Towards Automatic Code Generation for GPUs*

Javier Setoain[1], Christian Tenllado[1], Jose Ignacio Gómez[1], Manuel Arenaz[2], Manuel Prieto[1], and Juan Touriño[2]

[1]ArTeCS Group
Complutense University of Madrid, Spain
jsetoain@pdi.ucm.es
{tenllado,jigomez,mpmatias}@dacya.ucm.es

[2]Computer Architecture Group
University of A Coruña, Spain
{arenaz,juan}@udc.es

Abstract. Graphics Processing Units (GPUs) have become highly parallel and programmable systems used as commodity data-parallel coprocessors. Moreover manufacturers have developed new software interfaces that facilitate their use. Thus, new compilation strategies that enable automatic mapping of sequential code would very likely arise in the near future. To open this path, we need to define some performance metrics or heuristics that steer the mapping, as well as to develop new tools that support such metrics or heuristics. In this paper we sketch a new compiler intermediate representation built on top of the hierarchy of computational kernels provided by the XARK compiler framework. In addition, we outline several compilation strategies based on such intermediate representation and present several experiments aimed at analyzing the relative importance of the two principal factors that NVIDIA reports to influence the most on performance of their GPUs: occupancy and memory hierarchy usage. Our results show that taking decisions based only on the optimization of the memory system provide the best performance, even if some resources are underutilized.

1 Introduction

Graphics Processing Units (GPUs) have evolved from application-specific units for 3D scene rendering into programmable and highly parallel processors, that can satisfy extremely high computational requirements at low cost. Their numbers are impressive. Today's fastest GPUs can deliver a peak performance in the order of 500 Gflops [2], more than four times the performance of the fastest x86 quad-core processor [5]. This astonishing performance has captured the attention of many developers and researchers in different areas, who are using GPUs as commodity data-parallel coprocessors to speed up their own applications [6]. Manufacturers in turn, have developed new software interfaces that

* This research was supported by the Ministry of Education and Science of Spain and FEDER funds of the European Union (Projects TIN2007-67537-C03, TIN2005-5619, Ingenio 2010 Consolider CSD2007-20811 and the HIPEAC European Network of Excellence), and by the Galician Government (Projects PGIDIT05PXIC10504PN and PGIDIT06PXIB105228PR).

facilitate GPU programmability as general purpose parallel coprocessors. The most representative examples are NVIDIA's CUDA [1] and AMD's Brook+ [9].

This new scenario should redirect some of the efforts of the GPU research community from ad-hoc porting of applications, to the development of new compilation strategies that enable automatic mapping of sequential code. State-of-the-art tools for automatic recognition of program constructs, such as the XARK compiler framework [3, 4], can play an important role in this respect. XARK provides a high level hierarchical representation of the program with valuable semantic information for the GPU mapping process. However, it is still needed to define some performance metrics and heuristics in order to steer this mapping, and extend the compiler framework accordingly.

In this paper we perform several experiments aimed at analyzing the main factors behind GPU's performance in an attempt to define those heuristics. As a driven example we have used a real world algorithm [8] that exhibits some of the computing patterns present in many scientific and image processing applications.

The rest of the paper is organized as follows. In Section 2 we give an overview of some related work. Section 3 briefly describes the CUDA programming model and the main factors that influence on performance. Using an image processing application as a guide, Section 4 describes the XARK-based representation of the algorithm and illustrates how some different GPU implementations can be easily generated. In Section 5 we design some experiments to determine the relative influence of the different performance factors and extract our mapping guidelines. Finally, Section 6 presents the main conclusions of this paper and outlines the future work.

2 Related Work

Focusing on general purpose computing on GPUs (GPGPU), most of the research activity works towards finding efficient strategies for algorithm mapping on these platforms. Generally speaking, it involves developing new implementation strategies following a stream programming model, in which the available data parallelism is explicitly uncovered, so that it can be exploited by the hardware. This programming challenge has been studied by several researchers who have successfully ported a large number of scientific applications [6].

Regarding code generation for GPUs, most efforts have been focused on developing new software interfaces to build GPGPU applications with less programming effort. The most representative examples are NVIDIA's CUDA [1] and AMD's Brook+ [9]. Essentially, both of them provide kernel-style data parallel extensions to the standard C Language.

Finally, some groups are trying build a performance model for GPUs. Shane Ryoo et. al. [7] have studied the effect of loop unrolling and tiling on matrix multiplication codes, focusing on the impact of the occupancy factor on performance. This paper extends [7] by considering the interaction of occupancy and memory hierarchy usage on a more complex application.

3 Execution model and performance limiting factors

GPUs have evolved from application specific processors to highly parallel multicore architectures that exploit hardware multi-threading to maximize the utilization of computational resources. CUDA-enabled GPUs from NVIDIA organize those cores as SIMD multiprocessors and overall, they are able to execute thousands of concurrent threads.

Their memory hierarchies have also improved substantially and include different on-chip memories to exploit data locality. For instance, CUDA-enabled GPUs integrate software controlled scratchpad memories, which are shared among the cores of the SIMD multiprocessors, and read-only caches to speed up the access to texture and constant data.

Under the CUDA programming model, the GPU is view as a coprocessor that executes data-parallel kernel functions. The data-parallel threads are arranged into groups of up to 512 threads, called *CUDA blocks*. Threads within a block can cooperate with each other by (1) efficiently sharing data through the low latency shared memory that resides in the multiprocessor and (2) synchronizing their execution, for hazard-free shared memory accesses. Each multiprocessor can allocate multiple *blocks* concurrently.

For scheduling purposes, threads are arranged in smaller groups, known as *warps*, that executes the same instruction in a SIMD fashion. Memory accesses from threads within the same warp can be coalesced as long as they access groups of contiguously aligned memory elements. Coalesced memory accesses are desirable since memory access penalties are one of the ultimate performance factors.

Given that multiprocessor's resources are statically partitioned among the assigned threads, the actual maximum number of concurrent threads per multiprocessor is limited by the allocated resources. Thus, besides predefined limitations in the number of threads per multiprocessor and threads per *CUDA block*, actual numbers could be lower due to resources demands. Within these limits, the programmer specifies how many *blocks* and how many threads per *block* are assigned to the execution of a given data parallel function. *Blocks* are then dispatched to the multiprocessors until all of them are completed.

According to NVIDIA, one of the main concerns for GPU programmers should be occupancy maximization. Current NVIDIA GPUs support a maximum of 768 concurrent threads per multiprocessor, that must be distributed across equally sized blocks. Other than unveiling parallelism, resource distribution must be considered to achieve an acceptable occupancy. For instance, the available registers are statically assigned to the given threads, so that each thread acquires the same number of registers.

The other most significant factor affecting performance is the memory usage. Despite taking advantage of multithreading to mitigate the impact of the large memory accesses latencies, having hundreds of threads accessing simultaneously to the off chip DRAM poses a hard problem. This means that taking advantage of memory coalescing and exploiting on-chip memories becomes almost mandatory.

4 Code Generation Strategies for GPUs using XARK

Today's optimizing compilers represent program behavior by means of several graphs that capture information at the statement and/or at the basic block levels. Well-known examples are the data dependence graph and the dominance tree. In contrast, the XARK framework [3] builds a hierarchical representation of the code that decomposes a program into a set of mutually dependent *kernels* that capture the behavior of the code fragment. The kernels capture well-known program constructs such as inductions, reductions and recurrences, even in the presence of complex control flows. Thus, XARK provides valuable information about the computations carried out at runtime on scalar and non-scalar variables.

Based on this hierarchical representation we have built a Kernel Data Dependence Graph (K-DDG), which consists of a pair $< N, E >$ where the set of nodes N represents the kernels recognized by XARK, and the set of edges E captures the dependence relationships between these kernels. More formally, let K_1 and K_2 be two kernels that capture the computations of the sets of source code statements $s_1^1, \ldots, s_1^n$ and $s_2^1, \ldots, s_2^m$, respectively. The K-DDG contains an edge for each dependence $s_1^i \longrightarrow s_2^j$ between statements of different kernels K_1 and K_2.

For illustrative purposes, consider the pseudocode of an application of the hyperspectral image processing domain shown in Figure 1. On the left-hand side, the figure presents the source code, which consists of a loop nest with maximum depth 4. Two array variables *res* and *respos* store the results of the execution of the loop. The values assigned to *res* and *respos* are calculated from temporary computations (*dp*, *acc*, *max*, *maxpos* and *np*) carried out at the beginning of each loop iteration. On the right-hand side, the figure shows the K-DDG of the program. The nodes are depicted as ovals labeled with the program variable that stores the results of the computation of the kernel. Attached to each node, the type of kernel is shown: *induction* for induction variables (in particular, loop indexes); *reinitialized reduction* for a reduction operation (e.g., sum, maximum) computed in an inner loop that is reset to a constant value at each iteration of an outer loop; and *array assignment* for the computation of the value of all the entries of an array variable. In order to highlight the relationship between the source code and the K-DDG, the nodes of the K-DDG that capture temporary computations (kernels associated with *dp*, *acc*, *max*, *maxpos* and *np*) are linked to the corresponding source code fragment. The edges are depicted as arrows that capture depencences between kernels. For instance, there exist two dependences between the reinitialized reduction associated to the array variable *acc* and the scalar variable *max*: the first one appears from its use in the predicate $acc[bij3] > max$ of the if statement; and the second one appears from its use in the right-hand side of the assignment statement $max = acc[bij3]$.

In the following subsections we use this K-DDG in order to explore different mapping alternatives on a GPU.

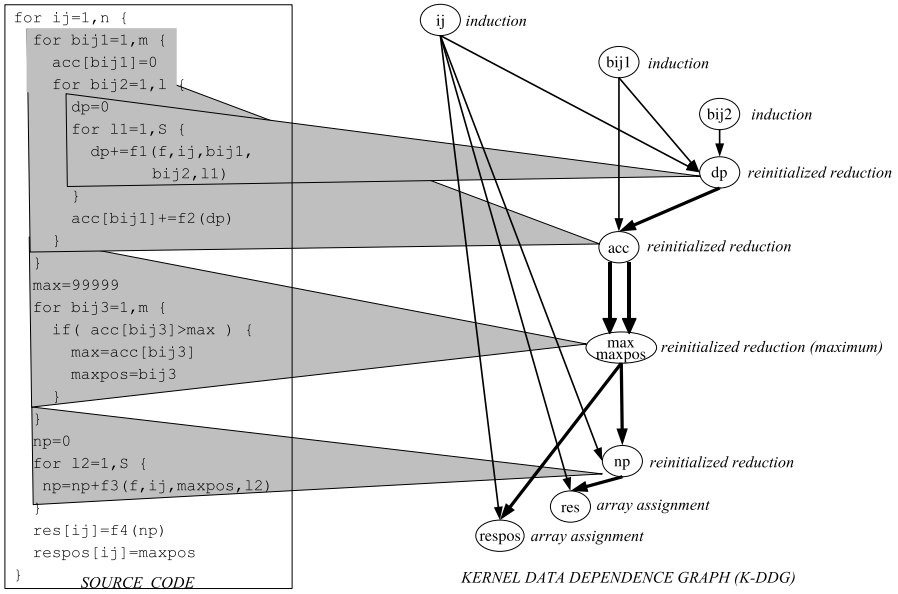

Fig. 1. Kernel Data Dependence Graph (K-DDG) of the pseudocode of an application from the hyperspectral image processing domain.

4.1 Exploiting Processor Occupancy

In order to study the influence of processor occupancy in the overall performance, we have experimented with two well-known code transformations: loop fission and loop unrolling. *Loop fission* breaks a loop into multiple loops over the same index range but each taking a part of the loop body. Loop fission is an effective way to reduce the size and the requirements of the GPU programs. Consider the example of Figure 1. The outermost loop $\texttt{for}_{ij}$ computes two array assignment kernels (*res* and *respos*) and thus it can be fully parallelized. However, as the code fragment is large, the number of registers required is too high to reach 100% occupancy. As a result, the compilation strategy may consist of applying loop fission to $\texttt{for}_{ij}$ in order to reduce the size of the GPU program. For instance, the loop body can be broken by the dependences between *acc* and *max* as they are loop-independent dependences. However, this strategy forces to expand the *acc* array, as it must now store the results for all the ij iterations. Moreover, we must now run sequentially two GPU kernels instead of one, which could means an important performance penalty if they are too light. Finally, loop fission may also lead to a loss of locality, further restricting the potential performance gains achieved by maximizing occupancy.

Loop unrolling is another optimizing transformation that impacts on resource usage. It consists of replicating the loop body to build a single sequence of

instructions and reduce the number of loop iterations. Thus, the pressure on the registers is increased and the number of concurrent threads is limited. According to [7], loop unrolling (in the context of matrix multiplication) is still worthy due to the increase of ILP. However, in many occasions the loss in occupancy could not be compensated by the increase in ILP.

Moreover, the number of threads per *block* further restricts the total occupancy. Given the limit of 768 threads per multiprocessor, using maximum size *CUDA blocks* of 512 threads would always lead to underutilized resources. On the other extreme, too small *blocks* (less than 64 threads) are neither advisable due to the warp-scheduling explained above.

4.2 Exploiting Memory Hierarchy

In many situations it is not possible to guarantee alignment access in the whole GPU kernel program. The example code in Figure 1 exhibits this problem. As described above, the iterations of the outermost loop $\texttt{for}_{ij}$ can be computed concurrently by different threads. Hence, if memory alignment is guarantee for a given value of the loop index $bij1$, then the remaining iterations of $\texttt{for}_{bij1}$ will perform misaligned memory accesses. We have performed several experiments aiming at evaluating the impact of this misalignment problem.

Texture Cache. GPUs can access the off chip DRAM through read-only texture caches: the programmer only needs to bind a *texture* to the given memory area. On cache misses the texture cache controller fetches a whole cache line. This way, misaligned memory accesses are avoided but at the expense of potentially increasing bandwidth demands. If spatial locality exists within the threads of the same *block*, texture fetching captures it and reduces the pressure on the memory system, but if not, it increases bandwidth demands.

Shared Memory. The main advantage of this software controlled memory is its low latency, but the actual benefits of using it will depend on how much reuse is possible and what are the overheads caused by (1) fetching the data from the off-chip RAM (misaligned problems still could exists) and (2) synchronizing thread execution for hazard free shared memory accesses.

5 Experimental Results

In this section we present the experimental results obtained from applying the ideas presented in Section 4 to our driving example (see Figure 1). All the experiments have been carried out on a NVIDIA GeForce 8800 GTX (G80).

To study the influence of occupancy we have run two versions of our example code varying the number of threads per *CUDA block* from 16 to 512 (see Figure 2(a)). In the first version, a single kernel programs encapsulates the whole code (we call it the *Single* version), whereas in the second one, we have applied *loop fission* as described in Section 4.1 (we call it the *Split* version). Surprisingly,

even if performance differences are low (less than 10%), the best results are obtained for 16 and 32 threads per *block*, where occupancy drops as low as 33%. Apparently, other factors are limiting performance much more than occupancy.

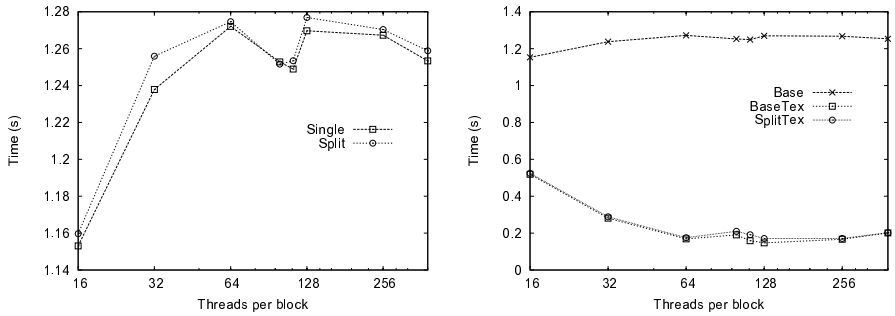

Fig. 2. Influence of the number of threads per *CUDA block* and the GPU program size on performance either without (a) or with (b) texture fetching.

Figure 2(b) confirms this intuition. Once memory access is enhanced by using texture fetching, which improves locality and removes misalignment problems, the behavior become closer to our expectations. The optimal performance is achieved using 128 threads per *CUDA block*, which reaches 83% occupancy, the maximum attainable in this example. Finally, it is important to note that the *Split* version never outperforms the *Single* one in any of the two scenarios. With texture fetching the performance gap becomes lower. However, in both cases the benefits of reducing the pressure on resources is lost due to the overheads associated with launching more kernel programs.

We have also studied the impact of unrolling the innermost loop ($l1$ loop in Figure 1) on performance. To keep under control the number of registers, we prevent the compiler from using more than 12, 16 and 20 registers per thread[1]. Without using textures (see Figure 3(a)), the execution time scarcely improves with unrolling in both versions: it has no impact on the *Split* version, whereas for the *Single* one, it achieves a small speedup when registers are not limited.

Figure 3(b) shows the result of the same set of experiments but accessing the DRAM through texture fetching. Again, the execution time significantly improves over non-texture counterpart. Furthermore, we also observed that (1) unrolling never has positive effects on the execution time of our driving example and (2) limiting the number of registers to the compiler slightly improves the *Split* implementation.

Since the performance of both versions are similar, for the rest of this Section we only consider the *Single* implementation.

[1] None of the program versions ever requires more than 20 registers per thread, so this results in non register usage restriction

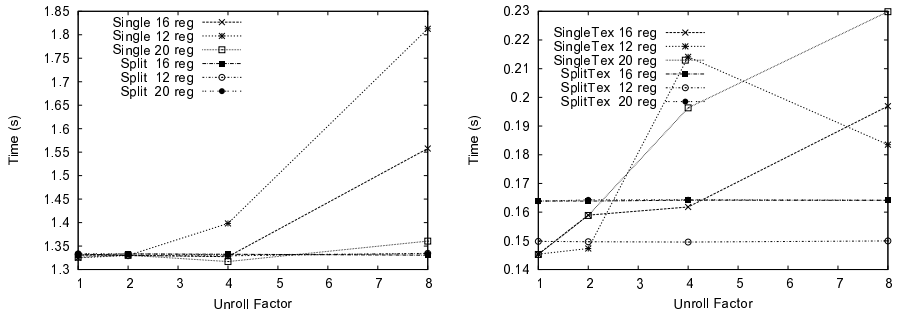

Fig. 3. Influence of the unroll factor and number of registers available on performance either without (a) or with (b) texture fetching. These curves were obtained using 128 threads per *CUDA block*.

In the next experiments we analyze the effect of exploiting shared memory. The loop nests that contribute most to the total execution time are $l1$ and $l2$, labeled by XARK as *reinitialized reduction* (see Figure 1)). We need to tile these loops so that all the elements accessed in a tile fit in the shared memory. Figure 4 compares the execution time of six different versions of the *Single* code. *Base*, *BaseTex* and *BaseSh* stand for the original *Single* code, the original code with texture fetching and the original code with shared memory respectively.

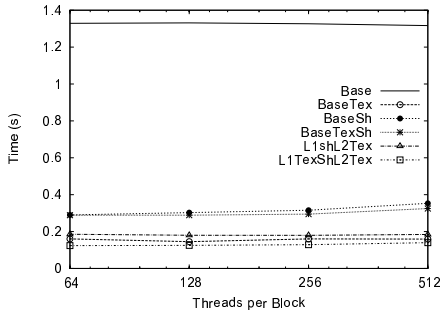

Fig. 4. Effect of using Shared memory and its synergy with the use of texture accesses.

Using texture fetching in both loops (*BaseTex*) outperforms either the shared memory counterpart (*BaseSh*) or the combined version (*BaseTexSh*), whereas *BaseTexSh* only improves *BaseSh* by a small factor. There are two reason that explains this behavior:

1. The nature of the $l2$ loop causes a lot of bank conflicts in the access to the shared memory. Whereas the access function to f is known at compile time

in the first loop, it is modified at run time (through variable *maxpos*) in the second one. The actual addresses accessed in this case by the different threads will frequently match, causing bank conflicts in the shared memory.
2. Texture fetching on the $l1$ loop is slightly more efficient than using shared memory (compare *BaseTex* and *L1ShL2Tex* curves) due to the overhead introduced by the code that fetches the data from the off-chip DRAM to the shared memory (it exhibits misalignment problems).

Overall, the best performance is achieved when texture fetching is combined with shared memory only in the $l1$ loop (*L1TexShL2Tex*). This reinforces the observation that using texture fetching significantly improves performance by improving data locality and eliminating the problems introduced by the misaligned accesses to the off chip DRAM. In addition, it shows that using shared memory does not always improve performance and it must be employed carefully. Essentially, codes shall fulfill two conditions to effectively exploit it: 1) they must exhibit enough data reuse and 2) there arithmetic density should be high enough. This way, the overhead introduced by intra-block thread synchronization and bank conflict accesses can be compensated. Finally, a very interesting finding is the synergistic effect of using texture fetching to reduce alignment problems in the code that fetches the data to the shared memory.

6 Conclusions

In this paper we have presented a comparative analysis of the factors that influence algorithm performance when mapped on modern GPUs, as a first step to automatize code generation for these platforms. Although the occupancy factor is reported by NVIDIA as being essential for performance, the overhead introduced by misalignment accesses to the off-chip memory happen to be one of the most dominant factors in codes that exhibit this problem.

In this scenario, occupancy should only be considered once the alignment problems have been mitigated. Texture fetching can ease these problems is spatial locality is high enough. On the contrary, shared memory does not always improves performance. We do recommend to avoid shared memory when we cannot guaranty at compile time that accesses to this memory are bank conflict-free. In those cases, texture fetching is a more robust alternative that usually provides better results. Furthermore, when accesses to the shared memory are bank conflict-free combining texture fetching and shared memory could be beneficial since texture fetching reduces the overhead introduced by misalignment accesses to the off-chip

Finally, we have proposed the use of the XARK compiler as a kernel-style parallelism uncovering tool, showing how its *computational kernel* representation can be used to extract most of the information needed for the algorithm mapping.

As future work we plan to obtain efficient techniques for mapping thread identifiers to parallel loop iterations using the XARK representation as input.

This process should give us enough information to decide which code transformations are needed to exploit locality and minimize the probability of bank conflicts in shared memory.

References

1. NVIDIA CUDA programming guide. Available online at: http://www.nvidia.com/object/cuda_develop.html.
2. NVIDIA Tesla. GPU computing technical brief. Available on-line at: http://www.nvidia.com/object/tesla_computing_solutions.html, May 2007.
3. Manuel Arenaz, Juan Touriño, and Ramón Doallo. Xark: An extensible framework for automatic recognition of computational kernels. *Accepted for Publication on the ACM Transactions on Programming Languages and Systems (TOPLAS).*
4. Manuel Arenaz, Juan Tourio, and Ramon Doallo. Program behavior characterization through advanced kernel recognition. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *EuroPar*, volume 4641 of *Lecture Notes in Computer Science*, pages 237–247. Springer, 2007.
5. Nebojsa Novakovic. Harpertown benchmarks show a monster in the making. http://www.theinquirer.net/en/inquirer/news/2007/09/18/harpertown-benchmarks-show-a-monster-in-the-making, September 2007.
6. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
7. Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.
8. Javier Setoain, Manuel Prieto, Christian Tenllado, Antonio Plaza, and Francisco Tirado. Parallel Morphological Endmember Extraction Using Commodity Graphics Hardware. *IEEE Geoscience and Remote Sensing Letters, vol. 4, issue 3, pp. 441-445*, 4:441–445, July 2007.
9. AMD Whitepaper. AMD stream computing software stack. Available online at http://ati.amd.com/technology/streamcomputing/index.html.