Programming Many-Cores on Different Levels of Abstraction

Pieter Hijma[1], Rob V. van Nieuwpoort[2], and Henri E. Bal[1]

[1]*VU University Amsterdam*
{pieter,bal}@cs.vu.nl

[2]*Netherlands eScience Center*
R.vanNieuwpoort@esciencecenter.nl

Abstract

Many-core hardware is targeted specifically at obtaining high-performance. However, obtaining high-performance is often challenging because hardware-specific details have to be taken into account. This leads to low-level code that is difficult to read, maintain, and port to other architectures.

This is a well-recognized problem and there are many programming models that try to balance a high-level programming model with high-performance. We propose an entirely different, novel approach that allows programmers to define and choose their own level of abstraction: a high level of abstraction for readability and portability, and user-defined, lower levels to incorporate more hardware-specific details to obtain higher performance.

1 Introduction

The high performance that many-cores offer makes them a compelling target for the growing performance needs in industry and science. This growing interest requires access to many-core programming for a large group of programmers. However, programming many-cores is challenging because hardware-specific details have to be taken into account to obtain high-performance.

Low-level programming models offer much control to incorporate hardware-specific optimizations. However, since knowledge about the architecture is required to optimize code, only experts can effectively obtain high-performance. Moreover, optimizations are often implicit in the code, leading to low-level code that is difficult to read, maintain, and port.

A common solution for this problem is to introduce high-level abstractions that hide the low-level details. This simplifies programming many-cores and makes it accessible to a large group of programmers. However, since the focus of programming many-cores is so much on performance and performance usually depends on hardware-specific details, we question whether high-level abstractions are the right solution for programming many-cores.

We do not oppose high-level abstractions, but we are in search of a programming model that can provide programmers high-level abstractions while not losing control and understanding over hardware-specific details. Our approach tries to combine these requirements by offering a programming model with multiple levels of abstraction. We aim to provide programmers control and understanding of many-core hardware on a level of abstraction they can choose themselves. Programmers can start working on a high level and, if they wish so, can apply step-wise refinement for performance by moving to lower levels, which gently introduces them to more specific hardware details.

To accomplish this, we propose an entirely different, novel approach that centers around three main ideas:

Hardware descriptions As architectural details are important for obtaining performance on many-cores, user-defined hardware descriptions and the way in which an algorithm is mapped to the hardware form an integral part of the programming model.

Multiple levels of abstraction These user-defined hardware descriptions are organized in a hierarchy. Each lower level describes hardware in more detail, which allows programmers to choose the abstraction level to trade off performance against code readability, maintainability, and portability.

Performance Feedback Programmers receive detailed feedback from the compiler about performance and optimization strategies that matches the level of abstraction on which programmers are writing code.

In the next section we discuss the requirements we deem necessary for many-core programming. We relate these requirements to other research in Sec. 3. Section 4 puts forward our view in detail. In Sec. 5 we discuss the research agenda of our work and the status of the system we are implementing. We conclude our paper in Sec. 6.

2 Requirements

We will discuss the set of requirements from two perspectives. The performance perspective is motivated by the growing need for performance and performance being the main purpose of many-core hardware. The software-engineering perspective is motivated by the growing interest in many-cores.

R1: control and understanding From the performance perspective, the main requirement can be summarized as *control and understanding*. To obtain performance, programmers typically want and need control over low-level hardware details. They also want to have a thorough understanding of low-level details of hardware to be able to exploit them. Often, understanding the compiler is also necessary since compilers can have a significant impact on performance. Finally, programmers want to have a good understanding of the performance obtained by their code. To summarize, requirement R1 focuses on three aspects: control over and understanding of the hardware, the compiler, and the performance.

R2: access to a large group of programmers From the software-engineering perspective, having only experts being able to obtain high-performance does not scale. Therefore, many-core programming must be made accessible to a large group of programmers with different levels of expertise in such a way that they too can obtain high-performance.

R3: maintainable code The growing interest in many-core programming leads to a growing code-base of many-core programs. With a growing code-base, typical software-engineering topics such as readability and maintainability become important.

R4: retain knowledge about optimizations This code-base may contain codes that are heavily optimized. Optimizing programs brings forth difficult issues. First, optimizing codes usually requires a large time-investment from programmers. Second, optimized codes are often challenging to understand and maintain because they contain low-level hardware details that obscure the original algorithm, and because the optimizations are often implicit in the code (e.g. coalescing or cache optimizations).

These observations lead to the following requirement that focuses on the knowledge, the rationale behind the optimizations, that is, why programmers make certain decisions in a specific implementation. Because of the large time-investment and the issue that these codes are difficult to understand, it is important that this knowledge is *retained* as much as possible and that this knowledge can be *reused* for other implementations.

R5: portability Finally, due to the fact that there is a large variety of many-core hardware and the architectures evolve fast, an important requirement is portability between many-cores and different generations of many-cores with a focus on performance portability.

3 Related Work

CUDA [23] and OpenCL [25] are languages that give much control over hardware-specific details, making them strong on requirement R1 for the aspect of hardware. They have explicit constructs for hardware features such as fast on-chip memory, which often makes the code less maintainable (R3). Other hardware features can be controlled but are *implicit*, such as access to parallel memory banks in fast on-chip memory. However, as not all hardware features are explicitly part of these languages but are often necessary for performance, optimizing programs often leads to code that does not explain certain optimization decisions. This makes these languages weak on requirement R4, retention of knowledge. An example is the use of a data-structure with a non-standard layout that is beneficial for memory-bank access.

The great level of control that CUDA and OpenCL provide makes them less accessible to mainstream programmers (R2). Requirement R5, portability, is different for CUDA and OpenCL. CUDA code cannot be ported to non-NVIDIA GPUs, whereas OpenCL is available for other architectures, although performance portability cannot be guaranteed.

To provide programmers insight in the performance, an aspect of requirement R1, profilers are available for CUDA and OpenCL that give programmers statistics about their code. Although the feedback is useful, the feedback is usually per function, only for a specific instance of a function, and does not map back to specific code within the function. NVIDIA's NSight has an interesting feature to increase the knowledge of programmers by pointing to relevant sections in the programming guide.

Several systems raise the level of abstraction to make many-core programming more mainstream (R2). Often, these languages center around a core abstraction, such as streaming [4, 11], Bulk-Synchronous Parallelism [26, 15], Nested Data-Parallelism [3, 2, 6, 22] or powerful arrays [10, 13, 16, 18, 8, 9, 21]. Another approach is instrumenting legacy codes with directives to accelerate these codes [14, 17, 19, 29, 20].

These languages force programmers to work on a single abstraction-level. A high level of abstraction means that details are hidden, which makes it hard to satisfy requirement R1. Less control often means that programmers wonder whether more performance is possible and how to express this at this higher level of abstraction. However, hiding details is beneficial for mainstream programmers (R2), maintainable code (R3), and portability (R5).

Several systems automatically optimize naively written GPU kernels [30, 27]. An interesting feature is that the GPGPU compiler [30] emits code that is reasonably easy to read for every optimization-step it performs. This allows programmers to understand the optimizations, contributing to requirement R1 on the compiler aspect. The more high-level interface is a good match for mainstream programmers (R2) and maintainable code (R3). However, compilers have to be conservative, need to work for the general case, and do not have the application knowledge that programmers have. Furthermore, programmers are also limited in their control (R1) when their automatically optimized kernels do not perform as they expected.

Delite [7] and the work by Cartey et al. [5] provide frameworks for building DSLs (Domain-Specific Languages) on top of a performance library to separate the concerns between domain-experts and performance-experts. This is an interesting approach that can be tailored to the needs of domain-experts (R2). However, building DSLs is a difficult task that might not scale to the demand from the growing interest in many-core programming. Besides, it provides the domain-experts, the ones requiring performance, no understanding of and control over the performance (R1). Additionally, the performance depends on good communication between the performance-experts and the domain-experts.

Our work relates to the following systems. Sequoia [12] introduces tasks that recursively call each other as main programming abstraction. It also provides user-defined descriptions of memory hierarchies that define how different task variants are mapped to the memory hierarchy. In comparison to Sequoia, our proposed system generalizes the memory hierarchy descriptions to hardware descriptions, our proposed system does not depend on task abstractions, we aim to provide a more direct mapping between algorithm and hardware than Sequoia has, and we offer multiple levels of abstraction.

NVIDIA's Thrust library [1] provides a C++ Standard Template Library-like interface with a vector data-structure. Thrust is tightly integrated with CUDA which makes it possible to replace performance critical Thrust code with specialized CUDA functions. This gives programmers a choice in two levels of abstraction. However, Thrust only provides one-dimensional vectors whereas our proposed system provides a more general programming model with multiple levels of abstraction.

4 Our View

This section describes our position on how the requirements in Sec. 2 can almost all be satisfied simultaneously. In summary, we want to give programmers control and understanding of the hardware and the performance on a level of abstraction they can choose themselves.

4.1 Hardware descriptions

Our first idea follows from the observation that a program can be regarded as an algorithm that is mapped to hardware. In most programming models, the hardware aspect is implicit, but as hardware details are important for many-core programming, we propose to make the hardware explicit. This allows programmers to express both the algorithm and the hardware to make a clear connection between the two.

To this end, we introduce hardware descriptions as an integral part of the programming model. Programmers can firstly specify *what* hardware they are mapping their algorithm to, and secondly, they can define *how* the algorithm is mapped to the hardware.

Having explicit hardware descriptions as an integral part of the programming model has several benefits. First,

category	examples of features
parallelism hierarchy	threadblocks, warps, threads
memory hierarchy	on-chip memory, caches, memory sizes, access pattern rules, alignment rules, coherency rules
instruction execution	vector instructions, vector-size, control-flow rules
configurable options	cache sizes, memory-bank width

Table 1: Features that should be described by hardware descriptions.

they allow programmers to understand and control important aspects of the hardware, which is needed by requirement R1. Additionally, this fulfills requirement R4, retention of optimization knowledge. With hardware descriptions, programmers obtain a more formal and explicit way of specifying the connection between the algorithm and the hardware they target, making hardware-specific optimizations explicit in the program. Finally, hardware descriptions can guide programmers to take into consideration low-level details of the architecture and it gives programmers the opportunity to understand the hardware in relation to the algorithm to implement.

To make this relation between hardware and algorithm clear and understandable, the programming model should provide constructs to define the mapping between algorithm and hardware. Furthermore, the hardware descriptions should be high-level, clear, and understandable by programmers. The hardware descriptions do not have to be a complete and accurate description of the hardware, but they should contain the details that are important for performance. They should also be general enough to specify features of different architectures. Table 1 shows several examples of features that should be considered for inclusion. Typically, these details can be found in the programming guides of many-core hardware. In essence, we want to formalize these details and incorporate them in our programming model.

4.2 Multiple layers of abstraction

A quote by Alan J. Perlis reads: "A programming language is low-level when its programs require attention to the irrelevant." Low-level hardware descriptions may contain an over-specification of details that are not relevant for obtaining performance for a certain algorithm. Mapping an algorithm to low-level details may also be time-consuming.

Therefore, we propose to organize the hardware descriptions in a hierarchy. Each lower level in the hierarchy

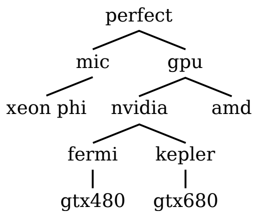

Figure 1: An example of a hierarchy of hardware descriptions.

extends its parent and describes hardware in more detail. The hardware description is chosen *per function*, allowing programmers to apply step-wise refinement for performance and to focus on the most performance-critical parts of the code. Additionally, it should be possible to automatically translate code that was written for higher-level hardware descriptions to code that adheres to lower-level hardware descriptions using source-to-source translation.

Figure 1 shows an example of a possible hierarchy of hardware descriptions. At the root we describe "perfect" many-core hardware that is relatively easy to program. A possible work-flow could constitute a programmer writing code for level "perfect" aiming to compile this for a "gtx680". The code is first translated to human-readable code for level "gpu", then for level "nvidia", then "kepler", and finally for level "gtx680". At each stage, the programmer is free to adjust the code to better incorporate hardware-specific details in the code. The hierarchy defines the direction of the translation (only down in the tree). It does not necessarily mean that descriptions of specific features, for example a SIMD unit, in one branch of the tree cannot be reused in hardware descriptions in other branches.

To make our view more concrete, Fig. 2 and 3 show a simple vector addition program for hardware description "perfect". The hardware description language is inspired by Aspen, a performance modeling language [24]. Aspen describes properties of compute kernels on a high-level together with hardware of cluster computers, whereas we describe many-core hardware on a lower level and incorporate the descriptions in a programming language. In Fig. 2, every colored identifier is a keyword in the hardware description language. The first code-block defines an abstract parallelism hierarchy with only one memory space `main` that is fully consistent and one layer of countably infinite units of parallelism called `threads`. The next code-block defines the device `perfect` with a memory `mem`, an interconnect `ic`, and execution units `cores`. The memory holds memory space `main` and has unlimited capacity. The interconnect connects the memory to all cores, has 1 cycle latency, and unlimited bandwidth. Finally, there are countably infinite cores available, that each execute 1 thread. Operations, for example addition, take only one cycle.

```
parallelism hierarchy {
    memory_space main {
        consistency = full;
    }

    group threads {
        nr_units = countable;

        unit par_unit thread {
}   }   }

device perfect {
    mem;
    ic;
    cores;
}

memory mem {
    space(main);
    capacity = countable B;
}

interconnect ic {
    connects(mem, cores.core[*]);
    latency = 1 cycle;
    bandwidth = countable bit/s;
}

group cores {
    nr_units = countable;

    unit execution_unit core {

        slots(thread, 1);

        instructions ops {
            (+), latency = 1 cycle
            ...
}   }   }
```

Figure 2: The hardware description "perfect".

```
import perfect;

perfect void add(int size, float c[size],
        float b[size], float a[size]) {

    foreach (int i in size threads) {
        c[i] = a[i] + b[i];
}   }
```

Figure 3: An example of a vector addition program for hardware description "perfect".

Figure 3 shows the vector addition program. The function `add()` is annotated to adhere to the rules of hardware description "perfect". The `foreach` loop expresses parallelism, in this case for `size` many `threads` that each have an index `i`. The identifier `threads` refers back to the group of parallelism units defined in parallelism hierarchy `hierarchy` in Fig. 2. Hardware descriptions on lower levels have more complicated parallelism hierarchies, for example threadblocks and threads, more memory-spaces, for example shared, local, constant, global, different interconnects, caches, or SIMD units.

A hierarchy of hardware descriptions has several advantages. It offers multiple levels of abstraction giving programmers a choice in writing high-level or low-level code. High-level code is easier to understand, maintain (R3), and port (R5). It can also serve as reference code for documentation purposes and automatic correctness checks for lower-level code. Moreover, programmers can apply step-wise refinement for performance, giving them more control over the hardware on each lower level (R1).

Furthermore, such a hierarchy can make many-core programming accessible to a large group of programmers with different levels of expertise (requirement R2). Programmers with less expertise can start working on a high-level and be gently introduced to lower-level hardware details. Programmers can trade off the amount of time they want to invest, portability of their code, and performance, depending on their level of expertise.

Often, compilers are black boxes that do not give much feedback about how the code was compiled. Although a compiler plays a large role in the performance of the code, programmers cannot verify the behavior of the compiler well. The human-readable source-to-source translation between the abstraction levels gives programmers good insight in the compiler (R1). It provides a framework to understand the limitations of the compiler while not giving up control because programmers can override the generated code at each level.

We believe that control to override the compiler is necessary because compilers have to be conservative, need to work for the general case, and do not have the application knowledge that programmers have. Although high-level code gives the opportunity to be ported automatically to code for lower-level hardware descriptions, we do not believe that *performance* can be ported automatically. This leads to not fully satisfying requirement R5. However, once code is hand-optimized for a specific architecture, similar techniques are easier to incorporate in low-level code for other architectures, since the optimizations are more explicitly and formally described in relation to the hardware.

To make this possible, the programming model should fulfill additional conditions. The hardware descriptions should be extensible, for example using standard object-oriented techniques. Furthermore, a standard library of hardware descriptions should be available. However, to give programmers a high-degree of control on aspects important for their algorithms, the hardware descriptions should be user-defined. For example, programmers can make separate hardware descriptions for compute-intensive or data-intensive algorithms. Finally, the compiler should be able to use these user-defined hardware descriptions to generate code. More specifically, the compiler should be able to automatically map code written with a high-level hardware description into code for lower-level hardware descriptions.

4.3 Performance feedback

The previous subsection explained that the compiler generates human-readable code in the same language that programmers specify their algorithms and mappings to hardware in. This is already important feedback for programmers who want to obtain performance. However, for requirement R1, control and understanding performance, we want the compiler to give more feedback about performance and optimization strategies. To come back to the quote by Alan J. Perlis in the previous subsection, in our experience the difficult issue in many-core programming is deciding which details are relevant and which are not. Detailed feedback can alleviate this issue.

Hardware descriptions contain structured knowledge about hardware specifics that can be used to implement user-defined performance functions to generate feedback. For instance, programmers can write functions that give feedback about the complexity of the algorithm, the amount of parallelism, the amount of memory-bank conflicts, control-flow divergence, or the ratio of data-loads compared to the floating point operations or indexing operations.

There are several benefits to this performance feedback framework that incorporates hardware descriptions. Programmers get details about performance on the abstraction level they are working on. However, programmers can obtain more detailed performance feedback by translating the code to lower-levels and apply the feedback functions on these levels. As the mapping between the algorithm and the hardware is more formally described, the compiler is capable of giving feedback with a strong relation to the code. We strive for good interaction between programmer and a compiler that suggests optimization strategies such as data-layout changes. In this role, the compiler does not have to be conservative since the programmer remains responsible.

To support this, the programming model should satisfy several conditions. The hardware descriptions should be extended with means to specify performance functions. The compiler should provide basic functions, such as counting instructions and data accesses that can be used in the performance functions. The compiler should also be able to express performance in terms of statically unknown information, for example using symbolic execution.

5 Status and Research Agenda

Our research focuses on the technical issues that need to be overcome to make the above a reality. A challenging issue is providing hardware descriptions that are not only general enough for current and future hardware, but also high-level enough to be understood by programmers and detailed enough to be informative for compilers. Ideally, we would like hardware manufacturers to develop hardware descriptions on different levels in addition to the informal programming guides. Another challenging issue is the automatic translation of code between levels in the hardware description hierarchy. This means that a compiler should make default decisions for incorporating the more detailed hardware features of lower levels. Finally, it is challenging to provide programmers with accurate feedback. This is especially true when hardware contains caches and when not enough information is available statically.

We have started to implement a system that is not yet mature enough to be evaluated against what we propose in this paper. Currently, our system supports GPUs, provides a DSL for describing hardware and a language for specifying algorithms. The hardware description language is under development to support describing hardware in more detail and hierarchies of hardware descriptions. However, the hardware descriptions do support performance functions. For instance, we can generate feedback for the number of floating point instructions, the number of data-accesses, the arithmetic complexity, and the occupancy. We can compile programs to executable code that automatically reports the attained memory bandwidth and the amount of gigaflops. In addition, the system provides programmers feedback in the form of a visualization to inspect data-dependencies per thread and data-access patterns for each thread.

There are several other directions that can be explored. For example, enabling automatic optimizations in such a way that programmers can track and understand the reasoning of the compiler. An important optimization is fusion of functions to minimize overhead [28]. An interesting direction is to make fusion possible, automatic, and insightful for programmers between functions on different abstraction levels. Another example is to research whether the more accurate hardware information can prune the search-space for auto-tuners and can provide an understanding of why a certain set of parameters leads to high-performance. Another direction is to research whether clusters of many-cores can be described, for example by incorporating networking in the hardware descriptions. Finally, the concept of multiple layers of abstraction may be generalized to other programming models that need to trade off high-level abstractions against low-level details for other reasons than performance.

6 Conclusion

Programming many-cores is challenging because hardware-specific details need to be taken into account to obtain high-performance. Low-level programming models give experts control over the hardware, but this is not in reach of a large group of programmers. The common solution is to introduce high-level abstractions, but these abstractions hide the hardware-specific details that are important for performance. In our view, many-core programmers need a model that gives them control and understanding of hardware on a level of abstraction they can choose themselves to trade off maintainability, portability, and performance. In this paper, we proposed an entirely different, novel approach that integrates hardware descriptions in the programming model, offers multiple levels of abstraction by organizing the hardware descriptions in a hierarchy, and provides performance feedback functions using the structured knowledge about the hardware in the hardware descriptions.

References

[1] Bell, N., and Hoberock, J. Thrust: A Productivity-Oriented Library for CUDA. In *GPU Computing Gems*. Morgan Kaufmann Publishers, 2011, pp. 359–371.

[2] Bergstrom, L., and Reppy, J. Nested Data-Parallelism on the GPU. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional programming* (New York, NY, USA, 2012), ICFP '12, ACM, pp. 247–258.

[3] Blelloch, G. E. Programming Parallel Algorithms. *Commun. ACM 39*, 3 (1996), 85–97. Ok.

[4] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph. 23*, 3 (Aug. 2004), 777–786.

[5] Cartey, L., Lyngsø, R., and de Moor, O. Synthesising Graphics Card Programs from DSLs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 121–132. Ok.

[6] Catanzaro, B., Garland, M., and Keutzer, K. Copperhead: Compiling an Embedded Data Parallel Language. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 47–56. Ok.

[7] Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., and Olukotun, K. A Domain-Specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 35–46. Ok.

[8] Chakravarty, M. M., Keller, G., Lee, S., McDonell, T. L., and Grover, V. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming* (New York, NY, USA, 2011), DAMP '11, ACM, pp. 3–14.

[9] Claessen, K., Sheeran, M., and Svensson, B. J. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Proceedings of the 7th workshop on Declarative Aspects and Applications of Multicore Programming* (New York, NY, USA, 2012), DAMP '12, ACM, pp. 21–30. Ok.

[10] Cunningham, D., Bordawekar, R., and Saraswat, V. GPU Programming in a High Level Language: Compiling X10 to CUDA. In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop* (New York, NY, USA, 2011), X10 '11, ACM, pp. 8:1–8:10.

[11] Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F., and Fink, S. J. Compiling a High-Level Language for GPUs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 1–12. Ok.

[12] Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. Sequoia: Programming the Memory Hierarchy. In *SC 2006 Conference, Proceedings of the ACM/IEEE* (nov. 2006). Ok.

[13] Guo, J., Thiyagalingam, J., and Scholz, S.-B. Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), DAMP '11, ACM, pp. 15–24.

[14] Han, T. D., and Abdelrahman, T. S. hiCUDA: High-Level GPGPU Programming. *IEEE Trans. Parallel Distrib. Syst. 22*, 1 (Jan. 2011), 78–90.

[15] Hou, Q., Zhou, K., and Guo, B. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Graph. 27* (August 2008), 19:1–19:12. Ok.

[16] Larsen, B. Simple Optimizations for an Applicative Array Language for Graphics Processors. In *Proceedings of the sixth workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), DAMP '11, ACM, pp. 25–34.

[17] Lee, S., and Eigenmann, R. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (Washington, DC, USA, 2010), SC '10, IEEE Computer Society, pp. 1–11.

[18] Lee, S., Grover, V., Keller, G., and Chakravarty, M. M. GPU Kernels as Data-Parallel Array Computations in Haskell. In *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009)* (2009).

[19] Lee, S., Min, S.-J., and Eigenmann, R. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2009), PPoPP '09, ACM, pp. 101–110.

[20] Lee, S., and Vetter, J. S. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 23:1–23:11.

[21] Mainland, G., and Morrisett, G. Nikola: Embedding Compiled GPU Functions in Haskell. *SIGPLAN Not. 45*, 11 (Sept. 2010), 67–78.

[22] Newburn, C. J., So, B., Liu, Z., McCool, M., Ghuloum, A., Toit, S. D., Wang, Z. G., Du, Z. H., Chen, Y., Wu, G., Guo, P., Liu, Z., and Zhang, D. Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2011), CGO '11, IEEE Computer Society, pp. 224–235. Ok.

[23] Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable Parallel Programming with CUDA. *Queue 6*, 2 (2008), 40–53.

[24] Spafford, K. L., and Vetter, J. S. Aspen: A Domain Specific Language for Performance Modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 84:1–84:11.

[25] Stone, J., Gohara, D., and Shi, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering 12*, 3 (2010), 66–73. Ok.

[26] Valiant, L. G. A Bridging Model for Parallel Computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[27] Wang, C., Kang, K., Zhu, M., and Deng, Y. A Polyhedral Modeling Based Source-to-Source Code Optimization Framework for GPGPU. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (Los Alamitos, CA, USA, 2012), vol. 0, IEEE Computer Society, pp. 1964–1970. Ok.

[28] Wang, G., Lin, Y., and Yi, W. Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU. *IEEE-ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing 0* (2010), 344–350.

[29] Wienke, S., Springer, P., Terboven, C., and an Mey, D. OpenACC - First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. Spirakis, Eds., vol. 7484 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2012, pp. 859–870.

[30] Yang, Y., Xiang, P., Kong, J., and Zhou, H. A GPGPU Compiler for Memory Optimization and Parallelism Management. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation* (New York, NY, USA, 2010), PLDI '10, North Carolina, Central Florida, ACM, pp. 86–97. Ok.