Writing self-adaptive codes for heterogeneous systems

Jorge F. Fabeiro, Diego Andrade, Basilio B. Fraguela, and Ramón Doallo

Depto. de Electrónica e Sistemas, Universidade da Coruña, Spain
{jorge.fernandez.fabeiro,diego.andrade,basilio.fraguela,doallo}@udc.es

Abstract. Heterogeneous systems are becoming increasingly common. Relatedly, the popularity of OpenCL is growing, as it provides a unified mean to program a wide variety of devices including GPUs or multicore CPUs. More recently, the Heterogeneous Programming Library (HPL) targets the same variety of systems as OpenCL, intending to improve their programmability. The main drawback of such unified approaches is the lack of performance portability, as codes written using OpenCL or HPL may obtain a good performance in a given device but a poor performance in a different one. HPL allows to generate different versions of kernels at run-time by combining C++ and the HPL embedded language. This paper explores the development of self-adaptive kernels that exploit this characteristic so that their code depends on configuration parameters that are tuned using a genetic algorithm through an iterative optimization process. The results show that these self-adaptive kernels are faster than those generated by hand following heuristics.

1 Introduction

One of the most important problems that hamper the wider use of heterogeneous systems is the current poor portability of the codes for these devices. The truly portable programming of heterogeneous system needs: (1) a unified programming language for any kind of device and, (2) a method to achieve performance portability. OpenCL [1] solves the first challenge as it enables the programming of a wide variety of devices. The second requirement, performance portability, has been widely addressed in the bibliography. For example, the framework [2] separates functionality from implementation details using specialized functions that allow to explore a great variety of implementations and to select the optimal one for a certain platform. VForce [3] provides performance portability in a transparent way across different kinds of accelerators to programs written in a domain-specific language focused on image and signal processing.

Performance portability can also be achieved through iterative processes. For example, [4] uses iterative compilation to select the optimal parameters for GPU codes according to a set of pre-defined, parametrized templates for linear algebra problems. An auto-tuning approach that selects the best execution plan for the SkePU skeleton programming framework in multi-GPU systems based on predictions is presented in [5]. The PARTANS framework [6], which is specifically designed to express stencil computations in multi-GPU systems, includes autotuning mechanisms to optimize this kind of computations.

Focusing on OpenCL, uCLbench [7] characterizes the properties of the device and the OpenCL implementation where the code is intended to run, seeking to guide programmers in the hand-tuning of their codes. The main changes required to port the performance of OpenCL codes that have been tuned for GPUs to CPUs are discussed in [8][9]. GLOpenCL [10] is a development framework consisting of a compiler and a runtime library that supports OpenCL on different types of multicores. OCLoptimizer [11] searches optimal unroll factors for OpenCL kernels based on compiler directives and a configuration file. Finally, Dolbeau et al [12] discuss the performance that the same OpenCL code achieves on different platforms. They use the CAPS compiler to generate autotuned OpenCL code.

The Heterogeneous Programming Library (HPL) [13] is a C++ framework that improves the programmability of heterogeneous systems by combining special data types and an embedded language to write kernels, which express the parallelized computations to run in the devices. HPL is a unified approach for programming heterogeneous systems as it uses as backend OpenCL, so that its kernels can run on any device. It also provides appropriate tools to provide performance portability, as the combination of its embedded language and C++ to write the kernels enables run-time code generation (RTCG), which can be used to write self-adaptive generic kernels. While other tools enable RTCG using similar mechanisms [14][15], they only target regular CPUs, and therefore they have sought other purposes. This way, this paper explores the development of kernels with portable performance by combining C++ and the HPL embedded language to generate parametrized generic kernels. The configuration parameters of each kernel change certain aspects of how its code is optimized, and they are adjusted using a genetic algorithm through an iterative process. The performance of the kernel generated using each combination of values of its parameters is evaluated by executing the code. The configuration parameters select the optimal unroll factors for some loops, the optimal granularity for the work performed by each instance of the kernel, the base version of the algorithm used, and which data structures are stored in local memory. The performance results, focused on a matrix product code, show that our approach generates kernels that can be up to 4.67 times faster than kernels generated following heuristics.

The rest of the paper is organized as follows. Section 2 briefly introduces the HPL library. Section 3 explains how RTCG can be used in HPL to write parametrized generic kernels. Section 4 explains the method derived to select the optimal values for the configuration parameters of the kernel using iterative optimization. Section 5 shows the experimental results and Section 6 concludes.

2 The Heterogeneous Programming Library

The Heterogeneous Programming Library (HPL), which is publicly available at `http://hpl.des.udc.es`, intends to improve the programmability of heterogeneous systems while providing portability through an approach where the computational kernels that exploit heterogeneous parallelism are written in a language embedded in C++. This characteristic enables run-time code generation (RTCG), which is a powerful tool to provide performance portability, as we will see through this paper. HPL provides portability because OpenCL is the intermediate representation (IR) it currently generates, thus this library targets the same range of devices supported by OpenCL.

The HPL library supports the same programming model as CUDA and OpenCL. Its hardware model is composed by a host equipped with a standard CPU and memory, with a number of computing devices attached. The host runs the sequential parts of the code, while the devices run the parallel parts. Each device has processors that execute SPMD parallel code on data present in the memory of their device. As in OpenCL or CUDA, we can create groups of threads that can be synchronized through barriers and share a small scratchpad memory.

The memory model distinguishes the same kinds memory as OpenCL (global, local, constant and private) and with the same properties. As kernels can only work with data available in the devices, data must be transferred between host and devices, but this process is totally automated by the library.

Several instances of each kernel, or work-items using OpenCL terminology, can be executed in parallel, each instance being univocally identified. The number of instances of the kernels and their identifiers are defined by a global domain of non-negative integers with up to 3 dimensions. This way, instances are identified inside this domain with tuples of global ids. In turn, these instances can be associated in groups. With this purpose, we can define local domains as equal portions of the global domain. Instances are identified inside its group using tuples of local ids. Now, Section 2.1 explains how to program using HPL.

2.1 Programming using HPL

The library provides three main components to the programmers:

- A template class `Array` to define both the variables to be transferred between the host and the devices, and the variables that are local to the kernels.
- The kernels, which are functions written in a language embedded in C++. This embedded language is an API in C++ consisting of data types, functions, macros and predefined variables.
- An API that will be used by the code to inspect the devices available in a given platform and to order the execution of the kernels.

All the kernel variables must have type `Array<type, n [, memFlag]>`, which represents an `n`-dimensional array of elements of a C++ `type`, or a scalar for `n`=0. Scalars and vectors can also be defined with special data types like `Int`, `Float`, `Int4`, `Float8`, etc. The optional `memFlag` can specify one of the kinds of memory supported (`Global`, `Local`, `Constant` or `Private`). The arrays passed as parameters to the kernels must be declared in the host using the same type. These variables are initially stored in the host memory, but when they are used as

Listing 1.1: SAXPY HPL code

```
void saxpy(Array<float,1> y, Array<float,1> x, Float a) {
    y[idx] = a * x[idx] + y[idx];
}

int main(int argc, char *argv) {
  Float a;
  Array<float, 1> x(1000), y(1000);
  //x, y and a are filled in with data (not shown)
  eval(saxpy).global(1000).local(10)(y, x, a);
}
```

kernel parameters they are automatically transferred to the device. The outputs are also automatically transferred to the host when needed.

HPL kernels also require that their control flow structures are written using special keywords. The embedded language uses the same constructs as C++ but their name finishes with an underscore (`if_`, `for_`, ...). Also, the arguments to for loops are separated by commas instead of semicolons. The library provides an API based on predefined variables to obtain the global, local and group identifiers as well as the sizes of the domains and numbers of groups. For example, `idx` provides the global id of the first dimension, while `szx` provides the globalsize of that dimension. If we add the `l` prefix to these keywords we obtain their local counterparts and if we replace the letter `x` with `y` or `z`, we obtain the same values for the second and the third dimensions respectively.

Kernels are written as regular functions or functors that use these elements and whose parameters are passed by value if they are scalars, and by reference otherwise. The `saxpy` routine in Listing 1.1 implements using this language the SAXPY (Single-precision real Alpha X Plus Y) vector BLAS routine, which computes $Y = a \times X + Y$. In this kernel, each instance `idx` computes a different position of the result `y[idx]`.

Regarding the host interface, its most important component is the function `eval`, which requests the execution of the kernel `f` with the syntax `eval(f)(arg1, arg2, ...)`. The execution of the kernel can be parametrized by inserting specifications, in the form of methods, between `eval` and the argument list. For example, the global and the local sizes can be specified using methods called global and local respectively. This way, the `saxpy` routine is invoked in Listing 1.1 with a global domain of 1000 elements and a local domain of 10 elements.

3 Performance portability in HPL

HPL generates the internal representation (IR) of its kernels by running them as regular code in the host when an `eval` requests their execution for the first time. Subsequent requests just reuse the IR generated the first time, which is stored in an internal cache, unless this cache is erased in order to force the regeneration of the IR. The HPL macros and data types capture all the expressions in which they are involved during the execution of the kernel in the host, allowing the runtime to generate the associated IR. However, regular C++ sentences found within the kernel are simply executed and they do not appear in the resulting IR. This characteristic enables RTCG, which can be used, for example, to choose between different versions of the same code, or to parametrize the generation of code. The method proposed in this paper combines RTCG and generic kernels to generate different versions of the same kernel based on different input parameters. In this context, generic kernels are those written for generic values of some parameter, such as the granularity, which can be adjusted at run-time.

First, we describe the strategy we have followed to parameterize the kernels. We have defined the HPL kernels using functors, so that for each kernel we define a class with the name of the kernel that defines the `operator()`. The arguments and the body of this method are the arguments and the body of the kernel, respectively. The parameters that will be used to parametrize the kernel at runtime, are defined as properties of this class, thus, they can be accessed from the `operator()` method. Besides, they can be set from the host before the generation of the kernel code is initiated by an `eval` invocation.

Based on a set of parameters, we have used RTCG and generic kernels to generate codes that at the same time: (1) apply the unrolling technique to one or several loops using a given unroll factor, (2) select the best granularity of the computation performed by each instance of the kernel, (3) select the most suitable variant of an algorithm depending on the device that will be used and (4) decide which data structures are stored in local memory. The methods used to introduce these features in the kernels are now explained in turn.

`Unrolling:` Loop unrolling is a popular optimization technique whose main benefits are that it unveils instruction level parallelism, minimizes branch penalty and reduces the number of control instructions executed. Loop unrolling using arbitrary unroll factors can be introduced in HPL kernels using RTCG. The C++ code will be used in conjunction with the embedded language to generate the unrolled loops. Let us see an example starting from the matrix-vector product (MxV) code shown in Listing 1.2. This code defines the HPL kernel in lines 2-6. Each instance of the kernel processes one row from the input matrix, thus a single loop is required to multiply each element of the row by the corresponding element of the input vector.

Listing 1.3 shows an unrolled version of the kernel. The loop between lines 4-7 is an unrolled version of the original loop, thus, its stride is now the unroll factor (`uf`). The body of the loop is replicated `uf` times by a native C++ loop (lines 5-6). As the number of iterations of the loop `N` may not be a multiple of `uf`, to prevent out of range array accesses, the loop limit is `N-uf`. If there are some iterations left after that loop, they are processed without unrolling by the code in lines 8-9. The value for the unroll factor is passed to the kernel from the main procedure by setting the appropriate attribute of the class that defines the kernel (line 15).

`Granurality:` HPL creates one instance (or thread in HPL terminology) of the kernel for each point of the global domain. The optimal amount of work performed by each thread must be tuned for each platform in order to maximize

Listing 1.2: MxV code: original version

```
class MxV {
  void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y) {
    Int k;
    for_(k=0, k<N, k++)
      y[idx] += (a[idx][k] * x[k]);
  }
};
int main(...) {
  //Declare and initialize ax,xv and yv Arrays
  MxV matvec
  eval(matvec).global(M)(av, xv, yv);
}
```

Listing 1.3: MxV code: unrolled version

```
class MxV { //Other portions of the class have been elided
  void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y) {
    Int k;
    for_(k=0, k <= (N - uf), k += uf) {
      for(aux=0; aux<uf; aux++)
        y[idx] += (a[idx][k+aux] * x[k+aux]);
    }
    for_(k,k<N,k++)
      y[idx] += (a[idx][k] * x[k]);
  }
}
int main(...) {
  ...
  MxV matvec
  matvec.set_uf(unrolling_factor);
  eval(matvec).global(M)(av, xv, yv);
}
```

the performance. For example, CPUs tend to be more effective using threads with larger workloads than GPUs. It is interesting to be able to tune that granularity at run-time depending on the type of device we are using. We can do that in HPL by changing the number of points in the global domain. For example, in our MxV code, the number of threads created is equal to the number of rows of the input matrix, thus, each thread processes one row of this matrix. If we reduce the number of threads, each thread should process several rows from the input matrix. This modification requires that the code is rewritten for a generic grain size, the grain size being in this case the number of rows of the input matrix processed by each thread. In our proposal, the rows are distributed using a block-cyclic policy, thus, grains of `bszx` rows are assigned cyclically to the threads available. The optimal value of `bszx` is found for each device. In the MxV code, this block size will not have a big influence in the performance, but in other problems some values of `bsz`s may benefit locality or coalescing, so, they will have a big impact in the performance.

In order to implement this distribution of the rows, the MxV kernel code must be changed to add two outer loops that process the blocks of `bszx` rows assigned

Listing 1.4: MxV code: auto-adjustable granularity version

```
1  class MxV { //Other portions of the class have been elided
2    void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y) {
3      Int ii, i, ilim, k;
4      for_(ii = idx*bszx, ii < M, ii += szx*bszx)
5        for_(i = ii,i < min(xx+bszx, M), i++)
6          for_(k = 0, k < N, k++)
7            y[i] += a[i][k] * x[k];
8    }
9  }
10 int main(...) {
11   ...
12   int szx = <# threads of the global domain>;
13   int bszx = <block size>;
14   ...
15   eval(matvec).device(dev).global(szx)(av, xv, yv);
16 }
```

Listing 1.5: MxV code: algorithm version selection

```
class MxV { //Other portions of the class have been elided
  void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y) {
    if (device==CPU) {
    //Version better suited to CPUs
    } else {
    //Version better suited to other devices
    }
  }
}
```

to each thread. Loop headers in lines 4-5 of Code 1.4 select the appropriate set of rows to be processed by each thread following the block-cyclic policy. The resulting kernel does not use RTCG but it is written in a generic way, so that if different values are provided for the size of the global domain and the block size, the granularity of the work performed by each thread is automatically adjusted at run-time.

`Algorithm selection`: The type of device used for a kernel execution is known at run-time. HPL can use this information to choose between different versions of the same algorithm, or portions of the algorithm, using RTCG. For example, a version that exploits local memory is good for GPUs but it may introduce unnecessary synchronization points in CPUs. The best strategy to divide the work among the threads varies depending on the type of device. RTCG can be used to select the appropriate base version or implementations of portions of the algorithm at run-time. Figure 1.5 shows the skeleton of a MxV vector kernel where a different variant of the algorithm is selected depending on the type of device. In the same vein, the size of the problem can advise the usage of different base versions of the algorithm.

`Local memory`: The usage of local memory is crucial for some devices like GPUs. We propose a technique to dynamically adjust the usage of local memory in HPL kernels. The idea is to write kernels where one or several data structures

Listing 1.6: MxV code: local memory usage

```
 1 class MxV { //Other portions of the class have been elided
 2   void operator()(Array<float,2> a, Array<float,1> x, Array<float,1> y,
 3                   Array<float,1,Local> lx) {
 4     Int k;
 5     if(copyX) {
 6       for_(k=lidx, k<N, k+=lszx)
 7         lx[k] = x[k];
 8       barrier(LOCAL);
 9     }
10     for_(k=0, k<N, k++)
11       y[idx] += a[idx][k] * (copyX ? (Float)lx[k] : (Float)x[k]);
12   }
13 }
14 int main(...) {
15   ...
16   eval(matvec).device(dev).global(M).local(localsize_x)(av, xv, yv, lxv);
17 }
```

may optionally be stored in local memory or not. For example, in the MxV code, we can choose vector `x` for this purpose. A boolean parameter `copyX` will be set in the host to indicate whether we want to place that array in local memory. Listing 1.6 contains the MxV kernel modified to implement this behavior. The kernel uses RTCG to make the copy of `x` to local memory if `copyX` is activated, see lines 5-9. When the computation is done, the global array `x` or its local copy will be used depending on the value of the `copyX` parameter in line 11.

4 HPL portable kernels through iterative optimization

The search of the optimal parameters for the kernel is performed using an iterative optimization process guided by a Genetic Algorithm (GA). Concretely, we have built the iterative search on top of the sequential version of the GAlib genetic algorithm package [16]. The chromosomes of our GA, which are potential solutions to our problem, have one gene per configuration parameter of the kernel. The initial population of the algorithm is composed of a configurable number of individuals that have been fixed by experimentation. The individuals and chromosomes of the initial population are randomly generated. Each individual generates a different version of the kernel using the values selected for each configuration parameter. These versions are evaluated using their fitness function, which is its execution time.

The minimum execution time obtained by a member of the population is used to decide whether the search must finish. The condition for this is that the fitness function (the execution time) has not improved for five generations. When this happens, the chromosomes that provided the best solution are used to generate the optimal kernel. If the condition has not been reached, a new generation of individuals is generated. This generation is created starting from the best individuals of the previous generation, and using mechanisms such as crossover and mutations. The process is repeated until the fitness function has not improved for five generations.

5 Experimental results

The techniques just described have been applied to implement a self-adaptive version of a matrix multiplication ($C = A \times B$) that has been tested on a **CPU** socket of two Intel Xeon E5-2660 Sandy Bridge with eight 2.2Ghz cores and hyper-threading (8×2 threads per processor, for a total of 32) and 64 GB of RAM, an Intel Xeon Phi 5110P **Accelerator** with sixty 1.053GHz cores with 8 GB of RAM, and an NVIDIA Tesla Kepler K20m **GPU** with 5 GB GDDR5. The Intel OpenCL 2013 R3 was used for the CPU and the accelerator, and the NVIDIA CUDA 5.0.35 toolkit for the GPU.

At the top level, our kernel chooses between two base versions of the algorithm in which matrices are processed following a block-cyclic approach. One version is more suitable for CPUs and the Xeon Phi, as it uses neither local memory nor cooperation among threads and each thread works on blocks of sizes specified by the user at runtime. The other version, where the work is distributed among the threads in tiles of a given size, and the data of each one of the input matrices used by each thread group can be copied or not to local memory, so that local domain sizes play an important role, better fits GPUs for this algorithm. Both versions also allow to adjust the size of the global domain of the execution as well as the degree of unroll of the innermost loop at runtime. Table 1 lists the configuration parameters of the kernel that are adjusted by the GA through the iterative process, and their baseline values. The N/A labels indicate the combinations in which the parameter is not applicable to the device, whose type is obtained at run-time. The baseline of the experiments is the parametrized HPL code. In this baseline, the global domain has been chosen following policies adequate to each platform and the usage of local memory is disabled. This way, the CPU baseline uses a global domain of (8,4) threads, as there are 32 threads available in the socket, and a consecutive block distribution. The baseline of the Xeon Phi and the GPU version has a global workspace with the shape of the destination matrix (so each kernel instance computes a single item) and it directly reads the input matrices from global memory. Furthermore, in all the baselines the size of the local workspace was automatically selected by the OpenCL driver, and no innermost loop unrolling was applied. The usage of an HPL baseline over an OpenCL implementation allow us to measure the impact of the tuning of the parameters.

Table 2 shows for each one of the platforms and for three matrix sizes the execution time of the baseline HPL version of the kernel, the execution time of the version tuned using our tool, the speedup of this version with respect to the baseline, and the execution time of the tool itself. Although HPL has very small overheads with respect to native OpenCL [13], we have exclusively measured the runtime of the underlying OpenCL kernel generated to provide maximum accuracy. We can see that speedups of between 1.01 and 4.67 were achieved in all

Table 1: Parameters adjusted by the genetic algorithm and their baseline values, where c and r are the number of columns and rows of the destination matrix C, respectively. N/A=not applicable.

Name(s)	Description	Baseline		
		CPU	ACC	GPU
uf	Unroll factor	1		
szx, szy	Global size of both dimensions	(8,4)	(c,r)	(c,r)
lszx, lszy	Local size of both dimensions	Auto		
bszx, bszy	Block size of the block-cyclic distribution	(c/szx,r/szy)		N/A
T	Tile size for the copies of A and/or B	N/A		-
CopyA, CopyB	Copy or not arrays A and B respectively	N/A		false

Table 2: Execution times and speedups achieved by generated kernels

Device	Size	Baseline time (s)	Kernel time (s)	Speedup	Tool time (s)
CPU	1024	0.176	0.092	1.91	207
	2048	1.252	0.706	1.77	627
	4096	114.369	24.478	4.67	20651
ACC	1024	0.034	0.031	1.09	1534
	2048	0.246	0.243	1.01	2598
	4096	2.207	2.129	1.03	4479
GPU	1024	0.016	0.013	1.24	198
	2048	0.167	0.108	1.54	490
	4096	1.509	0.996	1.52	2241

the platforms over hand-tuned baselines, justifying the interest of this approach. The high execution time of the tools is due to the fact that the search process is guided by the execution time. This time could be reduced if the search is guided, or at least pruned, using analytical models [17, 18].

Table 3 shows the optimal values found for each test case. The wide variety of solutions indicates the difficulty of finding a priori heuristics to choose the best parameters, making search necessary. In fact, some results are counterintuitive. For example, large numbers of workitems, much larger than the number of cores available, yielded always the best performance in the CPU and the Xeon Phi. The reasons are probably that in these devices the OpenCL framework coarsens multiple kernel instances into a single task ([19] indicates a work group is the smallest task scheduled on a software thread in the Xeon Phi), and that it may use several software threads per hardware thread in order to achieve the best performance. In fact this selection matches for example the manufacturer recommendation for the Xeon Phi [19], which was also followed to choose its baseline parameters. Still, the tool is able to further tune the parameters to increase the performance of the code in this platform. Similarly, copying the input matrices to the local memory not always achieved the best performance in the GPU, as we can see in the experiment with the 2048×2048 matrix. It is also interesting that adjusting isolatedly some of the parameters to their optimum

Table 3: Optimal values selected for each generated kernel

Device	Size	Optimal values					
		szx, szy	lszx, lszy	bszx, bszy	T	CopyA, CopyB	uf
CPU	1024	(1024,8)	(8,4)	(1,256)	-	-	2
	2048	(1024,8)	(8,8)	(1,1024)	-	-	8
	4096	(1024,256)	(1024,1)	(1,4)	-	-	4
ACC	1024	(1024,1024)	(64,1)	(1,1)	-	-	2
	2048	(2048,2048)	(64,1)	(1,1)	-	-	1
	4096	(4096,4096)	(32,1)	(1,1)	-	-	1
GPU	1024	(128,1024)	(16,16)	-	16	true, true	4
	2048	(2048,256)	(32,16)	-	4	false, true	4
	4096	(2048,1024)	(16,32)	-	4	true, true	1

value in the original kernels can actually generate slowdowns, which justifies the need to take into account all the parameters simultaneously in the search process.

6 Conclusions

Performance portability is an open problem in heterogeneous systems. This work proposes a set of techniques to generate codes that self-adapt to different devices at run-time. Our approach generates HPL kernels that can be tuned through a set of parameters whose optimal values are searched following an iterative process based on a genetic algorithm. The results show that our strategy generates versions of the kernels up to 4.67 faster than baselines based on heuristics. Such improvement is observed in a CPU, whereas the improvements achieved in a Xeon Phi and a GPU reach 9% and 54%, respectively. We plan to explore the application of more optimization techniques using our approach and to enhance the search process with effective heuristics or analytical models.

Acknowledgements

This work was supported by the Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (Ref. GRC2013/055) and by the Spanish Ministry of Science and Innovation (Ref. TIN2010-16735), both of them cofunded by FEDER funds of the European Union.

References

1. Munshi, A., Gaster, B., Mattson, T.G., Fung, J.: OpenCL Programming Guide. Addison-Wesley Professional (2011)
2. Wernsing, J.R., Stitt, G.: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. SIGPLAN Not. **45**(4) (April 2010) 115–124

3. Moore, N., Leeser, M., Smith King, L.: VForce: An environment for portable applications on high performance systems with accelerators. J. Parallel Distrib. Comput. **72**(9) (2012) 1144–1156
4. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Comput. **38**(8) (2012) 391–407
5. Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: Proc. 4th Intl. Workshop on Multicore Software Engineering. IWMSE '11 (2011) 25–32
6. Lutz, T., Fensch, C., Cole, M.: PARTANS: An autotuning framework for stencil computation on multi-GPU systems. ACM Trans. Archit. Code Optim. **9**(4) (January 2013) 59:1–59:24
7. Thoman, P., Kofler, K., Studt, H., Thomson, J., Fahringer, T.: Automatic OpenCL device characterization: Guiding optimized kernel design. In: Euro-Par'11. Volume 6853 of LNCS. Springer-Verlag (2011) 438–452
8. Lan, Q., Xun, C., Wen, M., Su, H., Liu, L., Zhang, C.: Improving performance of GPU specific OpenCL program on CPUs. In: Proc. 13th Intl. Conf. on Paral. and Distrib. Computing, Applications and Technologies (PDCAT'12). (2012) 356–360
9. Shen, J., Fang, J., Sips, H., Varbanescu, A.: Performance traps in OpenCL for CPUs. In: Proc. 21st Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2013). (2013) 38–45
10. Daloukas, K., Antonopoulos, C.D., Bellas, N.: GLOpenCL: OpenCL support on hardware- and software-managed cache multicores. In: Proc. 6th Intl. Conf. on High Performance and Embedded Architectures and Compilers. (2011) 15–24
11. Fabeiro, J.F., Andrade, D., Fraguela, B.B.: OCLoptimizer: An iterative optimization tool for OpenCL. In: Proc. Intl. Conf. on Computational Science (ICCS 2013). (2013) 1322–1331
12. Dolbeau, R., Bodin, F., de Verdiere, C.: One OpenCL to rule them all? (2013)
13. Viñas, M., Bozkus, Z., Fraguela, B.B.: Exploiting heterogeneous parallelism with the Heterogeneous Programming Library. J. Parallel Distrib. Comput. **73**(12) (2013) 1627–1638
14. Beckmann, O., Houghton, A., Mellor, M., Kelly, P.H.J.: Runtime code generation in C++ as a foundation for domain-specific optimisation. In: Domain-Specific Program Generation. Volume 3016 of LNCS. Springer Verlag (2004) 291–306
15. Newburn, C., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S.D., Wang, Z.G., Du, Z., Chen, Y., Wu, G., Guo, P., Liu, Z., Zhang, D.: Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In: 9th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO 2011). (2011) 224–235
16. Wall, M.: GAlib: A C++ Library of Genetic Algorithm Components. (1996)
17. Fraguela, B.B., Carmueja, M.G., Andrade, D.: Optimal tile size selection guided by analytical models. In: Procs. of Parallel Computing (ParCo). (2005) 565–572
18. Fraguela, B.B., Voronenko, Y., Püschel, M.: Automatic tuning of discrete fourier transforms driven by analytical modeling. In: Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques. (2009) 271–280
19. Intel Corp.: OpenCL design and programming guide for the Intel Xeon Phi coprocessor. http://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor (2014) [Online; accessed 29-May-2014].