systems: A Complex Networks Approach. In *Proceedings of the International Conference on*

Computer-Aided Design (ICCAD), pages 217–224, Irvine, CA, USA, 2017. doi: 10.1109/ICCAD.2017.8203781. (page: 53)

[134] L. Yang, W. Liu, P. Chen, N. Guan, and M. Li. Task Mapping on SMAT NoC: Contention Matters, not the Distance. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, pages 88:1–6, Austin, TX, USA, 2017. doi: 10.1145/3061639.3062323. (page: 57)

[135] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar. Equality Saturation for Tensor Graph Superoptimization. arXiv: 2101.01332, 2021. (page: 193)

[136] A. Yasin. A Top-Down Method for Performance Analysis and Counters Architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, Monterey, CA, USA, 2014. doi: 10.1109/ISPASS.2014.6844459. (page: 125)

[137] K. Yotov, K. Pingali, and P. Stodghill. Automatic Measurement of Memory Hierarchy Parameters. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 181–192, Banff, AB, Canada, 2005. doi: 10.1145/1071690.1064233. (page: 81)

A

MARTA Configuration

A.1 CLI Options

MARTA integrates two main drivers, the `marta_profiler` (Profiler module) and the `marta_analyzer` (Analyzer module). The first one presents the following CLI:

```
usage: marta_profiler [-h] [--version] {project,po,profile,perf} ...

Productivity-aware framework for micro-architectural profiling and performance
characterization

optional arguments:
    -h, --help            shows this help message and exits
    --version             shows program's version number and exits

subcommands:
    {project,po,profile,perf}
                          additional help
    project (po)          project help
    profile (perf)        profile help
```

The `marta_profiler` driver features two main group of functionalities, `project` and `profile`. The first one is meant to help write new projects and benchmarks, providing default templates and configuration files. Its options are:

```
usage: marta_profiler project [-h] [-n NAME] [-u] [-c CHECK_CONFIG_FILE]
                              [-dump]

project subcommand is meant to help with the configuration and generation of
new MARTA-compliant projects.

optional arguments:
    -h, --help            shows this help message and exits
    -n NAME, --name NAME  name of the new project
    -u, --microbenchmark  sets new project as micro-benchmark
    -c CHECK_CONFIG_FILE, --check-config-file CHECK_CONFIG_FILE
                          quits if there is an error checking the configuration
                          file
    -dump, --dump-config-file
                          dumps a sample configuration file with all necessary
                          files for the profiler to work properly
```

The `profile` functionalities, as the name suggests, provide all options relative to micro-benchmarking:

```
usage: marta_profiler profile [-h] [-o OUTPUT] [-r] [-d]
                              [-log {debug,info,warning,error,critical}]
                              [-nsteps ITERATIONS] [-nexec EXECUTIONS]
                              [-x] [-q] [-v] [-s SUMMARY [SUMMARY ...]]
                              input

MARTA requires an input file with the configuration parameters, but some of
them can be overwritten at runtime, as described below.

optional arguments:
    -h, --help            shows this help message and exits

required named arguments:
    input                 input configuration file

optional named arguments:
    -o OUTPUT, --output OUTPUT
                          output results file name
    -r, --report          output report file name, with data regarding the
```

```
                        machine, compilation flags, warnings, and errors
    -d, --debug         verbose debugging messages
    -log {debug,info,warning,error,critical}, --loglevel  {debug,info,warning,
                                                            error,critical}
                        log level
    -nsteps ITERATIONS, --iterations ITERATIONS
                        number of iterations
    -nexec EXECUTIONS, --executions EXECUTIONS
                        number of executions
    -x, --no-quit-on-error
                        quits if there is an error during compilation or
                        execution of the kernel
    -q, --quiet         quiet execution
    -v, --version       displays version and quits
    -s SUMMARY [SUMMARY ...], --summary SUMMARY [SUMMARY ...]
                        prints a summary at the end with the given dimensions
                        of interest
```

The `marta_analyzer` driver provides an interface with the Analyzer module. The list of options available are:

```
usage: marta_analyzer [-h] [-q] [-v] [-dump] [input]

wrapper for analyzing data given a csv file

optional arguments:
    -h, --help          shows this help message and exits

required named arguments:
    input               input configuration file

optional named arguments:
    -q, --quiet         quiet execution
    -v, --version       displays version and quits
    -dump, --dump-config-file
                        dumps a sample configuration file with all necessary
                        files for analyzer to work properly
```

A.2 C Macros/Directives

Table A.1: Description of the macros included in the headers of MARTA. Some of them use functions and directives included in the PolyBench/C libraries, which are also integrated in the system.

Macro	Pseudo-code	Description

MARTA_BENCHMARK _BEGIN	`int main(int argc, char **argv) {`
	`  cpu_set_t mask;`
	`  CPU_ZERO(&mask);`
	`  CPU_SET(MARTA_CPU_AFFINITY, &mask);`
	`  int result = sched_setaffinity(0,`
	`  sizeof(mask),`
	`  &mask);`
		Sets benchmark's start. It also sets CPU affinity for the execution using Linux system calls.

MARTA_BENCHMARK _END	`  return 0;` `}`	Sets the end of the main function in the program.

MARTA_FLUSH_CACHE	`for (int i = 0; i < s; i += 64) {`
	`  asm volatile("clflush (%0)\n\t" ::`
	`  "r"(&p[i]) :`
	`  "memory");`
	`}`
		Performs `clflush` of size `s` at memory address `p`. Flush is performed at cache line level, typically 64 bytes.

PROFILE_ FUNCTION(X)	polybench_start_instruments; #if TSTEPS>1 __asm volatile("mov $"(TSTEPS)", %%ecx" ::: "ecx"); __asm volatile("begin_loop:"); #endif X; #if TSTEPS>1 __asm volatile("sub $1, %%ecx\n\t" "jne begin_loop"); #endif polybench_stop_instruments; polybench_print_instruments;	Instruments function X in a loop with TSTEPS iterations without performing any loop optimization. This is useful for collecting stable measurements, even though it causes a hot cache effect for small functions without saturating caches. It avoids the loop if TSTEPS is not greater than 1.
CLOBBER_MEM	asm volatile("" : : : "memory");	Acts as a read-/write barrier. Useful for micro-benchmarking, avoiding the reuse of values in register and bringing values from memory.
DO_NOT_TOUCH(X)	asm volatile("" : "+x"(var) : :);	Useful to avoid any compiler optimization on a variable X materialized in a register.

B

Random Vector Packing: Instructions

B.1 Load Instructions

Table B.1: Load instructions for the float data type considered in our model.

Instruction	Mnemonics	Description
`_mm_load_ss()`	`movss xmm, MEM`	Loads 32 bits in the LSB of the register.
`_mm_load_ps()`	`movaps xmm, MEM`	Loads 4 32-bit contiguous elements (aligned).
`_mm_loadu_ps()`	`movups xmm, MEM`	Loads 4 32-bit contiguous elements (unaligned).
`_mm256_load_ps()`	`vmovaps ymm, MEM`	Loads 8 32-bit contiguous elements (aligned).
`_mm256_loadu_ps()`	`vmovups ymm, MEM`	Loads 8 32-bit contiguous elements (unaligned).
`_mm_maskload_ps()`	`vmaskmovps xmm, xmm, MEM`	Loads 4 32-bit contiguous elements, zeroing those whose value in the mask is zero.
`_mm256_maskload_ps()`	`vmaskmovps ymm, ymm, MEM`	Loads 8 32-bit contiguous elements, zeroing those whose value in the mask is zero.

B.2 Swizzle Instructions

Table B.2: Swizzle instructions for the float data type considered in our model.

Instruction	Mnemonics	Description
_mm_shuffle_ps()	shufps xmm, xmm, xmm, imm8	Shuffles 4 elements.
_mm256_shuffle_ps()	vshufps ymm, ymm, ymm, imm8	Shuffles 8 elements within 128-bit lanes.
_mm_blend_ps()	blendps xmm, xmm, imm8	Blends 4 elements.
_mm256_blend_ps()	vblendps ymm, ymm, ymm, imm8	Blends 8 elements.
_mm_extract_ps()	extractps r32, xmm, imm8	Extracts 32 bits from a 128-bit register.
_mm256_extractf128_ps()	vextractf128 xmm, ymm, imm8	Extract 32 bits from a 256-bit register.
_mm_insert_ps()	insertps xmm, xmm, imm8	Inserts a position from one register into another.
_mm256_insertf128_ps()	vinsertf128 ymm, ymm, xmm, imm8	Inserts a complete 128-bit lane into another register.
_mm_permute_ps()	vpermilps xmm, xmm, imm8	Permutes within a 128-bit register.
_mm256_permute_ps()	vpermilps ymm, ymm, imm8	Permutes within a 256-bit register (not crossing 128-bit lanes).
_mm256_permute2f128_ps()	vperm2f128 ymm, ymm, ymm, imm8	Permutes within 128-bit lanes.
_mm_permutevar_ps()	vpermilps xmm, xmm, xmm	Permutes within a register.
_mm256_permutevar_ps()	vpermilps ymm, ymm, ymm	Permutes within 128-bit lanes.
_mm256_permutevar8x32_ps()	vpermps ymm, ymm, ymm	Permutes 8 elements (crossing lanes).

C

MACVETH Configuration

C.1 CLI Options

MACVETH requires an input file to compile, but it also accepts the set of options listed next:

```
usage: macveth [options] <source0> [...<sourceN>]

options:
generic options:

  --help                              - Displays available options (--help-hidden
                                        for more)
  --help-list                         - Displays list of available options
                                        (--help-list-hidden for more)
  --version                           - Displays the version of this program

MACVETH options:
  --debug-file=<string>               - Output file to print the debug
                                        information
  --debug-mv                          - Prints debug information
  --fma                               - Support for FMA instructions
  --format-fallback-style=<string> - The name of the predefined style used as
                                        a fallback in case clang-format is
                                        invoked with -format-style=file, but cannot
```

```
                                find the .clang-format file to use. Use
                                -format-fallback-style=none to skip
                                formatting
--format-style=<string>       - Coding style, currently supports:
                                LLVM, GNU, Google, Chromium,
                                Microsoft, Mozilla, WebKit.
                                Use -format-style=file to load style
                                configuration from .clang-format file
                                located in one of the parent directories
                                of the source file (or current directory
                                for stdin). Use -format-style="{key: value,
                                 ...}" to set specific parameters, e.g.:
                                -format-style="{BasedOnStyle: llvm,
                                IndentWidth: 8}"
--func=<string>               - Target function to vectorize
--march=<value>               - Target architecture
    =cascadelake                -   Intel Cascade Lake (2019) architecture
    =znver3                     -   AMD Zen3 (2020) architecture: AVX2
    =amd                        -   AMD architecture not specified
    =intel                      -   Intel architecture not specified
--min-redux-size=<int>        - Advanced option: minimum number of
                                reductions to pack together
--misa=<value>                - Target ISA
    =native                     -   Detects ISA of the architecture
    =sse                        -   SSE ISA
    =avx                        -   AVX ISA
    =avx2                       -   AVX2 ISA
--no-format                   - MACVETH by default reformats code as
                                clang-format does, using LLVM style. If
                                this option is enabled, then no
                                reformatting is applied
--no-headers                  - If set, does *not* include header files
--no-svml                     - Disables Intrinsics SVML
--nofma                       - Disables FMA optimizations
--nofuse                      - Disables the fusion of reductions
--novec-orphan-redux          - Disables the vectorization of orphan
                                reductions
-o=<string>                   - Output file to write the code, otherwise
                                it will just print in the standard output
--redux-win-size=<int>        - Advanced option: size of window of
                                reductions to consider
```

```
--scatter                          - Supports AVX-512 scatter instructions
--simd-info=<string>               - Report with all the SIMD information
```

C.2 Pragma Options

In order to control the unrolling of the loops, MACVETH provides different directives or options for the pragmas. A region of interest for MACVETH might include basic blocks or affine loops. The syntax should be as described in Listing C.1. The available options for the pragmas are:

- `unroll [var0 val0] [...[varN valN]]`: explicitly tells the compiler the unroll factor of each dimension of the nested loop/s. `valN` can be either a positive integer or `full`. This is the default option so it does not have to be explicit, with a factor of 4 if the upper bound is not known at compile time, and full unrolling otherwise.
- `nounroll`: avoids unrolling the code within the region. This may be useful if we have an irregular code and we just want to vectorize it. This has the same behavior as unrolling with factor 1.
- `unroll_and_jam`: performs unroll-and-jam for loop nests within the region of interest (if possible). Currently this is only implemented for those loop nests where the loop bounds are known, i.e., for performing full unroll-and-jam.
- `scalar|nosimd`: does not generate SIMD code; useful, for instance, for only unrolling the code.

```
1 #pragma macveth <options>
2 // Region of interest
3 #pragma endmacveth
4 ...
5 #pragma macveth <options>
6 // Another region of interest
7 #pragma endmacveth
```

Listing C.1: Pragmas required to indicate the regions of interest for MACVETH.

C.3 Matrices Used

Table C.1: Full description of the sparse matrices from SuiteSparse [21] used for the experiments (sorted by NNZ).

Name	Group	Columns	Rows	NNZ
Trec3	JGD_Kocay	2	1	1
Trec4	JGD_Kocay	3	2	3
GL7d10	JGD_GL7d	60	1	8
Trec5	JGD_Kocay	7	3	12
b1_ss	Grund	7	7	15
ch3-3-b2	JGD_Homology	18	6	18
rel3	JGD_Relat	5	12	18
cage3	vanHeukelum	5	5	19
lpi_galenet	LPnetlib	14	8	22
lpi_itest2	LPnetlib	13	9	26
lpi_itest6	LPnetlib	17	11	29
n3c4-b1	JGD_Homology	6	15	30
n3c4-b4	JGD_Homology	15	6	30
Tina_AskCog	Pajek	11	11	36
GD01_b	Pajek	18	18	37
Trec6	JGD_Kocay	15	6	40
farm	Meszaros	17	7	41
Tina_DisCal	Pajek	11	11	41
cage4	vanHeukelum	9	9	49
GD98_a	Pajek	38	38	50
klein-b2	JGD_Homology	30	20	60
lpi_bgprtr	LPnetlib	40	20	70
wheel_3_1	JGD_Margulies	25	21	74
LF10	Oberwolfach	18	18	82
problem	Meszaros	46	12	86
GD02_a	Pajek	23	23	87
Stranke94	Pajek	10	10	90
Hamrle1	Hamrle	32	32	98
lp_afiro	LPnetlib	51	27	102
p0033	Meszaros	48	15	113
football	Pajek	35	35	118
GlossGT	Pajek	72	72	122
d_dyn	Grund	87	87	230
GD02_b	Pajek	80	80	232

GD96_c	Pajek	65	65	250
GD95_c	Pajek	62	62	287
mesh1em6	Pothen	48	48	306
lp_kb2	LPnetlib	68	43	313
GD97_a	Pajek	84	84	332
GD98_c	Pajek	112	112	336
bfwb62	Bai	62	62	342
flower_4_1	JGD_Margulies	129	121	386
tub100	Bai	100	100	396
odepb400	Bai	400	400	399
ch5-5-b1	JGD_Homology	25	200	400
lp_adlittle	LPnetlib	138	56	424
sphere2	Pothen	66	66	450
bfwa62	Bai	62	62	450
lpi_ex73a	LPnetlib	211	193	457
lp_scagr7	LPnetlib	185	129	465
lesmis	Newman	77	77	508
Trec8	JGD_Kocay	84	23	549
GD01_Acap	Pajek	953	953	645
cis-n4c6-b15	JGD_Homology	920	60	960
lp_vtp_base	LPnetlib	346	198	1051
pde225	Bai	225	225	1065
lp_lotfi	LPnetlib	366	153	1136
flower_7_1	JGD_Margulies	393	463	1178
football	Newman	115	115	1226
ch7-6-b1	JGD_Homology	42	630	1260
bibd_9_5	JGD_BIBD	126	36	1260
n3c5-b5	JGD_Homology	252	210	1260
Cities	Pajek	46	55	1342
n3c6-b2	JGD_Homology	105	455	1365
bibd_15_3	JGD_BIBD	455	105	1365
lp_bore3d	LPnetlib	334	233	1448
oscil_dcop_36	Sandia	430	430	1544
TF11	JGD_Forest	236	216	1607
CSphd	Pajek	1882	1882	1740
p0548	Meszaros	724	176	1887
lp_capri	LPnetlib	482	271	1896
mk11-b1	JGD_Homology	55	990	1980
GL6_D_10	JGD_GL6	341	163	2053
ch5-5-b3	JGD_Homology	600	600	2400
lp_gfrd_pnc	LPnetlib	1160	616	2445

lp_etamacro	LPnetlib	816	400	2537
Erdos971	Pajek	472	472	2628
qh768	Bai	768	768	2934
D_11	JGD_SL6	461	169	2952
model1	Meszaros	798	362	3028
nsic	Meszaros	897	465	3449
L	AG-Monien	956	956	3640
stufe	AG-Monien	1036	1036	3736
b2_ss	Grund	1089	1089	3895
celegans_metabolic	Arenas	453	453	4065
b_dyn	Grund	1089	1089	4144
pde900	Bai	900	900	4380
l9	Meszaros	1483	244	4659
lp_agg3	LPnetlib	758	516	4756
tumorAntiAngiogenesis_8	VDOL	490	490	4776
spaceStation_3	VDOL	467	467	5103
S40PI_n	Rommes	2182	2182	5341
n4c5-b3	JGD_Homology	455	1350	5400
fpga_dcop_08	Sandia	1220	1220	5888
fpga_dcop_18	Sandia	1220	1220	5892
lowThrust_1	VDOL	584	582	6133
GL6_D_8	JGD_GL6	637	544	6153
ukerbe1_dual	AG-Monien	1866	1866	7076
iiasa	Meszaros	3639	669	7317
Chem97ZtZ	Bates	2541	2541	7361
bfwa782	Bai	782	782	7514
GD06_Java	Pajek	1538	1538	8032
diag	AG-Monien	2559	2559	8184
cep1	Meszaros	4769	1521	8233
EX2	JGD_SPG	560	560	8736
rail_1357	Oberwolfach	1357	1357	8985
G18	Gset	800	800	9388
lp_fit1p	LPnetlib	1677	627	9868
S80PI_n1	Rommes	4028	4028	9927
dw1024	Bai	2048	2048	10114
lp_maros	LPnetlib	1966	846	10137
lp_25fv47	LPnetlib	1876	821	10705
lp_czprob	LPnetlib	3562	929	10708
email	Arenas	1133	1133	10902
adder_dcop_01	Sandia	1813	1813	11156
adder_dcop_06	Sandia	1813	1813	11224

G52	Gset	1000	1000	11832
Trefethen_700	JGD_Trefethen	700	700	12654
grid2	AG-Monien	3296	3296	12864
Pd	MathWorks	8081	8081	13036
lpi_klein3	LPnetlib	1082	994	13101
power	Newman	4941	4941	13188
Erdos972	Pajek	5488	5488	14170
adder_trans_01	Sandia	1814	1814	14579
shyy41	Shyy	4720	4720	20042
p2p-Gnutella08	SNAP	6301	6301	20777
bayer07	Grund	3268	3268	20963
circuit_2	Bomhof	4510	4510	21199
bcsstk34	Boeing	588	588	21418
Hamrle2	Hamrle	5952	5952	22162
deter8	Meszaros	10905	3831	22299
spaceShuttleEntry_2	VDOL	1428	1428	24073
shermanACa	Shen	3432	3432	25220
lp_degen3	LPnetlib	2604	1503	25432
3elt_dual	AG-Monien	9000	9000	26556
n4c5-b7	JGD_Homology	4735	3635	29080
bayer03	Grund	6747	6747	29195
cavity08	DRIVCAV	1182	1182	29675
deter5	Meszaros	14529	5103	29715
data	DIMACS10	2851	2851	30186
lp_greenbea	LPnetlib	5598	2392	31070
hep-th	Newman	8361	8361	31502
ex4	FIDAP	1601	1601	31849
rajat03	Rajat	7602	7602	32653
dynamicSoaringProblem_8	VDOL	3543	3543	38136
lp_qap12	LPnetlib	8856	3192	38304
freeFlyingRobot_12	VDOL	5578	5578	41940
bibd_49_3	JGD_BIBD	18424	1176	55272
meg1	Grund	2904	2904	58142
rdist3a	Zitney	2398	2398	61896
gyro_k	Oberwolfach	17361	17361	1021159
Dubcova2	UTEP	65025	65025	1030225
twotone	ATandT	120750	120750	1206265
wi2010	DIMACS10	253096	253096	1209404
li	Li	22695	22695	1215181
al2010	DIMACS10	252266	252266	1230482
hvdc2	HVDC	189860	189860	1339638

web-NotreDame	SNAP	325729	325729	1497134
crashbasis	QLi	160000	160000	1750416
roadNet-PA	SNAP	1090920	1090920	3083796
TSOPF_FS_b39_c30	TSOPF	120216	120216	3121160
amazon0312	SNAP	400727	400727	3200440

D

Resumo Estendido en Galego

Historicamente, o desenvolvemento do *hardware* seguiu a coñecida Lei de Moore, que establecía que o número de compoñentes presentes nos circuítos integrados duplicaríase cada dous anos. Esta lei formulouse dende un punto de vista económico, pero pronto correlacionouse co desempeño dos computadores debido a que os transistores máis pequenos eran capaces de operar a frecuencias máis altas. Con todo, nas últimas décadas as frecuencias dos procesadores non se incrementaron ao mesmo ritmo ao que se reduciu o tamaño dos transistores. A lei do escalado de Dennard establece, *grosso modo*, que a densidade enerxética pode manterse constante ao reducir o tamaño dos transistores (baseados na tecnoloxía MOSFET). Un aspecto importante desta lei era que a redución no tamaño dos transistores permitía reducir a voltaxe e, así, os transistores poderían operar a frecuencias máis altas sen incrementar a densidade enerxética. O problema deste escalado é que ignorou dous factores limitantes: a voltaxe umbral e a corrente de fuga. A primeira establece a potencia mínima que require un transistor para operar correctamente. A segunda complica a disipación térmica dos chips, limitando o número de transistores que poden operar simultaneamente no chip sen deteriorarse fisicamente. Estas limitacións establecen o que se coñece hoxe en día como o "muro enerxético" (*power wall*).

Desta maneira, é discutíbel se a Lei de Moore segue vixente ou non, ou canto vai aguantar, pero está claro que a ruptura no escalado de Dennard condicionou as tendencias no desenvolvemento das microarquitecturas. A frecuencia xa non é o factor clave en canto ao rendemento, motivando o desenvolvemento de novas

aproximacións para acadar a computación a exascala, tales como o *dark silicon*, arquitecturas heteroxéneas, reconfigurables e *manycores*. Estas últimas fan referencia a aqueles procesadores que empaquetan un gran número de núcleos no mesmo chip sobre unha rede de interconexión. A computación a exascala refírese á habilidade de executar 10^{18} operacións en punto flotante nun segundo (FLOPS/s). É un dos retos a curto prazo máis ambiciosos para a computación de altas prestacións. A data 2021 aínda non hai listado no Top500 ningún supercomputador que alcance tal rendemento, aínda que conseguiuse chegar á exascala baixo certas condicións empregando computación distribuída. Esta magnitude de FLOPS/s permitiría unha mellor precisión en aplicacións e tarefas científicas complexas como a predición meteorolóxica, que depende dun vasto número de parámetros mutuamente dependentes; a simulación neuronal (The Human Brain Project [81]), que require simular billóns de neuronas interconectadas; a medicina personalizada para clasificar patoloxías baseadas en historiais médicos; aplicacións de dinámica de fluidos que requiren solucións máis precisas que as actuais aproximacións para a resolución das ecuacións diferenciais parciais de Navier-Stokes; e moitas outras. Deste xeito, este reto é moi importante para a enxeñaría de computadores.

Por estes motivos, o paralelismo gañou protagonismo no desenvolvemento do *hardware*. Ademais, o paralelismo preséntase de varias formas na computación de altas prestacións: dende a decodificación e execución de instrucións (paralelismo a nivel de instrución, ILP), ao número de nodos interconectados nun *cluster* ou a nivel global. Dende un punto de vista arquitectónico, o paralelismo exhíbese no número de núcleos gravados nunha placa. Con todo, este incremento no número de elementos interconectados complica a escalabilidade dos sistemas. Ademais, hoxe en día os núcleos dos procesadores son extremadamente sofisticados xa que implementan complexas arquitecturas segmentadas con amplos anchos de banda para as unidades vectoriais. Deste xeito, a presente Tese, "*Manycore Architectures and SIMD Optimizations for High Performance Computing*", aborda estas dúas dimensións ortogonais analizando arquitecturas *manycore* modernas e poñendo énfase no descubrimento de potenciais melloras de deseño en dous diferentes niveis da xerarquía arquitectónica, e desenvolvendo técnicas para a xeración de código vectorial eficiente dependente da plataforma.

Na primeira parte desta Tese investigamos as redes de interconexión (NoCs) pre-

sentes nas arquitecturas *manycore* modernas e o impacto do tráfico xerado polos directorios de coherencia caché distribuídos no desempeño destas arquitecturas. Puxemos o foco na Intel Mesh Interconnect introducida inicialmente no Xeon Phi x200 Knights Landing (KNL) e continuada nas subseguintes xeracións de Xeon Scalable. Cando comezamos a analizar o tráfico de coherencia na NoC do Knights Landing observamos un claro efecto no desempeño das aplicacións debido á afinidade entre os núcleos e os seus directorios caché (ou CHAs), é dicir, observamos un comportamento NUMA. Por esta razón, xeramos un mapa de correspondencia entre cada liña caché e o CHA asociado para revelar o mapeo físico en función do enderezo de memoria. Con este mapeo, desenvolvemos un algoritmo inspector-executor para caracterizar e optimizar o impacto da afinidade núcleo-a-CHA. Obtivemos resultados prometedores de modo que estendemos este traballo mediante a realización de enxeñaría inversa para recuperar as funcións encargadas de distribuír as liñas caché entre os diferentes CHAs. A idea de recuperar estas funcións era aliviar a sobrecarga introducida polo algoritmo inspector-executor en tempo de execución, de forma que tamén puidésemos realizar optimizacións en tempo de compilación. Con todo, a forma destas funcións de mapeo baseadas en portas XOR son pouco custosas de implementar en *hardware*, pero non en *software*. Este custo podería mitigarse se este mapeo presentase algún tipo de regularidade que se puidese explotar optimizando o código e a súa planificación, pero este non era o caso para o KNL dado que o número de *tiles* non era potencia de 2, o cal producía funcións de mapeo non lineais. A nosa avaliación demostra a importancia da afinidade dos datos en sistemas que integran directorios caché distribuídos, e a importancia e impacto do tráfico de coherencia caché. Tendo en conta que as arquitecturas *manycore* son consideradas como o futuro da arquitectura de computadores, é desexábel mellorar estes deseños evitando estes escollos na NoC empregando para este obxectivo un mapeo máis regular e predicible dos bloques de memoria para permitir aos programadores, particularmente no dominio da computación de altas prestacións, ter un maior control sobre o tráfico de coherencia. A aproximación seguida nesta parte da Tese puxo especial énfase no Intel Xeon Phi 7210, pero tamén é potencialmente aplicable aos procesadores Xeon Scalable xa que empregan a mesma tecnoloxía na NoC.

Nas mesmas liñas de traballo, construímos un modelo da arquitectura KNL en Tejas, que é un simulador arquitectónico cunha precisión a nivel de ciclo. Con este

modelo, fomos capaces de realizar unha análise do comportamento da complexa rede de interconexión. Primeiro validamos o noso modelo fronte ao *hardware*, atendendo ás limitacións na tradución das instrucións do simulador (Tejas non soporta todo o conxunto de instrucións x86). Tamén presentamos un caso de estudo analizando o comportamento de baixo nivel da rede de interconexión e das optimizacións descritas anteriormente para mellorar a localidade e afinidade entre datos e unidades de procesamento, así como o mapeo dos fíos de execución en aplicacións paralelas. A nosa avaliación confirma a redución do tráfico de coherencia na rede e a redución no número de colisións.

Nunha dimensión ortogonal, na segunda parte da Tese investigamos diferentes optimizacións vectoriais para computacións que empregan estruturas de datos irregulares como as matrices dispersas. Puxemos énfase en dous tipos concretos de optimizacións: o empaquetado de posicións de memoria aleatorias no mesmo rexistro vectorial e a fusión de reducións independentes. Para esta parte do traballo precisamos dunha ferramenta para a automatización dun gran volume de experimentos, dado que tiñamos que compilar os códigos con diferentes configuracións de parámetros, avaliar o rendemento empregando contadores *hardware* e analizar estes valores. Deste xeito desenvolvemos MARTA (Multi-configuration Assembly pRofiler and Toolkit for performance Analysis), unha ferramenta deseñada para mellorar a produtividade para este tipo de experimentos que requiren *micro-benchmarking* empregando diferentes configuracións con múltiples parámetros. MARTA, ademais de caracterizar, tamén permite extraer coñecemento dos datos xerados empregando técnicas de *data mining* e *machine learning*. MARTA permite caracterizar o desempeño de acordo a unha serie de dimensións de interese. Así avaliamos o rendemento de MARTA en varios casos de uso como outra alternativa para calquera experimento de caracterización do desempeño, aínda que orixinalmente foi deseñada para construír os modelos de custo das optimizacións vectoriais.

A primeira optimización foi o empaquetado de posicións de memoria aleatorias en rexistros vectoriais para arquitecturas x86. O obxectivo principal era xerar implementacións eficientes que empregasen instrucións dun conxunto concreto. Primeiro definimos un espazo de exploración atendendo ao conxunto de instrucións a empregar e unhas clases de equivalencia de acordo á contigüidade e tipo de datos das posicións de memoria a empaquetar. Con esta información construímos MRKVS

(Mega-Random Kernel Vector SMT), un sistema baseado en *Satisfiability Modulo Theories* (SMT) para xerar conxuntos de instrucións ou candidatos para cada clase de equivalencia. Facendo uso de MARTA, para cada plataforma xeramos un modelo de custo individual. Os candidatos empregados para as clases de equivalencia definidas dependen da arquitectura obxectivo. Avaliamos o desempeño dos candidatos xerados polo sistema fronte á instrución *gather* equivalente obtendo, para a maioría dos casos, amplas marxes de mellora en diferentes arquitecturas.

Por último, desenvolvemos MACVETH (Multi-Architectural C-VEcTorizer for HPC applications), un compilador fonte-a-fonte baseado en Clang que inclúe o empaquetado de posicións aleatorias de memoria en rexistros vectoriais e a fusión de reducións independentes. Para este propósito desenvolvemos diferentes algoritmos para o empaquetado de reducións independentes nun programa, sintetizando código vectorial eficiente. Tamén integramos unha estratexia para vectorizar aquelas operacións de redución que non poden ser empaquetadas xuntas (denominadas como reducións orfas). Avaliamos o desempeño da nosa aproximación empregando diferentes patróns e formas de bucles. Os resultados son prometedores e confirman potenciais liñas de optimización en códigos que presentan patróns irregulares.

Obxectivos e Metodoloxía de Traballo

Os obxectivos principais desta Tese descríbense a continuación, incluíndo os subobxectivos clave.

1. Análise e modelaxe do procesador Intel Xeon Phi x200 (Knights Landing, KNL).
    - Análise e caracterización da arquitectura do núcleo, o directorio caché distribuído e a rede de interconexión.
    - Implementación do modelo arquitectónico do Intel Knights Landing como unha extensión no simulador Tejas.
    - Validación experimental da precisión do simulador executando diferentes aplicacións.

2. Optimización do tráfico de coherencia caché en arquitecturas *manycore*.

    - Aproximación para descubrir a disposición física dos compoñentes.
    - Enxeñaría inversa das funcións *hash* de memoria na arquitectura KNL.
    - Aproximacións en tempo de execución e compilación para optimizar o tráfico de coherencia caché na rede de interconexión.
    - Avaliación das aproximacións estática e dinámica propostas.

3. Desenvolvemento dunha ferramenta de *profiling* e análise do rendemento deseñada especificamente para experimentos que requiren a configuración de moitos parámetros.

    - Automatización da compilación, execución e análise dado calquera programa ou benchmark e os parámetros de interese, e.g., o tamaño da matriz, o incremento nun bucle, etc.
    - Técnicas de minería de datos para extraer coñecemento a partir dos experimentos e as dimensións de interese, i.e., cuantificar a influencia das variables.
    - Ferramenta compatible con calquera tipo de aplicación, deseñada para mellorar a produtividade, calidade e reproducibilidade dos experimentos.

4. Síntese de código vectorial x86 eficiente para o empaquetado de posicións de memoria aleatorias en rexistros vectoriais e a fusión de reducións.

    - Xeración de combinacións de empaquetado de datos aleatorios en rexistros vectoriais empregando as instrucións dispoñíbeis dado un conxunto concreto (ISA) e baseado nun modelo SMT.
    - Construción dun modelo de custo baseado no rendemento empírico destas combinacións para cada plataforma concreta.
    - Desenvolvemento dun compilador fonte-a-fonte para a síntese de código vectorial eficiente baseado na caracterización individual de cada plataforma.
    - Avaliación empregando diferentes aplicacións.

Contribucións Principais

As contribucións orixinais derivadas desta Tese descríbense a continuación:

- Desenvolvemento dunha extensión para o simulador Tejas para a exploración da arquitectura KNL ou outras similares que empreguen un sistema distribuído de coherencia caché. Esta extensión permite a análise do tráfico de coherencia caché nas redes de interconexión [44, 46].

- Enxeñaría inversa da arquitectura Intel Knights Landing para revelar a disposición física dos compoñentes. Baseadas neste modelo, desenvolvemos diferentes técnicas de optimización do tráfico de coherencia caché, a afinidade *thread-to-core* e a planificación de diferentes tarefas na rede, aproveitando as características únicas dun procesador concreto derivadas das variacións nos procesos litográficos [45].

- Revelación da función pseudo-aleatoria de mapeo dos enderezos físicos dos bloques de memoria sobre os compoñentes do directorio distribuído no KNL. Aproveitando esta información, estudamos diferentes optimizacións para mellorar as latencias de memoria mediante a optimización do tráfico de coherencia caché. Estas melloras no rendemento da memoria non se traducen directamente en melloras globais no rendemento debido ás inherentes sobrecargas derivadas da complexidade computacional no mapeo das funcións [68].

- Desenvolvemento de MARTA (Multi-configuration Assembly pRofiler and Toolkit for performance Analysis), unha ferramenta de *profiling* e análise do rendemento deseñada para incrementar a produtividade [50, 52]. Esta ferramenta non é unha substituta de ningunha outra ferramenta, pero a súa maior e orixinal contribución é a énfase na automatización, mellorando a produtividade e calidade dos resultados. Nunha dimensión ortogonal, esta ferramenta inclúe un módulo para a análise do rendemento empregando técnicas de minería de datos e aprendizaxe máquina.

- Desenvolvemento de MRKVS (Mega-Random Kernel Vector SMT), un sistema baseado en SMT para a xeración de modelos de empaquetado de datos aleatorios dado un conxunto de instrucións (ISA). A partir destes modelos,

construímos un modelo de custo que habilita o empaquetado de datos aleatorios en rexistros vectoriais.

- Desenvolvemento de MACVETH (Multi-Architectural C-VEcTorizer for HPC applications) [51], un compilador fonte-a-fonte C para a síntese de código vectorial eficiente dadas unhas rexións de interese no programa que conteñan patróns de acceso a memoria irregulares. Este compilador inclúe o modelo de custo construído con MRKVS, ademais de heurísticas para a vectorización e fusión de reducións independentes.

Traballo Futuro

A continuación describimos unha serie de potenciais liñas de traballo futuras:

- A arquitectura de Intel KNL abandonouse, pero a súa rede interconexión está presente nos novos procesadores Intel Xeon Scalable, que tamén integran directorios caché distribuídos empregando *snoop filters* (equivalentes aos CHA no KNL) e posúen unha malla 2D de interconexión. A nosa aproximación para mellorar a localidade dos datos poderíase trasladar a estas novas arquitecturas. As investigacións de McCalpin [88] revelan unha disposición similar destas arquitecturas á dos procesadores Intel KNL.

- Seguindo a mesma liña de investigación, e partindo do noso modelo para o KNL, poderíase implementar unha nova extensión para os procesadores Intel Xeon Scalable no simulador Tejas.

- MARTA é unha ferramenta nova e, polo tanto, require un desenvolvemento continuo, propoñendo novas melloras e extensións. Deberíanse integrar novas opcións no *front-end* para interpretar mellor o ficheiro de configuración e/ou facer o formato máis flexíbel. O compoñente de análise podería automatizar tarefas para a selección de hiperparámetros nos algoritmos de minería de datos, evitando a intervención innecesaria do usuario. Do mesmo xeito, o sistema podería soportar outros tipos de análises e algoritmos, como as análises de calibrado (empregando regresión isotónica, por exemplo). Este tipo de análises é interesante para complementar o coñecemento extraído da árbore de decisión.

- MACVETH concibiuse como un compilador fonte-a-fonte para arquitecturas x86 que implementa o empaquetado de posicións de memoria aleatorias en rexistros vectoriais e a fusión de reducións. Avaliamos o desempeño destas optimizacións empregando unha ampla variedade de códigos irregulares obtendo resultados prometedores, pero requírese unha avaliación máis exhaustiva empregando outras arquitecturas (e.g., AMD, ARM) e outras extensións vectoriais máis recentes a maiores de AVX2. Do mesmo xeito, algunhas das optimizacións vectoriais incluídas neste compilador, como a vectorización de reducións orfas, non contribúen á mellora do rendemento da forma esperada. Esta optimización era teoricamente prometedora, pero nos códigos SpMV probados nos experimentos causaba certa degradación. A razón para este comportamento é a distancia en memoria entre esas reducións, que causa unha degradación na localidade caché e, consecuentemente, no desempeño xeral. Como futura liña de investigación, sería interesante determinar a distancia máxima permitida en cada plataforma para empaquetar de forma eficiente eses nodos. Así, MACVETH podería implementar mecanismos máis intelixentes para determinar de maneira automática este tipo de parámetros para cada plataforma.

Software Desenvolvido

As bibliotecas e ferramentas desenvolvidas nesta Tese están dispoñíbeis de forma pública:

- Tejas KNL. Implementación do modelo arquitectónico de Intel Knights Landing no simulador Tejas. Dispoñíbel en https://github.com/UDC-GAC/tejas_knl.
- papi_wrapper: Biblioteca baseada en macros C para simplificar o uso da biblioteca PAPI. Dispoñíbel en https://github.com/UDC-GAC/papi_wrapper.
- MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. Ferramenta desenvolvida para incrementar a produtividade e calidade dos experimentos que requiren micro-benchmarking e análise post hoc do rendemento. Dispoñíbel en https://github.com/UDC-GAC/MARTA.

- MRKVS: Mega-Random Kernel Vector SMT. Sistema baseado en Z3 [23] para a xeración de combinacións de instrucións para o empaquetado de datos aleatorios no mesmo rexistro vectorial. Dispoñíbel en https://github.com/UDC-GAC/MRKVS.

- MACVETH: Multi-Architectural C-VEcTorizer for HPC applications. Compilador en C fonte-a-fonte para a vectorización de reducións e accesos aleatorios a memoria. Dispoñíbel en https://github.com/UDC-GAC/MACVETH.

Publicacións derivadas da Tese

Publicacións en revistas internacionais

- S. Kommrusch, M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Optimizing Coherence Traffic in Manycore Processors Using Closed-Form Caching/Home Agent Mappings. *IEEE Access*, 9:28930–28945, 2021. doi: 10.1109/ACCESS.2021.3058280. JCR Q2 [68].

- M. Horro, G. Rodríguez, and J. Touriño. Simulating the Network Activity of Modern Manycores. *IEEE Access*, 7:81195–81210, 2019. doi: 10.1109/ACCESS.2019.2923855. JCR Q1 [46].

Conferencias internacionais

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MARTA: Multi-configuration Assembly pRofiler and Toolkit for performance Analysis. Enviado para publicación. 2022 [52].

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. MACVETH: Multi-Architectural C-VEcTorizer for HPC applications. Enviado para publicación. 2022 [51].

- M. Horro, M. T. Kandemir, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Effect of Distributed Directories in Mesh Interconnects. *Proceedings of the 56th*

Annual Design Automation Conference (DAC), páxinas 51:1–6, Las Vegas, NV, EE.UU., 2019. doi: 10.1145/3316781.3317808. Core A. GII-GRIN-SCIE Clase 1 [45].

Conferencias nacionais

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Toolkit para (Micro-) Benchmarking y Análisis de Características de Rendimiento en Kernels. *Actas XXXI Jornadas de Paralelismo (SARTECO)*, páxinas 303–312, Málaga, España, 2021 [50].

Outras publicacións

- M. Horro, L.-N. Pouchet, G. Rodríguez, and J. Touriño. Exploring SIMD Instructions for Packing Random Vector Operands in Modern x86 CPUs. *Proceedings of the 17th International Summer School on Advanced Computer Architecture and Compilation for High-Performance Embedded Systems (ACACES)*, páxinas 143–146, Fiuggi, Italia, 2021 [49].

- M. Horro, G. Rodríguez, J. Touriño, and M. T. Kandemir. Study of the Intel Knights Landing (KNL) Memory System Tradeoffs. *Proceedings of the 13th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, páxinas 1–4, Fiuggi, Italia, 2017 [44].

Alphabetical Index

Symbols

A

B

C

N

P

Q

R

S

T