CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–39
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Stepwise-refinement for performance: a methodology for many-core programming

P. Hijma[1]*, R. V. van Nieuwpoort[2], C. J. H. Jacobs[1] and H. E. Bal[1]

[1] *VU University Amsterdam, Department of Computer Science*
[2] *Netherlands eScience Center*

SUMMARY

Many-core hardware is targeted specifically at obtaining high performance, but reaching high performance is often challenging because hardware-specific details have to be taken into account. Although there are many programming systems that try to alleviate many-core programming, some providing a high-level language, others providing a low-level language for control, none of these systems have a clear and systematic methodology as foundation.
In this article, we propose *stepwise-refinement for performance*: a novel, clear, and structured methodology for obtaining high performance on many-cores. We present a system that supports this methodology, offers multiple levels of abstraction to provide programmers a trade-off between high-level and low-level programming, and provides programmers detailed performance feedback. We evaluate our methodology with several widely varying compute kernels on two different many-core architectures: a GPU and the Xeon Phi. We show that our methodology gives insight in the performance and that in almost all cases we gain a substantial performance improvement using our methodology.
Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: methodology; stepwise-refinement; many-core; programming language; hardware description; abstraction-levels

1. INTRODUCTION

The high performance that many-cores offer makes them a compelling target for the growing performance needs in industry and science. However, obtaining high performance – the main purpose of many-cores – is challenging as hardware-specific details need to be taken into account, such as multiple levels of parallelism, explicit fast memories, and memory access patterns.

Languages such as CUDA and OpenCL offer much control over hardware-specific details. However, managing all these hardware-specific details to obtain high-performance is a complex task. Manually optimized programs are usually difficult to read and do often not portray well why they obtain high performance.

A common approach to simplify many-core programming is to raise the level of abstraction. However, whereas a high abstraction-level simplifies programming in *general purpose computing* where performance is less critical, in *many-core computing* it often results in hiding the very details that are important for performance.

In our view, this tension between control over hardware-specific details and raising the level of abstraction is not supported by a proper methodology in current work. In this article we

*Correspondence to: E-mail: pieter@cs.vu.nl present a methodology that we call *stepwise-refinement for performance*. This methodology presents programmers a structured approach in which they can start on a high-level and can then, if they desire, move on to lower levels of abstraction that expose more hardware details. We can illustrate this with a quote from Alan J. Perlis: "A programming language is low-level when its programs require attention to the irrelevant." In our experience, the difficult issue in many-core programming is deciding which details are relevant for performance. Our methodology aims to assist programmers with this decision.

A crucial aspect of the methodology is that the compiler and programmer work together to obtain high performance. The compiler is augmented with hardware knowledge with varying levels of detail resulting in multiple levels of abstraction. Using this hardware knowledge, the compiler assists the programmer by providing detailed *performance feedback*. The goal of our methodology is to structure the optimization process, give programmers a *trade-off* between high-level and low-level programming, and give programmers *understanding* of the performance they obtain.

We evaluate our methodology with Many-Core Levels (MCL), a system that supports multiple abstraction-levels for multiple many-core architectures. Our system has knowledge of many-core hardware by means of hardware descriptions encoded in our Hardware Description Language (HDL). Compute kernels are expressed in Many-Core Programming Language (MCPL) that makes the mapping between algorithm and hardware description explicit. Our compiler can generate code for each level of abstraction and combines the knowledge of the program and the hardware to provide detailed *performance feedback* to the programmer who can then *manually* transform the program.

This approach combines the strengths of both the compiler and the programmer. Programmers gain insight in the compiler, remain in control, and can use their application knowledge to transform the program. On the other hand, the compiler does not have to be as conservative in providing feedback as an automatically optimizing compiler has to be.

To show that our approach is effective, we implemented several, widely varying, and well-known compute kernels in MCL for two very different many-core architectures: an NVIDIA GPU and Intel's Xeon Phi. In almost all cases our approach results in substantial performance improvements over our highest-level code. We show that programmers in cooperation with our compiler have enough control to obtain high performance and gain an understanding of the performance during the stepwise-refinement process that leads the programmer through multiple abstraction-levels.

To summarize, this article presents the following contributions:

- A clear methodology for optimizing many-core programs,
- our system MCL that implements this methodology, released as open source [1],
- an evaluation of the methodology with several widely different many-core programs on two different many-core architectures,
- several implementation techniques that exploit the strong relation between hardware description and program.

Section 2 elaborates on many-core programming approaches. In Sec. 3 we introduce our methodology *stepwise-refinement for performance*. Section 4 gives an overview of Many-Core Levels and how our system implements our methodology. In Sec. 5 we give a detailed example of how the process of *stepwise-refinement for performance* takes place. Section 6 discusses several of the implementation techniques of our system. Section 7 evaluates our techniques for various well-known compute kernels. We conclude the article with a discussion and conclusion.

2. PROGRAMMING MANY-CORES

The challenges in many-core programming are widely recognized and there are many approaches that try to alleviate many-core programming. This section discusses the current status of programming many-cores and identifies issues (summarized in Table I) that we try to address in our work. There are roughly three programming approaches that can be distinguished: high-level programming, separation of concerns, and a tuning cycle approach.

Table I. Comparison of several programming approaches (○ is in between + and -)

approach	control	portability	understanding	high-level
high-level programming	-	+	-	+
separation of concerns	-	+	-	+
tuning cycle approach	+	-	+	-
MCL	+	○	+	(choice)

High-level programming The fact that low-level many-core programming is difficult is a well-recognized problem and several systems raise the level of abstraction to make many-core programming more accessible. Often, these languages center around a core abstraction, such as streaming [2, 3], Bulk-Synchronous Parallelism [4, 5], divide-and-conquer [6], Nested Data-Parallelism [7, 8, 9, 10] or powerful arrays [11, 12, 13, 14, 15, 16, 17].

High-level languages can be based on automatic optimization or on algorithmic skeletons. Automatic optimization relies on the premise that high performance can be obtained in all cases. This is difficult to realize in practice, because compilers have to work in the general case, have no application knowledge, and have to be conservative (e.g. assume aliasing). In general-purpose computing, since the focus is not on high-performance, sub-optimal performance may not be a problem, but it is an issue in many-core programming because obtaining high performance is the main goal.

There are two examples of automatic optimizers that optimize naively written GPU kernels [18, 19]. Although this is valuable work, a drawback of automatic optimizers is that the optimization knowledge is contained in the compiler and cannot be reused by programmers for applications that are less amenable to automatic optimization.

A second approach is to provide a programming model in which programmers express their algorithms in terms of algorithmic skeletons [20, 21, 22]. The skeletons are often manually implemented and optimized. However, these skeleton-based programming models rely on programmer insight to select the right parallel patterns for the application. Additionally, performance often relies on the composition of these algorithmic skeletons, which is a challenging transformation [23, 24], but has been applied to GPU programming [25].

Although raising the level of abstraction often provides a cleaner programming model and helps to achieve (non-performance) portability, it also means that hardware-specific details are hidden that may be necessary for obtaining high-performance. This makes understanding why a particular program has a particular performance more difficult and in case an application performs less than expected, programmers have fewer means to adapt the code to gain more performance.

Separation of concerns Another approach is based on separation of concerns. Delite [26] and the work by Cartey et al. [27] 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. However, building DSLs is a difficult task, especially when two different goals need to be addressed, namely making them expressive for the domain-experts and making the generated code amenable to optimizations. Besides, a DSL provides the domain-experts, the ones requiring performance, no understanding of and no control over the performance. Additionally, the performance depends on good communication between the performance-experts and the domain-experts.

Tuning cycle approach The tuning cycle approach is an iterative process that usually consists of the following steps: evaluate the performance of an application, analyze the gathered results, and refactor the code to increase the performance. This approach usually fits low-level languages such as CUDA [28] or OpenCL [29] that offer programmers high degrees of control over the code. However, it can also be applied to directive-based programming systems, where in each step more detailed directives are inserted. [30, 31, 32, 33, 34, 35].

The first step in this process, evaluating the performance of the application can be done in several ways. The performance can be measured using profilers or the performance can be estimated using performance models. Lopez-Novoa et al. conducted an excellent survey on performance modeling for many-cores [36].

The second step, analyzing the results can be very challenging. Profilers usually give feedback in the form of statistics about the code and give general feedback, which makes it difficult to understand how to act on the feedback. However, there is tool-support available in the form of the PerfExpert, which helps programmers to interpret the results [37]. PerfExpert does support many-core architectures but only gives feedback about which code-parts of an application could run well on a Xeon Phi or GPU [38, 39]. It is not capable of optimizing those kernels taking into account non-standard architectural details, such as warp execution or scratchpad memories. PerfExpert operates by interpreting the results from measurements, matching it against rules about the architecture to find a bottleneck in the program and recommends a list of optimizations, complete with code patterns that may solve the bottleneck.

In the final step, the code is refactored to increase the performance. Often this is done manually by the programmer, but PerfExpert can in some cases also apply the optimizations automatically [40] if statically can be verified that the transformation is possible.

This approach is performed on the lowest level of abstraction, which makes the applied optimizations not portable to other architectures, which is a major disadvantage considered the rapid evolution of many-core hardware. In addition, optimizations in low-level languages makes the code difficult to understand.

MCL Table I presents an overview of the discussed programming approaches in relation to our work. In comparison to high-level approaches and separation of concerns, MCL also provides a high-level of abstraction, but offers programmers control by allowing lower levels of abstraction, although this diminishes the portability. MCL is specifically targeted at providing programmers understanding of the performance they obtain and MCL offers a choice in the level of abstraction to work on.

In relation to the tuning cycle approach, tuning is often performed only to the lowest level of abstraction, where programmers have much control. This makes the optimizations not portable to other many-core architectures. In contrast, MCL allows to write optimizations on each level of abstraction, making the optimization available to all lower levels of abstraction in the sub-tree of the hardware-description hierarchy (Sec. 5 gives an example of this).

Additionally, low-level languages do often not provide enough language features to express optimizations clearly in relation to the underlying hardware. For example, CUDA and OpenCL have *explicit* constructs for low-level hardware features, such as accessing fast on-chip memory, but other important hardware features remain *implicit*, such as parallel banks in accessing memory, or GPU threads that execute in warps, a pipelined manner to overcome memory latency.

Because not all hardware features are explicitly part of these languages but are important for performance, in our experience, optimizing programs often leads to code that does not explain well why certain optimization decisions were made. Examples are data structures with a non-standard layout and loops that iterate in a particular manner for improved memory access. This makes implementing suggestions from analysis tools difficult because programmers can not relate the code to the underlying hardware. Because hardware descriptions in MCL are tightly coupled to the programming language, MCL provides more means to express the optimizations in relation to the hardware.

Furthermore, because optimizations are applied on multiple levels of abstraction, the optimizations are also traceable. It is clear which optimizations have been applied for which hardware and it is also clear on which feedback the optimizations are based.

MCL mainly relies on static information encoded in the program and hardware descriptions, which allows a fast development cycle for experimentation. In contrast, profilers need several runs to record data or runs with instrumented code for tools such as PerfExpert, which results in a longer development cycle. However, since run-time information is more accurate than static information, we consider this type of analysis as very valuable and in-line with MCL. However, in MCL run-time analysis is typically performed at the lowest-level of abstraction where static information is no longer precise enough and where the longer development cycle can be trade-off with performance.

There are several approaches from which our work draws inspiration. NVIDIA's Thrust library [41] 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 programming model advocates a methodology, albeit a simple one: prototype with Thrust, rewrite the hot-spots by falling back to CUDA and optimize these kernels.

Sequoia introduces tasks that recursively call each other as main programming abstraction [42]. 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 system generalizes the memory hierarchy descriptions to hardware descriptions, MCL does not depend on task abstractions, we aim to provide a more direct mapping between algorithm and hardware than Sequoia does, and we offer multiple levels of abstraction.

MCL's hardware description language is inspired by Aspen, a performance modeling language [43]. Aspen describes high-level properties of compute kernels with hardware of cluster computers. In MCL we focus on describing many-core hardware instead of cluster computers and we do this on a lower level. Our approach also differs in that we do not describe properties of compute kernels, but instead explicitly define the mapping between the algorithm and hardware constructs.

In general our hardware descriptions could be seen as models of the hardware that have certain performance characteristics. In this sense our work relates to performance modeling. In the context of distributed systems, Ammar [44] worked on hierarchical performance models. At the highest level they describe the performance characteristics and requirements of a distributed system at the application level. At lower levels, the performance characteristics of interacting software components are modeled with increasing precision, resulting in an accurate performance model. While Ammar's hierarchy models performance characteristics of software components, our hierarchy entails different abstractions for hardware.

3. STEPWISE REFINEMENT FOR PERFORMANCE

From the previous section we can distill several requirements for many-core programming. First, there is a need for low-level programming languages that provide programmers control over the hardware. Solutions differ in how explicit this control is in the language. Second, architectures of many-cores change rapidly which makes portability an important issue. A third important aspect is understanding performance. Tools such as profilers try to fill this gap. Furthermore, there is also a need for high-level programming that abstracts away hardware-details. However, this comes at the cost of loss of control. Finally, none of the approaches has a good solution for handling the tension between control over hardware and raising the level of abstraction, and shows support for the optimization process over more than one abstraction level. This is an important point because optimizing is a laborious process which often leads to code that is difficult to read and maintain. Often, it is also unclear why exactly these codes perform as well as they do.

In this section we propose the *stepwise-refinement for performance* methodology, a structured approach that combines all of these requirements to help the programmer to obtain high performance. For our methodology, we assume an existing application with computational hot-spots or kernels that are to be ported to a many-core device. Such a computational kernel is the starting point of our methodology.

3.1. Philosophy

Our *stepwise-refinement for performance* methodology addresses the tension between low-level programming that offers control and raising the level of abstraction. It gives programmers a trade-off between high-level programming which is good for portability and code maintainability,

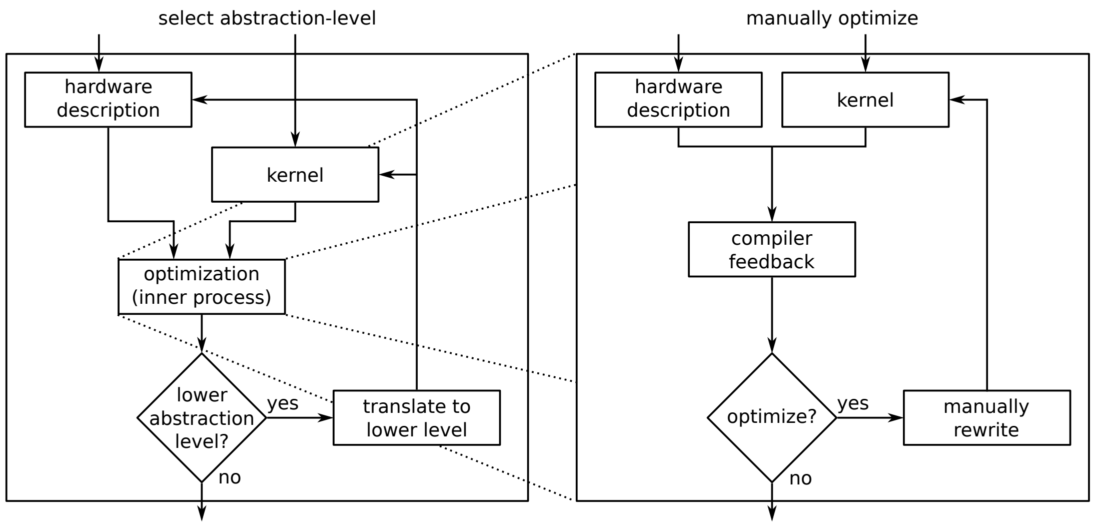


Figure 1. A diagram of the iterative processes. At the left is the outer process, at the right the inner process.

and low-level programming that gives programmers a clear, explicit, and well-defined interface to hardware features that are necessary for performance. To summarize, *stepwise-refinement for performance* provides programmers control and understanding on levels of abstraction they can choose themselves.

Unlike relying on black-box compilers that automatically optimize, our approach relies on cooperation between the compiler and programmer. The compiler is extended with knowledge about many-core architectures and combines this with the knowledge about the program to give detailed performance feedback. The programmer acts on the feedback and rewrites the program to obtain higher performance. This approach combines the strengths of both the compiler and programmer. The compiler does not have to be as conservative in providing feedback as it would normally be in applying transformations, whereas the programmer has application knowledge that the compiler lacks and gains an understanding of the performance of the program.

In each stage during the iterative process that our methodology advocates, a part of a program is rewritten to incorporate the feedback from the compiler. The different versions of the program that arise as a result of this process in effect capture the reasoning and knowledge about the optimizations.

3.2. Methodology

The *stepwise-refinement for performance* methodology contains two intertwined iterative processes where in each stage a program is rewritten to incorporate feedback from the compiler. The outer process centers around moving from a high abstraction-level to a lower abstraction-level. Hardware descriptions defined with different levels of detail effectively create multiple levels of abstraction.

Figure 1 shows that the outer process at the left has as input a hardware description and a computational kernel. With these inputs, the iterative inner process is started (discussed below). After this process has finished, the programmer has to decide whether to move to a lower level of abstraction to introduce more hardware details, or to stop, for example to remain portable among architectures.

If a programmer decides to move to a lower level of abstraction, the kernel has to be translated to a version that adheres to the rules of the lower-level hardware description. This process may be automated, but does not necessarily result in a kernel that has higher performance. It merely results in a kernel that represents the more detailed hardware features defined in the lower-level hardware description. The outer process restarts with this new kernel and its lower-level hardware description that serve as input for the inner process. If a programmer decides to not move to a lower level of abstraction, the outer process stops. The resulting kernel can now be further processed to be incorporated in an application.

The outer process gives programmers a trade-off in the level of abstraction for the inner process that focuses on optimizing the kernel. For example, there could be a hardware description that defines a generic GPU independent of vendor. The hardware description at the next level of abstraction could introduce hardware features that are vendor-specific. At this point, the programmer may decide to stop at this level to remain portable between GPU vendors.

The inner process at the right in Fig. 1 has as input a hardware description and a kernel and focuses on optimizing the kernel at the current level of abstraction. Instead of automatically optimizing the code, our approach gives the compiler a different role, namely providing the programmers with detailed performance feedback based on analysis of the kernel and the hardware description.

The role of the programmer is to decide whether the feedback is useful and can lead to better performance. If this is the case, the programmer manually rewrites the kernel to incorporate the feedback from the compiler. This rewritten kernel can then be reevaluated by the compiler to generate additional feedback. If the programmer decides that no further optimizations are necessary at this specific level of abstraction, the inner process stops after which the outer process continues giving the programmer an option to move to lower levels of abstraction.

4. DESIGN OF MCL

The previous section discussed our methodology without any implementation details. This section introduces Many-Core Levels (MCL) and discusses our choices in implementing the *stepwise-refinement for performance* methodology. MCL has been released as open source [1].

There are several consequences for programming systems that support the *stepwise-refinement for performance* methodology. First of all, compilers usually have a view of how hardware behaves, but to support this methodology, the compiler has to be able to interpret hardware descriptions with varying levels of detail to support the multiple abstraction-levels. Second, the compiler has to combine the knowledge from these hardware descriptions with the knowledge about the program to generate feedback for the programmer.

There are also consequences for the programming language. It has to show how the program is mapped to the hardware, so that the compiler can reason about the mapping. Moreover, the programming language may have to be more restricted to allow the compiler to give feedback. Finally, for the feedback to be useful, it has to be closely related to the code that the programmer wrote and not to already optimized machine code, as is traditionally done.

In MCL, we consider a program as a mapping of an algorithm to hardware. Since hardware-specific details are so important for performance, we make the mapping explicit in the programming model. Therefore, MCL introduces hardware descriptions in the programming model. The hardware descriptions are user-defined and can describe hardware with different levels of detail.

We can define two roles for our system: Programmers are the *users* of the system writing code in the programming language. The other role *describes hardware* in the hardware description language, ideally fulfilled by hardware vendors. However, the hardware descriptions can be adjusted by programmers to define new abstraction-levels, for example if programmers want a hardware description that focuses mainly on the memory hierarchy when working on data-intensive applications. MCL provides a library of predefined hardware descriptions. Before describing the hardware description language and the programming language, we first provide an overview of the interaction between the two languages.

4.1. Overview

MCL is solely targeted at writing computational kernels for many-core hardware. These kernels are often just a small part of a larger application. Since many-core hardware is highly parallel with complicated memory hierarchies, programmers need *programming abstractions* to control these hardware features. In Many-Core Programming Language (MCPL) parallelism is expressed in terms

```
 1 perfect void matmul(int n, int m, int p,
 2     main float[n, m] c,
 3     main float[n, p] a, main float[p, m] b) {
 4
 5   foreach(int i in n threads) {
 6     foreach (int j in m threads) {
 7       float sum = 0.0;
 8       for (int k = 0; k < p; k++) {
 9         sum += a[i, k] * b[k, j];
10       }
11       c[i, j] = sum;
12 } } }
```

Figure 2. A matrix multiplication program for hardware description *perfect*.

```
 1 parallelism hierarchy {
 2   memory_space main {
 3   }
 4   par_group threads {
 5     nr_units = unlimited;
 6     par_unit thread {
 7       memory_space reg {
 8         default;
 9 } } } }
10 device perfect {
11   mem;
12   ic;
13   cores;
14 }
15
16 memory mem {
17   space(main);
18   capacity = unlimited B;
19 }
20 execution_group cores {
21   nr_units = unlimited;
22   execution_unit core {
23     slots(thread, 1);
24 } }
25 interconnect ic {
26   connects(mem,cores.core[*]);
27   latency = 1 cycle;
28   bandwidth = unlimited bit/s;
29 }
30 ...
```

Figure 3. Part of the hardware description *perfect*.

of *units of parallelism* such as threads or vectors and it is possible to target specific memories by declaring variables to reside in specific *memory spaces*.

Figure 2 shows a matrix multiplication program written in MCPL. The foreach-loops on line 5 and 6 express parallelism in terms of parallelism units threads. The variables a, b, and c are declared to reside in memory space main (line 2 and 3). The units of parallelism and the memory-spaces are defined in the hardware description.

For each compute kernel, to select a level of abstraction, programmers indicate which specific hardware description they target. The function in Fig. 2 targets hardware description *perfect* (line 1). The targeted hardware description governs which memory spaces and units of parallelism are available to the programmer for a specific compute kernel. MCPL is explained in more detail in Sec. 4.3 and we refer to [45] for a complete description of the syntax.

A hardware description written in HDL has two distinct sections. The first section is called the *parallelism hierarchy* and defines the programming abstractions that are available to the programmer in terms of units of parallelism and memory spaces. Lines 1 to 9 in Fig. 3 show that a memory space main is defined (line 2) and that a par_group threads is defined (line 4) that can be used in foreach-statements.

The second part of the hardware descriptions defines the physical device. The main components are *execution units*, *memories*, and *interconnects* that connect the execution units and memories. The memories are associated with the memory spaces and the execution units with the units of parallelism. The next section explains HDL in greater detail, while we refer to [45] for a more thorough specification.

4.2. *Hardware Description Language HDL*

The main purpose of HDL is to describe hardware features, but only those that are important for performance. Hardware descriptions may vary in the level of detail to define the abstraction-levels.

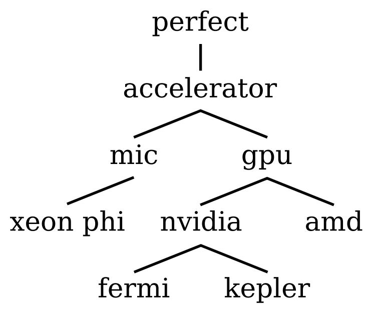


Figure 4. An example of a possible hierarchy of hardware descriptions. All hardware descriptions except *perfect* are user-defined.

Another goal is to formalize the knowledge that is often informally described in programming guides to make it accessible to both the programmer and the compiler, and to relate the hardware features with the program.

As shown in Fig. 4, hardware descriptions are organized in a hierarchy. The root is hardware description *perfect* that describes idealized many-core hardware and provides the highest level of abstraction for programmers. This description is provided in MCL and cannot be modified. Each lower level describes hardware in more detail and extends a parent resulting in the hierarchy.

Figure 3 shows a part of hardware description *perfect*. We first present an overview of what the hardware description expresses and then discuss the details. Hardware description *perfect* provides the programmer a flat parallelism hierarchy with only two memory spaces to consider. The memory mem is responsible for memory_space main (line 17) and has unlimited capacity (line 18). The device perfect has an unlimited amount of cores (line 20-22) and each core can run 1 thread (line 23). The memory is connected with all cores (line 26) and has 1 cycle latency (line 27) and unlimited bandwidth (line 28).

In HDL, the main syntactic construct to express hardware properties is a block with syntax:

```
Block          =  BlockKeyword Identifier "{" Statement* "}"
Statement      =  PropertyStatement
               |  Block
BlockKeyword   =  "parallelism"  |  "memory_space"  |  "par_unit"  |  ...
```

Hardware description *perfect* shows several of those blocks, for example on lines 1, 10, 16, 20, and 25. The semantics of a block are primarily determined by the specific BlockKeyword, such as parallelism, memory_space, or interconnect. Blocks contain a list of Statements that define some properties of the Block. A Statement can be a PropertyStatement or a nested Block. HDL defines precisely what kind of properties and nestings of Blocks are allowed. For example a parallelism block must have a memory_space block. A more complete description of the syntax and semantics is given in [45].

A valid hardware description with the name *name* has at least two blocks, a parallelism block (line 1 in Fig. 3) and a device block with *name* as Identifier (line 10). The parallelism block defines a hierarchy of programming abstractions and the device block describes the physical device.

Programming abstractions In a parallelism block the hardware description enforces which and how many units of parallelism programmers can use and to which memory spaces each unit of parallelism has access. This is achieved by nesting memory_space, par_group, and par_units in the parallelism block. The scope of the memory spaces defines which units of parallelism can make use of the memory space. For example, if a memory_space is defined within a par_unit, then this memory space is local to the par_unit.

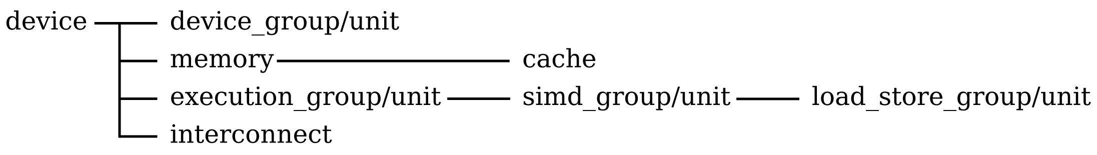


Figure 5. The device hierarchy.

Line 4 in Fig. 3 shows a par_group threads. Blocks ending with _group are special blocks that can group matching _unit blocks. The _group blocks must always contain a nr_units or max_nr_units property that expresses the allowed number of units.

Lines 1-9 show that the parallelism hierarchy hierarchy defines a memory_space main (line 2) and a par_group block threads (line 4) with an unlimited number of parallelism units par_unit thread (lines 4-6). This means that the programmer can use a memory space main in a program and can express parallelism with threads. The scoping of the memory_space blocks indicates that a thread has private access to reg, but that all par_units have access to memory_space main. The default property will be explained in Sec. 4.3.

Physical device Describing the physical device is achieved with device blocks that can be organized in a subclass hierarchy similar to a class hierarchy in Object-Oriented Programming. The hierarchy is shown in Fig. 5. The root is a generic device block. A device_group and device_unit are generic devices to make it possible to group devices.

A memory is a more specialized device of which we can describe, among other properties, its capacity. The memory blocks need to have a space property with an identifier as argument. The identifier has to refer to a memory_space in the parallelism hierarchy and indicates for which memory space the memory is responsible. For instance, in Fig. 3, the memory mem is responsible for memory_space main (line 17). A cache is a specialized memory. It inherits all the properties of memory, but it is also necessary to specify the cache-line size.

Execution units can be described by the various execution_group/unit blocks. Execution units execute instructions that are defined in the instruction block (not shown in Fig. 3). The block execution_unit describes a generic execution unit. A simd_group describes an execution unit that executes its instructions in lock-step: each simd_unit executes the same instruction in parallel. A load_store_group is a simd_group that only executes the two instructions “load” and “store”. This execution unit is responsible for moving data between the memories and each hardware description must have a load_store_unit (not shown in Fig. 3).

Execution groups and units must have a property slots. This property takes as argument an identifier and an integer expression. The identifier has to refer to a par_unit and the integer expression defines how many “slots” or “contexts” are available for an execution unit or group, which indicates how much parallelism is available on an execution unit. For example, if an execution group has 4 execution units and 8 slots for threads, this would mean that 2 threads have to share the time on each execution unit and have to be scheduled after each other. Scheduling cannot be further expressed in HDL.

An interconnect block describes a device that connects the various device blocks with each other. It is possible to specify the latency, the bandwidth or the width of the bus (in bits). The connects property specifies how execution units and memories are connected to each other. It takes two identifiers as arguments with possibly a [*] suffix. The suffix [*] has to be used for a device in a _unit in a _group and means that the connection leads to all units.

Figure 3 shows that the device perfect on line 10 holds a memory mem, an interconnect ic and an execution_group cores. A statement with solely an identifier means that this specific statement can be replaced with the block it refers to. This mechanism also supports inheriting blocks from other hardware descriptions, but in this case the identifiers have to be fully qualified. For example, a hardware description may reuse the par_group threads from hardware description *perfect*. It then has to refer to this block with the qualified identifier perfect.hierarchy.threads.

4.3. Programming Language MCPL

To provide a familiar interface to many-core programmers, MCPL is imperative and C-like. In contrast to C, MCPL has constructs to explicitly map the program to hardware using the programming abstractions defined in the parallelism block in a hardware description. We describe the syntax in more detail in [45].

Figure 2 shows matrix multiplication written for hardware description *perfect*. On line 1, the function matmul() is declared to adhere to the rules of hardware description *perfect*. Thus, it is possible to specify a hardware description per function. The hardware description determines which memory spaces are available and how parallelism can be expressed. MCPL has a foreach construct that expresses parallelism in terms of units of parallelism. In Fig. 2 on lines 5 and 6, the units of parallelism are par_group threads (defined on line 4 in Fig. 3). Barrier statements (not shown in Fig. 3) and declarations are associated with memory spaces. Figure 2 shows that variables c, a, and b are declared to reside in memory space main (lines 2 and 3).

In an MCPL program, the compiler can automatically infer which variables are constant or written. In an MCPL module, the whole call hierarchy has to be fully specified. Library functions are annotated with signatures that specify which of the parameters are written and/or read.

Variables that have an array type or that are written need to have a memory space modifier unless the hardware description has the property default (line 8 in Fig. 3 for reg). Thus, sum (line 7 in Fig. 2) is implicitly declared to reside in memory space reg. Variables are passed by reference unless types are primitive and constant. Aliasing is restricted and MCPL has no global variables.

MCPL provides multi-dimensional arrays with array-size specifiers. This helps the compiler to reason about different dimensions and ensures that the compiler can express feedback in terms of array-sizes. Variables that are used in array-size specifications have to be constant. Figure 2 shows on lines 2 and 3 variables c, a, and b being declared as two-dimensional arrays.

MCPL supports views and tiles in multiple dimensions on arrays. For example, an array int[36] d can be declared as int[2,2][3,3] d2 with 2×2 tiles of 3×3 elements. This helps programmers to specify data-layout. More details are provided in [45].

A foreach-statement declares an indexing variable, contains a dimension expression and an identifier that indicates which units of parallelism the foreach targets. Line 5 in Fig. 2 states that there will be n units of parallelism of par_group threads each with a unique index i where $0 \leq i < n$. The dimension expressions in foreach-statements are not allowed to depend on the index of an outer foreach.

The parallelism block in the hardware description governs how foreach loops can be nested and where memory spaces can be declared. The nesting has to follow the nesting in the hardware description with an exception for foreach loops belonging to the same par_group, which are allowed to be nested (as shown on lines 5 and 6 in Fig. 2). Executing statements are only allowed in the innermost foreach loop. For a declaration of a variable, its nesting must match the nesting of its memory_space in the hardware description.

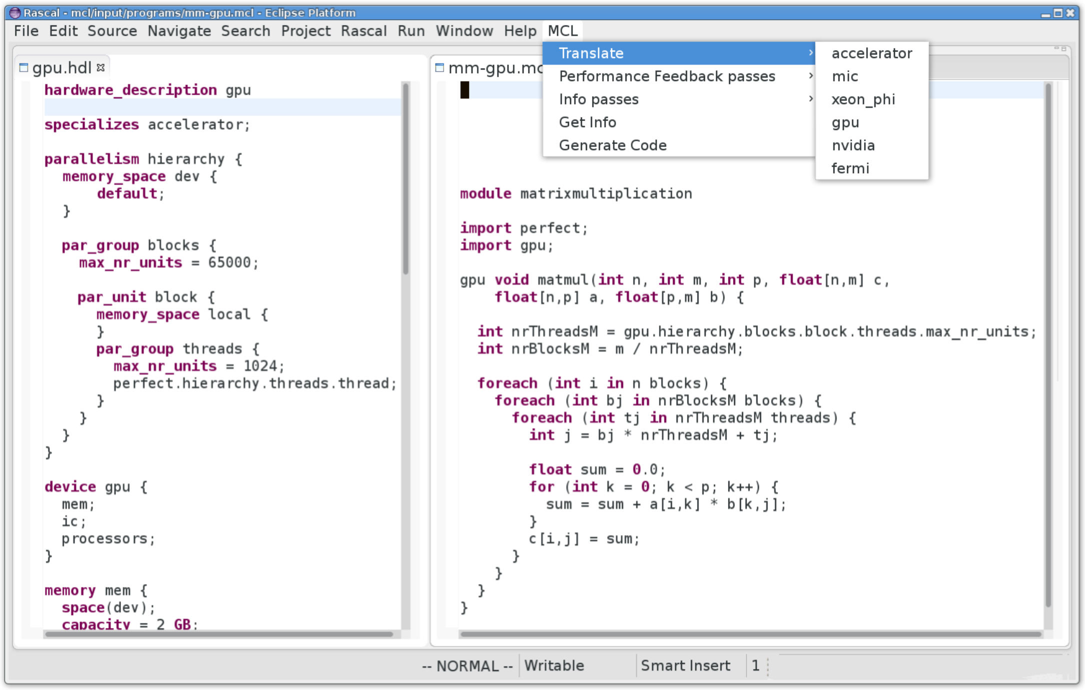
Figure 6 explains the nesting rules in more detail. It shows at the left a possible nesting of par_groups in a parallelism block inside a hardware description. The right side shows a possible nesting of foreach statements. The nesting follows the nesting in the hardware description and the figure shows that foreach statements of the same par_group can be nested. At the right side there is a declaration a with memory space main. The declaration is inside a foreach-loop with par_group blocks, which means that the declaration is inside a block. The left side shows that there is indeed a memory space main defined inside a block par_unit.

Units of parallelism can communicate with each other through memory that is declared in a scope outside of the units of parallelism. For example, in Fig. 6, threads cannot communicate with each other using memory space reg, but they can communicate with each other using memory space main. To ensure that a thread can read what another thread wrote, it is possible to insert a barrier(main) statement, with a parameter for the memory space.

```
par_group blocks {
  par_unit block {
    memory_space main {}
    par_group threads {
      par_unit thread {
        memory_space reg {}
} } } }
```

```
foreach (int ib in nb blocks) {
  foreach (int jb in mb blocks) {
    main int[n] a;
    foreach (int it in nt threads) {
      foreach (int jt in mt threads) {
} } } }
```

Figure 6. Nesting of memory spaces and units of parallelism in relation to nesting of foreach statements and declarations.


Rascal - mcl/input/programs/mm-gpu.mcl - Eclipse Platform
File Edit Source Navigate Search Project Rascal Run Window Help MCL
Translate
Performance Feedback passes
Info passes
Get Info
Generate Code
accelerator
mic
xeon_phi
gpu
nvidia
fermi
gpu.hdl
mm-gpu.mcl
-- NORMAL -- Writable Smart Insert 1

```
hardware_description gpu

specializes accelerator;

parallelism hierarchy {
  memory_space dev {
      default;
  }

  par_group blocks {
    max_nr_units = 65000;

    par_unit block {
      memory_space local {
      }
      par_group threads {
        max_nr_units = 1024;
        perfect.hierarchy.threads.thread;
      }
    }
  }
}

device gpu {
  mem;
  ic;
  processors;
}

memory mem {
  space(dev);
  capacity = 2 GB;
```

```
module matrixmultiplication

import perfect;
import gpu;

gpu void matmul(int n, int m, int p, float[n,m] c,
    float[n,p] a, float[p,m] b) {

  int nrThreadsM = gpu.hierarchy.blocks.block.threads.max_nr_units;
  int nrBlocksM = m / nrThreadsM;

  foreach (int i in n blocks) {
    foreach (int bj in nrBlocksM blocks) {
      foreach (int tj in nrThreadsM threads) {
        int j = bj * nrThreadsM + tj;

        float sum = 0.0;
        for (int k = 0; k < p; k++) {
          sum = sum + a[i,k] * b[k,j];
        }
        c[i,j] = sum;
      }
    }
  }
}
```

Figure 7. Screenshot of the MCL Eclipse Plugin.

4.4. *Compiler*

To support the *stepwise-refinement for performance* methodology, the compiler is not only responsible for generating code, but also for generating performance feedback for the programmer. MCL has been implemented in meta-programming language Rascal [46] which allows us to create an Eclipse Plugin with feedback annotations and syntax highlighting for both HDL and MCPL. Figure 7 shows a screenshot of the Eclipse Plugin with a menu for activating several feedback passes in the compiler. The programmer receives messages from the compiler associated directly with the source code, for example messages for a specific variable, foreach-loop, or function.

Another task of the compiler is to automatically translate between the abstraction-levels. Programs written for hardware x can automatically be translated to hardware description y if y is a child of x in the hierarchy in Fig. 4. The edges in the hierarchy define the translation possibilities, so it is for example possible to translate from *perfect* to *gpu*.

Finally, the compiler can generate OpenCL and C++ code from the leaf nodes of Fig. 4. For example, the compiler takes as input a function written for hardware description *fermi*. It reads in the hardware description *fermi* but also a configuration file that tells the compiler how the programming abstractions defined in hardware description *fermi* can be mapped to OpenCL and C++ constructs. Our system generates OpenCL code for the computational kernels, header files, and C++ code for setting up the execution on a many-core device. This means that the generated code can easily be included in many applications.

As the compiler automatically translates between abstraction-levels and can generate code from each leaf node, it can generate code for each abstraction level. For example, following the hierarchy of hardware descriptions in Fig. 4, we can generate code for *fermi* from a program written in *gpu*. First, the compiler translates the program to *nvidia*, then it translates to *fermi*, and from there the compiler generates OpenCL and C++ code.

This section presented an overview of the MCL programming system. Section 5 will give an example of the *stepwise-refinement for performance* methodology using our system. Section 6 will discuss several interesting implementation details.

5. EXAMPLE: MATRIX MULTIPLICATION

In this section we give a detailed overview of how the process of the *stepwise-refinement for performance* methodology took place for matrix multiplication. We refined the kernel in a stepwise manner to obtain high performance for two different many-core architectures: a GTX480 GPU and a 60-core Intel Xeon Phi.

Table II gives an overview of the feedback that the we received on each level, what refinements we applied, and the performance of each kernel. Below we will discuss the details. From level *accelerator* we will split in Sec. 5.1 for the GTX480 and Sec. 5.2 for the Xeon Phi. How the compiler translates and gives feedback is explained in Sec. 6. In Sec. 7 we will evaluate our approach with more applications.

perfect The goal of hardware description *perfect* is to define a high level of abstraction by making the many-core hardware as simple as possible. Usually, many-cores expose multiple layers of parallelism and several memories, but to create a high level of abstraction, hardware description *perfect* has only one layer of parallelism and two memory spaces (Fig. 3).

Figure 2 shows matrix multiplication written for this hardware. Since programmers write for idealized hardware, they do not have to pay attention to hardware-specific details. At this level of abstraction, it is no problem to create $n \times m$ threads or read the same data multiple times, because the hardware description defines that we have an unlimited number of execution cores and an access time of 1 cycle to data in main memory.

In the inner process, the compiler is not triggered to give optimization feedback on this level of abstraction, because the properties of the device do not give rise to this. For example, there is no faster memory. However, we can choose to investigate the algorithmic properties of the program. For instance, we can request statistics on the number of operations as shown in Table II.

Matrix multiplication written for *perfect* is a portable program. We first automatically translate this program down the hierarchy in Fig. 4 via *accelerator*, *gpu*, and *nvidia* to *fermi* to generate code. Running this on an NVIDIA GTX480 GPU delivers 89.0 GFLOPS. Automatically translating the program to *xeon_phi* (via *accelerator* and *mic*) and generating code from there results in a performance of 39.9 GFLOPS. Since the inner process does not give feedback that we can use to optimize, we move to a lower level of abstraction in the outer process. We let the compiler automatically translate the program to hardware description *accelerator* below *perfect* in Fig. 4.

accelerator In the library of predefined hardware descriptions, an accelerator is defined to be a many-core device that is connected to a host computer by means of a PCI-express bus, a common setup for many-core hardware. The *accelerator* hardware description specializes hardware description *perfect*. It inherits most features from *perfect*, but a special device host is added that is responsible for setting up the computation for the many-core machine. We define an interconnect pcie in a similar way as the interconnect ic in Fig. 3 on line 19, but now it connects the memory accelerator.mem and the device host and has different values for the latency and bandwidth.

Since the parallelism hierarchies of *perfect* and *accelerator* are the same, the translation pass of the compiler only has to change the keyword perfect on line 1 to accelerator to let the program adhere to the rules of hardware description *accelerator*.

Table II. Feedback from compiler on different abstraction-levels for matrix multiplication. The versions with (v*) are modified by the programmer according to the feedback from the compiler.

Version	Change compared to previous version	Feedback	Performance
NVIDIA GTX480 GPU			
perfect	(initial version)	#computational instructions: $2nmp$, #data accesses: $2nmp + nm$	89.0 GFLOPS
accelerator	(auto-translate)	PCIe transfers: $4np + 4mp + 4nm$ bytes	
gpu	(auto-translate)	expression a[i,k] loaded for each thread t: local memory can be leveraged	
gpu (v1)	Use local memory for p elements	consider computing multiple elements per threads	100 GFLOPS
gpu (v2)	Compute multiple elements per thread	expression b[k, j] loaded for each block and each iteration in for-loop (n times)	91.6 GFLOPS
gpu (v3)	pull b[k, j] out of for-loop	expression b[k, j] loaded for each block b1 (n/x times)	205 GFLOPS
nvidia	(auto-translate)	Using 1/8 blocks per SMP. Reduce the amount of shared memory used by storing/loading shared memory in phases	
nvidia (v1)	multiple store-load phases	Using 4/8 blocks per SMP.	489 GFLOPS
fermi	(auto-translate)	Optimal memory access. b[k, j] in loop bi does not benefit from cache, best case: 256 cache-line fetches	
fermi (v1)	compute more elements for b[k, j]	b[k, j] in loop bi does not benefit from cache, best case: 128 cache-line fetches	564 GFLOPS
Intel Xeon Phi (5110P)			
perfect	(same version as above)	(same feedback as above)	39.9 GFLOPS
accelerator	(same version as above)	(same feedback as above)	
mic	(auto-translate)	Consider computing multiple elements per threads	
mic (v1)	Compute multiple elements per thread	Data reuse: a[i, k] accessed in loop with index ej but does not depend on it.	35.7 GFLOPS
mic (v2)	Pull a[i, k] out of for-loop		47.6 GFLOPS
xeon_phi	(auto-translate)	b[k, j] in loop k does not benefit from cache, p cache-line fetches	
xeon_phi (v1)	Pull b[k, j] out of for-loop	c[i, j] in loop in loop ei does not benefit from cache, 32 cache-line fetches	87.8 GFLOPS
xeon_phi (v2)	c in temporary variable	Try to adjust size bTemp with size 4 * p bytes, to cache (32 kB total, cache_line 64 B)	115 GFLOPS
xeon_phi (v3)	Decrease bTemp		488 GFLOPS

At this level of abstraction, we receive feedback that the program has $4nm$ bytes of data transfers from device to host, and a data transfer of $4np$ bytes plus a data transfer of $4pm$ from host to device. The analysis to provide this feedback infers that variables a and b are read and that c is written inside the foreach loops. The compiler can give this feedback because the sizes of a, b, and c are known in terms of parameters n, m, and p.

At this level of abstraction and with this feedback, we have no opportunity to optimize the existing program. Therefore, we move to lower level hardware descriptions giving up portability between GPU and Xeon Phi. The next subsection will discuss the process for the GTX480, while Sec. 5.2 discusses the process for the Xeon Phi.

5.1. GTX480

gpu At this level of abstraction, the many-core device becomes more concrete. Representative values obtained from programming guides are filled in to give the compiler an indication of the amount of parallelism and the amount of memory to define a generic GPU. For example, we could

```
 1 parallelism hierarchy {
 2   memory_space dev {
 3   }
 4   par_group blocks {
 5     max_nr_units = 65535;
 6     par_unit block {
 7       memory_space local {
 8       }
 9       par_group threads {
10         max_nr_units = 1024;
11         perfect.hierarchy.
12           threads.thread;
13 } } } }
14
15 memory on_chip {
16   space(local);
17   capacity = 16 kB;
18 }
19 execution_group processors {
20   nr_units = 16;
21   execution_unit processor {
22     slots(block, 1);
23     on_chip;
24     execution_group alus {
25       nr_units = 32;
26       execution_unit alu {
27         slots(thread, 1);
28 } } } }
```

Figure 8. Part of the hardware description *gpu*.

```
 1 int nrThreadsM = gpu.hierarchy.blocks.block.
 2   threads.max_nr_units;
 3 int nrBlocksM = m / nrThreadsM;
 4
 5 foreach(int bi in n blocks) {
 6   foreach (int bj in nrBlocksM blocks) {
 7     foreach (int tj in nrThreadM threads) {
 8       int i = bi;
 9       int j = bj * nrThreadsM + tj;
10
11       float sum = 0.0;
12       for (int k = 0; k < p; k++) {
13         sum += a[i, k] * b[k, j];
14       }
15       c[i, j] = sum;
16 } } }
```

Figure 9. Transformation of the loops on level *gpu*

define that a *gpu* has a device memory of several GBs, 16 processors, and each processor has 32 small ALUs capable of performing floating point operations and a fast on-chip memory.

Conceptually, a GPU exposes two layers of parallelism to the programmer. The 32 ALUs can run threads in parallel and the 16 processors can run blocks of threads in parallel. Figure 8 shows how this is reflected in the parallelism hierarchy. Lines 4 to 13 express that there are two layers of parallelism. Programmers can define up to 65535 blocks to run in parallel. Outside the block scope there is a memory space dev that is shared by blocks. Inside the scope of a block there is a memory space local. It is not possible to synchronize between blocks using this memory because it is not declared in a scope that surrounds blocks. However, threads can synchronize using this memory because the group of parallelism units threads is defined in the same scope. Per block, programmers can use up to 1024 threads (line 10). On line 11 and 12, this hardware description inherits the unit of parallelism thread from hardware description *perfect*.

To automatically translate matrix multiplication to *gpu*, the compiler has to split the foreach loops for parallelism units threads to multiple foreach loops for parallelism units blocks and threads. The translation process is explained in Sec. 6, the result is listed in Fig. 9 (assuming that m can be divided by nrThreadsM for simplicity). The code from line 11 on remains unchanged compared to the code in Fig. 2.

Arrived in the inner process (Fig. 1), the compiler combines the knowledge from the hardware description and the program to give the following three very specific feedback messages: (1) "expression a[i, k] loaded for nrThreadsM threads tj: local memory can be leveraged.", (2) "expression b[k, j] is accessed for n blocks bi.", and (3) "It may be beneficial to consider computing more than one element of c per thread."

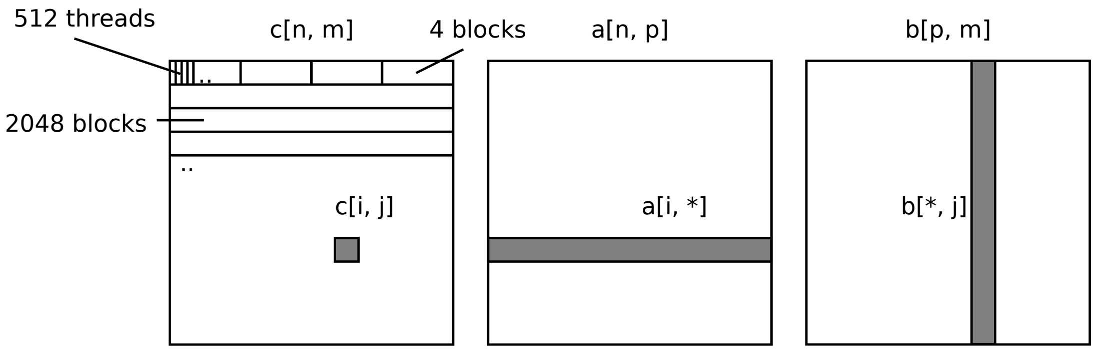


Figure 10. Schematic view from block division in a 2048×2048 matrix multiplication.

The compiler issues the first two messages after analyzing data reuse. Using data-flow analysis, it discovers that expression $\mathrm{a[i,\ k]}$ does not depend on tj and bj. Figure 10 shows a schematic view of how the blocks and threads defined in Fig. 9 use the input and output data. Each of the 512 threads (identified by tj) computes a single output element in a row in array c. To compute one element, a thread tj (and a block bj) needs access to a complete row in the a matrix. The compiler discovers this by concluding that expression $\mathrm{a[i,\ k]}$ does not depend on variable bj or tj. Similarly, the compiler notices that expression $\mathrm{b[k,\ j]}$ does not depend on bi. Therefore, the compiler issues the messages that $\mathrm{a[i,\ k]}$ will be loaded for each thread tj, each block bj, and that $\mathrm{b[k,\ j]}$ will be loaded for each block bi.

Additionally, the compiler combines this information with the information in the hardware description. In Fig. 8 on lines 7 to 13, $\mathrm{par\_group\ threads}$ shares a memory space local, which means that each thread tj can access this memory. The compiler investigates whether this memory space resides in faster memory than the memory of array a (memory space dev). Hardware description *gpu* shows that there is a memory $\mathrm{on\_chip}$ that holds memory space local (lines 15 to 18 in Fig. 8). The compiler can infer that this memory is faster than the memory that holds memory space dev (not shown). Since local memory is shared between threads tj and is faster than dev, the compiler can suggest to use memory space local for each thread tj for expression $\mathrm{a[i,\ k]}$.

The feedback that we receive in the inner process is now sufficient to act upon. We decide to first listen to the most specific feedback message, which is feedback message (1). We implement the suggested change in version *gpu (v1)* by loading p elements in local memory.

gpu (v1) In this version we want to load p elements of $\mathrm{a[i,\ *]}$ in local memory. To accomplish this, we add a declaration with memory space local, a for-loop that loads p elements in cooperation with the threads, and a $\mathrm{barrier(local)}$ statement to make sure that each thread can observe all the updates from the other threads. This is a standard technique in GPU programming.

The resulting version delivers 100 GFLOPS. The feedback we acted on disappears, but the other two messages remain. We decide to take another iteration in the inner process. Since it is not clear at this stage how to resolve the issue in feedback message (2), we decide to act on feedback message (3) by computing multiple elements per thread.

gpu (v2) We use this version to illustrate that although feedback from the compiler may not immediately lead to better performance, the feedback is not bad per se and can ultimately lead to better performance if we listen to the compiler carefully. In contrast to our structured approach to optimizing, in ad-hoc optimization, programmers might disregard this opportunity for higher performance because initially it leads to lower performance.

We implement computing multiple elements in a column per thread, but we do this naively by splitting the $\mathrm{foreach}$-loop bi into two. We decide to change the loop having n iterations into a $\mathrm{foreach}$-loop of n/x iterations, where x is a value that we chose. We add a sequential for-loop of x

```
local float[x][p] l_a;
...
// load x * p elements in l_a
...
for (int k = 0; k < p; k++) {
  float bkj = b[k, j];
  for (int ei = 0; ei < x; ei++) {
    sums[ei] += l_a[ei][k] * bkj;
} }
```

Figure 11. Structure of the loops in *gpu (v3)*.

```
shared float[x][nrThreadsM] l_a;
...
for (int l = 0; l < y; l++) {
  for (int ei = 0; ei < x; ei++) {
    l_a[ei][tj] = a2[bi,l][ei,tj];
  }
  barrier(shared);

  for (int k2 = 0; k2 < p/y; k2++) {
    int k = l * p/y + k2;
    float bkj = b[k,j];

    for (int ei = 0; ei < x; ei++) {
      sums[ei] += l_a[ei][k2] * bkj;
  } }
  barrier(shared);
}
```

Figure 12. Structure of the loops in *nvidia (v1)*.

iterations in the body of the foreach-statements. (This technique is commonly referred to as vertical thread-block merge in GPU programming.)

Indeed feedback message (3) disappears. However, feedback message (2) is repeated in a slightly different form and gives us a deeper understanding in the performance issue. The compiler repeats that expression b[k, j] does not depend on each block bi (but now for n/x blocks), and that it is also independent of the for-loop which we have just introduced having x iterations. This means that effectively, since b[k, j] is independent of both loops, it is still loaded n times and that we have only added control flow, compared to the previous version. This explains the worse performance.

Fortunately, this steers us in the right direction: since, b[k, j] is independent of the for-loop, we now understand that we have to try to pull expression b[k, j] outside of this for-loop to reuse the value for multiple iterations. We implement this in version *gpu (v3)*.

gpu (v3) It is more challenging to incorporate the feedback in this version. However, if programmers fail to act on the feedback, at least they are informed of the performance issue.

The insight for this version is that we want to move the for-loop with x iterations inside the loop on line 12 in Fig. 9 so we can reuse b[k, j] for each of the x iterations. Unfortunately, this is not immediately possible, since loading the elements in local memory introduced in *gpu(v1)* also relies on this loop. The solution is to use more local memory and load these elements in a second, separate for-loop of x iterations. Figure 11 shows the structure of the loop after rewriting.

This version delivers 205 GFLOPS. For programmers that want to remain portable between AMD and NVIDIA GPUs and are satisfied with the obtained performance, this is a good moment to stop optimizing since the hardware descriptions below *gpu* contain vendor-specific information. Therefore, this optimization applies to both AMD and NVIDIA GPUs. This clearly shows that MCL offers programmers a trade-off in high-level programming, portability, and performance. We decide to stop optimizing in the inner process and move to level *nvidia* in the outer process.

nvidia The parallelism hierarchy on level *nvidia* is almost the same as on level *gpu*, but memory space dev is renamed to global and memory space local is renamed to shared to conform to NVIDIA terminology. As a result of more specific details in the hardware description, the compiler triggers an analysis that requests example values for m, n, and p from the programmer. It then tells us that memory on_chip is a scarce resource and that we should carefully consider the usage of memory space shared, since we use this in our program. The compiler is even more specific. It tells us that we execute only 1 out of 8 blocks per SMP in parallel which depends on the number of threads, the number of registers, and the amount of shared memory we use. Finally, it proposes a strategy to manage shared memory, namely by reducing the amount of shared memory by using multiple store-load phases.

```
 1  execution_group smps {
 2    nr_units = 16;
 3    execution_unit smp {
 4      slots(block, 8);
 5      slots(thread, 1024);
 6      performance_feedback(
 7        "shMem");
 8      on_chip;
 9      regs;
10      gpu.processors.
11        processor.alus;
12  } } } }
```

Figure 13. Part of the hardware description *nvidia*.

```
 1  String shMem(Kernel k) {
 2    int usedShared = k.context.
 3      sizeMemSpace("shared");
 4    if (usedShared == 0) {
 5      return "";
 6    }
 7    else {
 8      int nrSlotsBlock = smps.
 9        smp.nrSlots("block");
10      ...
11  } }
```

Figure 14. Part of user-defined performance function used in hardware description *nvidia*.

To be able to give this feedback, the compiler combines many pieces of information from the hardware description. Figure 13 shows a part of hardware description *nvidia*. It defines that an SMP has 8 slots for blocks (line 4). It also defines that an execution-unit smp contains a memory on_chip (line 8) that holds memory space shared (not shown). Because of what is defined in the parallelism_hierarchy block, the compiler also knows that blocks cannot communicate through memory space shared (similar to the situation in *gpu* where blocks cannot communicate through local in Fig. 8 on line 7). Therefore, the compiler knows that memory on_chip has to be divided among 8 blocks, which makes memory on_chip a scarce resource.

On an NVIDIA GPU, the maximum number of blocks that can run on an SMP (between 1 and 8) is important but governed by complex rules. Therefore, MCL provides an API in which these complex rules can be encoded. Lines 6 and 7 in Fig. 13 point the compiler to a performance feedback function "shMem" that expresses these complex rules. Part of the feedback function is shown in Fig. 14. The compiler exposes an API in Java through a data-structure Kernel that we can ask for the size of memory spaces (lines 2 and 3), and the number of slots (line 8 and 9).

Because of the feedback, we understand that we have to minimize the amount of allocated shared memory to run more than one block on an SMP. In version *nvidia (v1)* we implement the suggestion from the compiler to load the elements in multiple load-store phases.

nvidia (v1) It may not be immediately clear how to implement this optimization. In the previous versions we loaded p elements cooperatively with threads in shared memory (or local in hardware description *gpu*) which happens in y iterations. To create multiple store-load phases, we make this loop the outer loop moving the loading of elements and the computation inside. The insight for this optimization is that we can merge the loop on line 4 in Fig. 11 with the loop of y iterations. The resulting loop structure is shown in Fig. 12. As a result of this optimization we use y times less shared memory.

We now receive the feedback that we use 4 out of 8 blocks which gives us a performance of 489 GFLOPS. We translate this version to level *fermi*.

fermi Hardware description *fermi* introduces a new layer warp in the parallelism hierarchy, as well as SIMD-units and caches with cache-line sizes. The difference between a normal execution_group and a simd_group is that the simd_units are defined to execute in lock-step, so each unit performs the same instruction in parallel. In the *fermi* hardware description, it is defined that a warp of 32 threads is executed on a SIMD-unit, which means that each thread executes in lock-step with the others.

Using a performance feedback function similar to the one of Fig. 14, the compiler can give feedback on the complex memory-access rules of the *fermi* architecture. It tells us that the memory access patterns of a, b, and c are optimal, and that there are no bank conflicts in the access of shared memory.

```
 1  parallelism hierarchy {
 2    memory_space dev {
 3    }
 4    par_group threads {
 5      max_nr_units = unlimited;
 6      par_unit thread {
 7        par_group vectors {
 8          nr_units = 16;
 9          par_unit vector {
10            memory_space reg {
11  } } } } } }
12  interconnect ring {
13    connects(mem,cores.core[*]);
14    latency = 20 cycles;
15  }
16  execution_group cores {
17    nr_units = 60;
18    execution_unit core {
19      slots(thread, 4);
20      regs;
21      vector_group;
22  } }
```

Figure 15. Part of the hardware description *mic*.

The new information about SIMD-units and how warps execute in lock-step in the hardware description is used in the cache analysis. If a thread in a warp can access an element in memory that is cached, then the other threads in a warp also have access to the cache-line and do not need a new cache-line fetch. The cache feedback gives us a worst case scenario and a best case scenario in terms of the number of cache line fetches. This makes clear that in a specific loop, access b[k, j] needs at least 256 cache-line fetches.

We decide to try to improve the cache behavior of the memory access of b[k, j] in version *fermi (v1)*.

fermi (v1) In this version we tweak the parameters x and y such that we compute twice the amount of elements for each fetch of b[k, j]. This gives us a performance of 564 GFLOPS.

At this point the compiler gives us no more feedback we can act on. We are also on the lowest level of abstraction, which ends the *stepwise-refinement for performance* process for the GTX480.

5.2. Xeon Phi

mic Intel's Many Integrated Core (MIC) architecture contains several tens of in-order x86 cores with powerful vector units and several hardware threads connected through a ring network. The MIC exposes two layers of parallelism: vector instructions and threads. In the library of hardware descriptions, the *mic* hardware description represents this as shown in Fig. 15. Besides registers (line 10), there is only one memory space dev (line 2). It is possible to define an unlimited amount of threads (line 5) and the vector unit is exposed through a par_group of 16 vectors (lines 7 to 11). The *mic* has a ring interconnect that connects the memory to all cores (line 13). A representative value for the number of cores is 60 (line 17). Each core has slots for 4 hardware threads (line 19), registers (line 20) and a simd_unit called vector_group that has slots for the vector par_units defined on lines 9 to 11).

After automatically translating the *accelerator* version to *mic*, the compiler tells us about data-reuse in variables a and b, similar to the feedback on level gpu. The compiler does not propose a faster memory because the *mic* hardware description does not expose faster memories. However, since the number of execution units is limited, it does propose to compute multiple elements per vector.

We implement this in a new iteration of the inner process. We decide to compute multiple elements in both dimensions in version *mic (v1)* as it is not clear what dimension we should compute multiple elements in.

mic (v1) In this version we naively introduce a for-loop for each dimension: a for-loop with index ei and a for-loop with index ej. For the same reasons as in version *gpu (v2)*, we get lower performance (35.7 instead of 39.9 GFLOPS), but again this leads to feedback that can help us to gain more performance.

The compiler tells us that a[i, k] is accessed in a loop that we have introduced with index ej, but that it does not depend on it. Similarly, it tells us that b[k, j] does not depend on the loop we have introduced with index ei.

Because of the way the loops are organized, it is difficult to do something about both feedback messages at the same time. Therefore, we decide to choose one of the two variables and pull a[i, k] out of the for-loop with index ej and store it in a temporary array to increase data-reuse. This optimization is comparable to the one we did in version *gpu (v1)*.

mic (v2) Having implemented the above in this version, we still get feedback for data-reuse for b, but similar to the previous version, it is not clear how to act on the feedback. We measure the performance at 47.6 GFLOPS and decide to automatically translate this version to *xeon_phi*.

xeon_phi The *xeon_phi* hardware description introduces many more details compared to *mic*. The most important difference is the introduction of level 1 and 2 caches. The hardware description language is expressive enough to give a realistic representation of the Xeon Phi. The ring interconnect connects the memory to 60 L2 caches of 512 kB. Each cache is associated with an execution unit core that holds 32 kB of L1 cache, which is in accordance with the real hardware. Both caches have cache lines of 64 bytes.

At the *xeon_phi* level, we receive feedback that we did not receive for GPUs, namely: “This is a cache-oriented architecture. Make sure that each access benefits from the cache.” The compiler gives this feedback if it learns from the hardware description that there are no other opportunities for fast memory access than caches, which is the case on a Xeon Phi. For every variable declared in dev memory space we get cache-behavior feedback (explained in Sec. 6). We learn that variable a does benefit from the cache, but that b and c do not benefit from the cache. The compiler tells us that b has the worst behavior, namely at least p cache fetches.

We now realize that on level mic we pulled the wrong array out of the for-loop, since array a has better cache behavior than b. We decide to undo pulling out a and pull out b instead of a, making sure that we compute as much as possible for each cache-line fetch for b. This leads to version *xeon_phi (v1)*.

xeon_phi (v1) This change almost doubles the performance to 87.8 GFLOPS. The compiler gives us two feedback messages: (1) “c may benefit from the cache: best case $2m$ cache-line fetches.”, (2) “try to adjust the size of declaration bTemp with $4p$ bytes in relation to the cache with capacities 32kB and 512kB and cache-line size 64B.”, where bTemp is the temporary variable we have just introduced.

Feedback message (1) tells us that we have bad cache behavior for variable c because m is typically large. A solution would be to store this in a temporary array. Investigating feedback message (2), we come to the conclusion that if we split up bTemp, then we have to store partial results for c for which we need temporary storage as well. Therefore, we decide to act on feedback message (1) first, which leads to the next version.

xeon_phi (v2) Measuring the performance for this version where we introduced temporary storage for variable c gives 115 GFLOPS. We decide now to act on feedback message (2) to reduce the size of bTemp to better fit the caches. This means that we need to compute partial results in the temporary variable of c we have just introduced. We implement this in version *xeon_phi (v3)*.

xeon_phi (v3) This has a dramatic effect on performance. Making the array small, for example a value of 16 elements for the bTemp array leads to a performance of 488 GFLOPS. The compiler gives us no longer feedback we can act on, so we stop the process.

5.3. Summary

In the process above, most optimizations are straightforward to do. However, several optimizations may require considerable thought: From *gpu (v2)* to *gpu (v3)*, the compiler complains that expression b[k, j] is loaded many times. To solve it, we allocate more local memory which may be counter-intuitive for some programmers. When more hardware details become available, we reduce this size in *nvidia (v1)*, which may also be a challenging optimization. The challenge here is to reuse the iteration space of one loop (line 3 in Fig. 12) for another (line 9). Finally, in *xeon_phi (v1)* it may not be easy to discover how to decrease the size of the bTemp array.

The benefit of our approach compared to a trial-and-error process is that our compiler assists and steers the programmer. The compiler helps the programmer to focus on aspects of the many-core hardware that the compiler deems appropriate at a certain stage of the program. Moving to lower levels of abstraction, the compiler gains more and more insight in the hardware. This is in accordance with the increasing insight that programmers gain while creating the different versions. Another benefit is that the hardware descriptions form a well-defined set of hardware features for different architectures which makes the trade-off between portability against performance more clear.

6. IMPLEMENTATION

This section discusses several implementation details of MCL. We focus primarily on the techniques that use both the information from the hardware description and the program for providing high-quality feedback. The analysis techniques work directly on the abstract syntax tree to create a strong relation with the actual code that the programmer wrote.

6.1. Translation between Abstraction Levels

The input for the translation pass is an MCPL module with functions and a string *target* indicating the target hardware description to translate to. Since each hardware description defines its parent in the hierarchy of hardware descriptions (Fig. 4), the compiler can find the path from *target* to the root *perfect*. Any function in the module with a hardware description that is in this path is iteratively translated to *target*.

Given hardware descriptions *hwdFrom* and *hwdTo* where *hwdTo* is a child of *hwdFrom* in the hierarchy, a function with hardware description *hwdFrom* is translated in three phases: First, the compiler finds equivalence between parallelism units and memory spaces. With this information it then translates all memory spaces of declarations and barrier statements to equivalent memory spaces of hardware description *hwdTo*. Finally, it translates the foreach statements, splitting them up if required. The following paragraphs give a high-level overview of the translation pass. The algorithms are provided in full in Appendix A.

Restrictions for hardware descriptions To make the translation possible, there are several restrictions that apply to the parallelism hierarchy of a child hardware description. The basic principle is that it is only allowed to add memory_spaces and par_groups or par_units.

Since hardware description *hwdTo* may have more layers of units of parallelism than *hwdFrom*, par_groups may have to be split up. The par_groups are only allowed to split up using a more specific number of parallelism units. For example, the par_group threads in hardware description *perfect* with unlimited units of parallelism can be split up in 65535 blocks and 1024 threads in hardware description *gpu*. In *nvidia* the 1024 threads can be split up in 32 warps of 32 threads.

The translation of memory spaces is also governed by rules that are necessary to satisfy the synchronization conditions for barrier statements. To ensure that the translation pass can always find equivalent memory spaces, the rules for a par_group *pgFrom* that splits up in two or more par_groups *pgsTo* are as follows: If *pgFrom* contains memory spaces, then the outermost par_group in *pgsTo* must contain equivalent memory spaces. If the par_unit in par_group *pgFrom* contains memory spaces, then these memory spaces have to be represented in the innermost par_unit in *pgsTo*. These rules guarantee that the translation phase never has to insert new barrier-statements and that existing barrier-statements are translated into barrier-statements that use equivalent memory spaces.

Finding equivalence in units of parallelism Since the compiler may need to split foreach statements, the output of this phase is a list of mappings from a par_group to lists of par_groups. First, for hardware descriptions *hwdFrom* and *hwdTo*, the compiler finds the *executing* par_unit, which is the innermost par_unit defined in the parallelism hierarchy. It retrieves the par_groups of both and compares the number of units of parallelism to decide whether to split up layers in the parallelism hierarchy or not.

For example, in hardware descriptions *perfect* (Fig. 3) and *gpu* (Fig. 8), the executing par_units are thread and thread. The surrounding par_groups are threads and threads. However, the number of units of parallelism of threads in perfect is larger (unlimited vs. 1024), which means that the compiler also needs to incorporate the surrounding par_group of threads in *gpu*, which is blocks. The result is that par_group threads in *perfect* is mapped to both blocks and threads in *gpu*. More details are provided in App. A.2.

Translating memory spaces Each declaration and barrier statement in the function that has a memory space *msFrom* is translated into a declaration or barrier statement with an equivalent memory space *msTo*. To find an equivalent memory space, the compiler finds the par_group or par_unit *pgFrom* where *msFrom* is defined. Using the output of the previous phase, the compiler finds the equivalent par_groups *pgsTo* which will be the starting point for searching equivalent memory spaces. A memory space can be defined within a par_group or within a par_unit. If *msFrom* was defined in a par_unit, the compiler will look in the innermost par_unit to find an equivalent memory space. If the memory space was found in a par_group, it will look in the outermost par_group. The pseudo-code is provided in App. A.3.

Translating foreach **statements** We illustrate this pass using the translation from *accelerator* to *gpu*, where the code for *accelerator* is identical to the code in Fig. 2, except for the keyword perfect on line 1. The resulting code is shown in Fig. 9.

As explained above, a par_group can be mapped to multiple par_groups. The foreach statements can also define multiple dimensions as occurs on lines 5 and 6 in Fig. 2. The compiler generates three kinds of statements: dimension statements that indicate the new dimensions for the foreach statements, the new foreach statements themselves, and indexing statements that define how the old indices are computed from the new indices in the foreach statements.

First, the compiler determines the dimension of the inner foreach taking a standard value from the hardware description for par_group threads (line 1 and 7 in Fig. 9, for clarity we assume that m is larger than nrThreadsM). Then it generates the foreach remembering the old index variable in combination with the new dimension expression. This leaves the compiler with n threads in one dimension and m / nrThreadsM in the other. Since the compiler already used the maximum number of threads, it moves on to blocks and generates two foreach statements with n blocks for one dimension and m / nrThreadsM in the other. As all dimensions are now clear (nrThreadsM, nrBlocksM, and n), the compiler generates the indexing statements on lines 8 and 9. The algorithms are provided in App. A.4.

6.2. Operation Statistics

MCL can give users feedback on the number of data accesses, computational operations, and overhead in the form of indexing and control-flow operations, and the arithmetic intensity, the ratio between the number of data-access and computational instructions.

The compiler accomplishes this by determining the *output-defining blocks*, which are control-flow graph blocks on which the output that a function generates depend. In MCL, the compiler statically

```
 1  foreach (int i in h threads) {
 2    foreach(int j in w threads) {
 3      float sum = 0.0;
 4      for (int y = 0; y < fh; y++) {
 5        for (int x = 0; x < fw; x++) {
 6          sum += f[y, x] * a[i + y, j + x];
 7        }
 8      }
 9      b[i, j] = sum / (fh * fw);
10    }
11  }
```

Figure 16. A convolution program for hardware description *perfect*.

knows which variables are written, since expressions can only be written in an assignment or in a function call and the call-graph is always clear to the compiler.

To determine the *output-defining blocks* the compiler finds the declarations in the parameter list that are written and the declarations of the variables that are used in return expressions. It then uses standard data-flow techniques to find all blocks on which the output depends. We define the *control-flow blocks* as the set of blocks that perform control-flow and all of their dependent blocks that are not in the output-defining blocks. Finally, we define the *indexing blocks* as the blocks that define the indexing expressions, but are neither in the *output-defining blocks* or the *control-flow blocks*.

For all these blocks, the compiler finds the number of iterations they are executed. This analysis may be imprecise because it may not be clear statically how many times a loop is executed, and because blocks may be inside if statements that limit the number of executions. In these cases, the compiler tries to determine the number of iterations and issues a warning that the result is an approximation. However, often the information is still useful as an upper-bound. This is a good example of where our compiler analysis is allowed to be imprecise. If the compiler had to base a transformation on this information, the compiler would have to be conservative. However, only providing feedback opens new possibilities for the compiler to be useful.

For all of these three sets, the compiler gathers what kinds of operations are performed: arithmetic operations, data-accesses with the memory-space, and which par_unit performs the operation. The call-graph is analyzed from the leaves up, such that it is clear how many operations call-expressions perform. Finally, the compiler summarizes all this data taking into account the approximation warnings if they occurred, and presents it to the user.

6.3. *Data Reuse Analysis*

To analyze possible data-reuse we limit ourselves to the variables in the *output-defining blocks* that perform indexing and are read. The compiler analyzes these variables in two phases. The first phase performs a simple data-reuse analysis, but if that analysis fails, it performs a more advanced analysis.

The simple form finds all dependencies of variable v including the dependencies of the indexing variables. If v does not depend on a loop, this is reported as data-reuse. For example, in Fig. 2, the data-reuse analysis reports that variable a[i, k] on line 8 does not depend on the loop variable j (line 7) and therefore this data is reused m times. Similarly, it reports that variable b[k, j] does not depend on loop variable i (line 6) and that the data-reuse is n times.

If an indexing expression depends on all loops, the compiler starts a more advanced data-reuse analysis that tries to report the data-reuse ratio. A data-reuse ratio less than or equal to one means no data-reuse, whereas a value higher than one means there may be data-reuse. Fig. 16 shows an example of such a case. Variable a on line 6 has an indexing expression that depends on all four loops.

The advanced data-reuse analysis can analyze per dimension or for all dimensions. It flattens the indexing expression (for all or one dimension) and it retrieves all loops surrounding a variable v taking into account where v was last defined. It then symbolically computes the data reuse ratio by dividing the number of iterations v is accessed divided by the indexing range of v.

For example, the data-reuse ratio for the first dimension of variable a on line 6 of Fig. 16 is (h * fh)/(h + fh). The compiler computes that the number of accesses to a is h * fh. However, since the indexing range of a is h + fh (i and y filled in, taking into account the offsets and steps of the loops), a maximum of h + fh different elements is accessed, leading to the above data-reuse ratio.

The compiler can report this as an expression (h * fh)/(h + fh), but users can also fill in representative values in the parameter list for h and fh, for example 2048 and 9 respectively. The compiler will then report back a sharing ratio of 8.96. This value is correct considering there is no sharing in the borders of a convolution.

Analyzing per dimension gives programmers an extensive insight in how data is reused. In the above situation, it makes clear that for a specific combination of i and x data in a will be loaded more than once. There are two features of MCL that make this possible: First, MCPL has true multi-dimensional arrays that make it possible to express the program in this manner. The second feature is that the compiler uses all the information expressed by the programmer to do analysis. For example, usually compilers work on an intermediate representation, for instance a representation that flattens the array accesses into one-dimension. However, this would make the data-reuse analysis less precise. In contrast, our compiler performs analysis using all dimensionality information available, which makes it possible to provide more precise feedback and to relate the feedback to the code that the programmer wrote.

Proposing faster memories An extension of the data reuse analysis tries to propose faster memories. Because the mapping from par_units and memory spaces to the physical hardware is well-defined in MCL, the compiler can give very specific feedback. For a variable that has data-reuse, the compiler retrieves the memory space it is declared in. As above there are two cases: the variable is independent from specific foreach statements, or the variable depends on all of them but there is still data-reuse.

In the first case, the compiler retrieves all memory-spaces that are available in the scope of this foreach. The foreach statement references the par_group in the parallelism hierarchy which also defines the relations between the memory spaces and the units of parallelism. It compares these memory spaces with the memory space of the variable to determine whether there is a memory space that is faster.

In the second case, the compiler performs the same analysis, but on each pair of loops for each dimension. The outer foreach loop is used to determine the set of available memory spaces for choosing a faster memory space from.

For determining which memory space is faster, the compiler inspects the memories that hold the memory spaces. For example, in Fig. 8 on lines 15 and 16, memory on_chip holds memory space local. To determine whether a memory a is faster than memory b there are several heuristics: First, the compiler can inspect the interconnects between the memories and the execution units and compare the bandwidth and latencies. Second, if these details are not available, it can determine the proximity to execution units: the closer, the faster. Finally, it can compare the capacities of the memories, the smaller the memory, the faster we assume the memory to be.

6.4. Cache Analysis

The cache analysis tries to provide information about the cache usage of variables inside loops in terms of best-case and worst-case scenarios. Given a cache defined in the hardware description, the analysis retrieves the memory spaces that the cache holds. The analysis proceeds for every declaration that has one of these memory spaces.

For each cache the compiler determines the par_units that have access to the cache. A cache can reside on an execution unit that has a restricted number of slots for the par_units. This is illustrated in Fig. 13 where execution unit smp has slots for 8 blocks and 1024 threads. This would represent an upper-bound for the cache misses for a variable if a cache were defined to reside on this execution unit.

For caches that do not reside on an execution unit but are shared among execution units, the compiler visits the interconnects in search of execution units and the slots property to find out which

$\mathrm{par\_units}$ have access to the cache. The upper-bounds are determined by interpreting the layout of the execution unit in relation to the cache. If a cache can be reached by multiple execution units, the upper-bound for cache misses for a variable is defined to be the number of parallelism units that can reach it. Additionally, the compiler determines whether a $\mathrm{par\_group}$ is executed in lock-step for the load and store instruction, which is the case if a load-store unit resides in a $\mathrm{simd\_group}$.

For a variable, the compiler retrieves the surrounding loops taking into account where the variable was last defined. If the array expressions of the variable are unpredictable, this is reported to the user. Array expressions are unpredictable if an array expression contains non-loop variables that are written or originating from arrays.

Otherwise, the compiler analyzes each loop separately. For a loop it symbolically fills in the start value of the loop into the array expression and the start + the step of the loop into the array expression. It symbolically subtracts these values to determine the stride that a loop causes in the array expression.

Based on this stride and using the knowledge about the cache-line size from the hardware description, the compiler gives the programmer feedback about the best and worst case scenario in terms of cache-line loads. If the stride is zero, then the best case is 1 cache-line fetch, otherwise the number of accesses per cache-line is $cachelinesize/stride$. The worst case is the number of iterations in a loop, or the upper-bound that was determined above if it is a $\mathrm{foreach}$ loop that has a limited amount of slots for the $\mathrm{par\_unit}$ under analysis.

By comparing the best-case and worst-case, this analysis provides the user feedback about whether this variable does, may, or does not benefit from the cache for this loop. Units of parallelism that execute in lock-step and load fewer cache-lines than the number of units of parallelism, are reported to benefit from the cache since multiple threads access the same cache-line.

6.5. Performance Feedback Functions

The complex many-core hardware often has complex rules for execution. As shown in Fig. 13 and 14, HDL provides an API in Java to encode complex rules, for example coalesced memory access or how many blocks can run on an SMP.

A performance feedback function is automatically called by the compiler when encountered in a hardware description. It takes as input a *Kernel* data-structure and returns a string with a message for the user that will be issued by the compiler. The *Kernel* data-structure contains a *Context*, a set of *Instructions*, and the variables from the hardware description. From an *Instruction* we can request whether it is a memory instruction or not, in which memory-space it loads or stores, which $\mathrm{par\_unit}$ is executing the instruction, and we can obtain a String representation. The *Context* data-structure contains a special execution context that can obtain values from the function under investigation. For example, the *Context* has a method $\mathrm{int\ getSizeMemorySpace(String)}$. Hardware description variables can be referenced using an expression such as $\mathrm{Kernel.on\_chip.capacity}$ for the capacity of on-chip memory.

The above API is dynamically generated by the compiler since certain functions rely on both the structure of the hardware description and the function under investigation. An example is the call $\mathrm{kernel.context.hierarchy.threads.getSize()}$ that relies on the structure of the parallelism hierarchy and the value of the number of threads that is specified in the program. An interpreter interprets exactly those statements in the MCPL program that are needed to compute the size. If a value of variable v is not available, the compiler issues a warning that it needs a value for v to complete the analysis.

The interpreter interprets for-loops, but not $\mathrm{foreach}$-statements. To iterate over the units of parallelism in a $\mathrm{foreach}$-statement, the API provides a $\mathrm{reset()}$ and $\mathrm{increment()}$ statement. This limits the execution for the interpreter making the analysis more scalable. For example in analyzing coalescing, we are only interested in 32 iterations for threads in a warp, which can be expressed by issuing $\mathrm{kernel.context.hierarchy.blocks.block.threads.increment()}$ 32 times in a for-loop.

This API is powerful enough to analyze the number of blocks per SMP as illustrated in Fig. 14, coalesced memory access for threads and the number of bank conflicts for threads.

7. EVALUATION

To evaluate our approach we implemented several computational kernels and recorded the received feedback and the performance increase for two very different architectures: an NVIDIA GTX480 GPU and the Intel Xeon Phi. The focus of our framework is to give feedback on individual kernels. Therefore, we selected a variety of programs (compute-bound, bandwidth-bound, regular, and irregular) with one kernel. In Table III we give an overview of the results we obtained. To reduce the size of the table, we collapsed the different versions in the inner process (v1, v2, v3) into one step (v1) in some of the applications.

Our compiler generates OpenCL code for the kernels. For each target hardware description, a configuration file describes how the programming abstractions map to OpenCL code. This ensures that the OpenCL compiler generates executables based on the same set of parameters for each compared kernel, which limits a potential performance difference caused by the OpenCL compiler.

We chose a GPU with the Fermi architecture because the cache is important on this architecture which makes the analysis more challenging for our compiler. Additionally, much related work makes use of GTX480 cards which allows us to compare our results. With the choice for the Xeon Phi, we show that we can support a very different architecture than a GPU. Since the Xeon Phi is a new many-core card, we could not find a comparison for all kernels. In Table IV we compare our results with other existing implementations. We discuss the details below.

matrix multiplication Matrix multiplication is a compute-bound, regular application. The process for this application has been thoroughly explained in Sec. 5 and resulted in a speedup greater than 6 on a GPU (564 GFLOPS) and a speedup of over 10 on the Xeon Phi (488 GFLOPS) compared to the naive implementation when used with 2048×2048 matrices. The CUBLAS 5.5 library obtains 892 GFLOPS and the Intel MKL library 695 GFLOPS. We consider our result as very good performance as these libraries are heavily tuned, most likely written in assembly (a much lower level of abstraction) by experts who have much more knowledge of the architecture than the programming guides provide us.

convolution Convolution is a stencil operation computing an element in the output matrix based on its neighbors. It is somewhat less regular than matrix multiplication and more bandwidth-bound. We used a 4096×4096 matrix with a filter size of 9×9. For hardware description *gpu*, the compiler identifies data-reuse in the filter and in both dimensions in the input data. This prompted us to load both data-structures in local memory. On level *nvidia*, we received the feedback that we should use more blocks per SMP. However, we as programmers have application knowledge and understand that this is a delicate balance between what can be shared and how many blocks can be run in parallel. We succeeded in running 2 blocks per SMP. On level *fermi* we improved cache behavior by changing the number of threads.

For the Xeon Phi, the compiler advised us to compute multiple elements per thread on level *mic*. Based on the data-reuse analyzed by the compiler, we also decided to load the filter in a temporary variable since it is heavily shared. At level *xeon_phi* the compiler identifies that the cache behavior of the output data has the most severe effect. Allocating private memory for the output resulted in 171 GFLOPS. We also received the feedback that we have unaligned vector loads in accessing neighboring elements. This limits the performance on the Xeon Phi.

For the GPU we obtained a speedup of more than a factor 2.3 and on a Xeon Phi a factor 2.5 versus our *perfect* implementation. The implementation with the highest performance on a GTX480 obtains 580 GFLOPS but uses loop unrolling and a specialized kernel for each filter size [47]. Our compiler generates a kernel for any filter size, so a more fair comparison is with the version that does not do loop unrolling which obtains 380 GFLOPS. The Xeon Phi performance is limited by unaligned vector loads in accessing neighboring elements.

histogram Histogram is a bandwidth-bound application with irregular data-access in the output histogram. In effect, it is a reduction, which means that there should be synchronization overhead.

Table III. Feedback and performance from compiler on different abstraction-levels for several applications.

Application	Version	Program Change	Performance
NVIDIA GTX480 GPU			
matmul	perfect		89.0 GFLOPS
	gpu (v1)	use faster memory	100 GFLOPS
	gpu (v2)	multiple elements	91.6 GFLOPS
	gpu (v3)	data reuse	205 GFLOPS
	nvidia (v1)	store-load phases	489 GFLOPS
	fermi (v1)	cache usage	564 GFLOPS
convolution	perfect		129 GFLOPS
	gpu (v1)	use faster memory multiple elements	215 GFLOPS
	nvidia (v1)	scarce memory	241 GFLOPS
	fermi (v1)	cache usage	302 GFLOPS
histogram	perfect		6.94 GB/s
	gpu (v1)	multiple elements data reuse	80.0 GB/s
	nvidia (v1)	scarce memory	80.0 GB/s
gesummv	perfect		2.58 GFLOPS
	gpu (v1)	use faster memory	2.63 GFLOPS
	nvidia (v1)	scarce memory	3.69 GFLOPS
	fermi (v1)	cache analysis	80.2 GFLOPS
blackScholes	perfect		297 GFLOPS
	gpu (v1)	multiple elements	432 GFLOPS
sparse matmul	perfect		1.01 GFLOPS
	gpu (v1)	data reuse	1.19 GFLOPS
	gpu (v2)	multiple elements	1.41 GFLOPS
	fermi (v1)	cache usage	3.88 GFLOPS
Intel Xeon Phi (5110P)			
matmul	perfect		39.9 GFLOPS
	mic (v1)	multiple elements	35.7 GFLOPS
	mic (v2)	data reuse	47.6 GFLOPS
	xeon_phi (v1)	cache usage	87.8 GFLOPS
	xeon_phi (v2)	cache usage	115 GFLOPS
	xeon_phi (v3)	cache usage	488 GFLOPS
convolution	perfect		66.4 GFLOPS
	mic (v1)	multiple elements	84.0 GFLOPS
	mic (v2)	data reuse	153 GFLOPS
	xeon_phi (v1)	cache usage	171 GFLOPS
histogram	perfect		0.35 GB/s
	mic (v1)	multiple elements data reuse	23.6 GB/s
gesummv	perfect		0.84 GFLOPS
	mic (v1)	multiple elements	0.68 GFLOPS
	xeon_phi (v1)	cache usage	55.4 GFLOPS
blackScholes	perfect		66.6 GFLOPS
	mic (v1)	multiple elements	54.1 GFLOPS
	xeon_phi (v1)	cache usage	115 GFLOPS
sparse matmul	perfect		2.71 GFLOPS
	mic (v1)	data reuse	2.84 GFLOPS
	xeon_phi (v1)	cache usage	2.90 GFLOPS

For this application, we computed a histogram of 256 buckets from an input of 16384×16384 elements. Since there is only one integer addition per word that we read, we measure the performance in GB/s (where a kB is 1024 bytes).

The compiler indicated that there is much data-reuse in the output array, which we can interpret as synchronization overhead. On level *gpu*, we first computed multiple elements and minimized the synchronization overhead by replicating the histogram in local memory. This led to a performance of 80.0 GB/s. We would expect a performance close to 145 GB/s, but because atomic operations are expensive on shared memory, we can not obtain better results. In comparison with the CUDA

Table IV. Performance comparison for the applications compared to known, fully optimized versions (* measured on a C2050, ** using a different input than in Table III).

Application	Naive	MCL	Other implementations
NVIDIA GTX480 GPU			
matmul	89.0 GFLOPS	564 GFLOPS	892 GFLOPS
convolution	129 GFLOPS	302 GFLOPS	380 GFLOPS
histogram	6.94 GB/s	80.0 GB/s	80 GB/s
gesummv*	1.25 GFLOPS	51.9 GFLOPS	2.98 GFLOPS
blackScholes	297 GFLOPS	432 GFLOPS	237 GFLOPS
sparse matmul	1.01 GFLOPS	3.88 GFLOPS	8.9 GFLOPS
Intel Xeon Phi (5110P)			
matmul	39.9 GFLOPS	488 GFLOPS	695 GFLOPS
convolution	66.4 GFLOPS	171 GFLOPS	n/a
histogram**	0.331 GB/s	11.9 GB/s	0.3 GB/s
gesummv	0.84 GFLOPS	55.4 GFLOPS	n/a
blackScholes	66 GFLOPS	115 GFLOPS	n/a
sparse matmul**	3.93 GFLOPS	5.28 GFLOPS	13 GFLOPS

histogram implementation, we have the same performance. On level *nvidia* we received the feedback that shared memory is scarce, but the improvement did not result in higher performance.

For the Xeon Phi, on level *mic* the compiler advised to compute multiple elements and that there is much data-reuse in the histogram. Computing more elements per thread and replicating histograms led to a dramatic increase in performance. The cache analysis on level *xeon_phi* did not give feedback we could act on. Our version outperforms other work [48] by a factor 40 (5.25 ms against 200 ms on an input of 2^{24}).

gesummv This application from the PolyBench benchmark [49] performs two matrix-vector multiplications. Implementing several improvements on level *gpu* and *nvidia* did not contribute to much performance gain. However, at level *fermi* it became clear that the application has bad cache behavior. After solving these issues, the performance reached 80.2 GFLOPS. The bandwidth reached 140 GB/s and the compiler reported that there is not much data-reuse in the matrices. Since the maximum bandwidth on a GTX480 is 145 GB/s and there is no data-reuse that we can leverage, we know that we are close to the maximum performance. We compared our results with the results in the paper by Grauer-Gray et al. [49]. Their auto-tuned version has a run-time of 25.4 ms and their manually written version is 1.5 times faster than their auto-tuned version on an input of 4096×4096 elements on a Fermi C2050 GPU. However, our version is an order of magnitude faster with a run-time of 1.46 ms on the same hardware. A possible explanation of this difference is that their version may have missed the cache optimization that resulted in a significant performance increase in our version (Table III).

The Xeon Phi initially obtained much lower performance than the GPU. Unfortunately, the feedback that we received initially made matters worse due to the overhead of computing multiple elements. However, on level *xeon_phi* we received feedback about similar cache issues as on the GPU and we solved it in a similar way resulting in a dramatic speedup of over 65.

Black-Scholes This application analyzes stock options, in our case $4 * 10^6$ transactions. The compiler reports that this is a very compute intensive application and the initial performance of 297 GFLOPS at level *perfect* on the GPU confirms this. Following the feedback on level *gpu* led to a performance of 432 GFLOPS. The feedback on levels *nvidia* and *fermi* did not provide any opportunity to improve the program. The performance we obtain is good as the CUDA SDK version obtains 237 GFLOPS.

For the Xeon Phi we are advised to compute multiple elements on level *mic*. Unfortunately, this led to worse performance. On level *xeon_phi* it becomes clear that the application suffered from bad cache behavior for the vector instructions. Correcting this led to a version that obtains 115 GFLOPS.

sparse matrix multiplication Sparse matrix multiplication is a very irregular application. We used a 65536×131072 sparse matrix, randomly filled with a .001 density in CSR representation.

In the GPU version, the feedback on levels *gpu* and *nvidia* did not help much. On level *fermi*, the cache feedback initially indicated that the source vector access is irregular and that the cache effect of the matrix access cannot be determined due to unknown loop bounds. This prompted us to use another approach, to at least improve the cache behavior in the index and matrix access, by using multiple threads to compute output elements. In the end, however, the performance of this application is dominated by the irregular access to the source vector. The end result is about 2.3 times slower than the CuSparse 5.5 implementation, which achieves about 8.9 GFLOPS on the same input. We suspect that the library is using texture memory, which MCL does not support yet.

The Xeon Phi version initially has good performance and can be slightly improved by taking the cache feedback of level *xeon_phi* into account. We compared our results with the results from [50] using the Mip1 matrix and observe that we are a factor 2.5 slower. This is comparable to the difference with CuSparse on a GPU.

The evaluation shows that following the *stepwise-refinement for performance* methodology has several benefits: Firstly, we obtain significant speedup in almost all cases. Only in sparse matrix multiplication we hardly obtain speedup using a specific input. However, for a different input (the Mip1 matrix from [50]), our implementation does show significant speed-up (Table IV).

Secondly, we are guided by feedback from the compiler. This steers us in the right direction and helps us understand the performance bottlenecks. For example, with convolution for the Xeon Phi, we receive the feedback that we have unaligned vector accesses. In the gesummv application we understand that we are close to the maximum performance, since the compiler tells us that there is no data-reuse and we observe a performance close to the maximum bandwidth.

Finally, our methodology also shows the trade-offs between abstraction-levels. A high-level of abstraction increases portability (programs written for level *perfect* can run on both the GPU and the Xeon Phi), but incorporating hardware-specific details in the program can increase the performance significantly and also helps programmers to understand the performance in relation to the hardware.

8. DISCUSSION

In this paper we show a methodology with which programmers work in cooperation with the compiler to gain both performance and understanding of this performance. Programmers with application knowledge can judge whether the compiler gives reasonable feedback. For example, in the convolution application, the compiler suggests to minimize the use of shared memory to allow for more blocks per SMP, which is usually good feedback. However, we as programmers understand that in this application, the speed-up we gain relies on the amount of data we share.

Our methodology can be combined with auto-tuning. Often, to reach even higher performance, auto-tuning of parameters such as the number of threads per block is necessary. Since programmers gain an understanding of the limits of the program and hardware after receiving feedback from the compiler, they have the opportunity to choose a good search space for auto-tuners, possibly making auto-tuning much more efficient.

We chose to design a new programming language besides the hardware descriptions because this makes it easy to define an explicit mapping to hardware and to restrict the language to enable better analysis. However, if the right restrictions on other languages such as CUDA or OpenCL are enforced, augmented with annotations, there is no reason why our techniques could not be applied to other languages. The annotations would have to provide data-structure size information, dimensionality of the data-structures, memory space information, and units of parallelism.

The different versions that the programmer creates could be automatically documented together with the received feedback for each version. For example a standard versioning system could maintain the different kernel versions together with meta-data files that store the feedback. This system then provides a way to not only automatically document the optimizations that have been applied, but also the trigger for applying a specific optimization in the form of the received feedback. In effect, it can document the reasoning of the programmer and it captures the optimization knowledge.

Another benefit of having multiple versions of compute kernels is that the system could verify the optimized versions in an automated way. The system could run example inputs on kernels on different abstraction-levels reporting inconsistent results between high-level and low-level versions.

The techniques that our compiler uses are mainly targeted at providing high-quality feedback. For future work, we would like to investigate more advanced techniques to make the feedback more powerful. For example, we would like to investigate whether polyhedral analysis [51, 19] can be combined with our language and compiler framework such that we can still provide high-quality feedback that helps the programmer to improve the program.

Another way to improve the analyses is to extend our performance feedback functions to support advanced performance models. For example, incorporating the Roofline model [52] might provide more accurate information on when to stop optimizing. Additionally, we would like to investigate whether we can incorporate the Xeon Phi's cache model by Ramos et al. [53] in a performance feedback function.

A further direction is to extend our system to describe clusters of many-core systems. Hardware descriptions could describe topologies between clusters. MCL could be modified to express programs for many-core clusters.

Finally, we would like to improve the hardware descriptions. For example, in the current hardware descriptions it is not determined how instructions are scheduled, although this may impact the performance. For instance, this may benefit the Xeon Phi programs because the Xeon Phi has multiple pipelines with complex rules for how instructions from multiple threads are scheduled onto those pipelines.

9. CONCLUSION

In this paper we presented the *stepwise-refinement for performance* methodology, a novel approach to provide programmers control and understanding on levels of abstraction they can choose themselves. Our methodology centers around hardware descriptions that specify many-core hardware with different levels of detail, resulting in multiple levels of abstraction. This provides programmers a trade-off between high-level programming which is good for portability and code maintainability, and low-level programming with a well-defined interface to the hardware.

The *stepwise-refinement for performance* methodology supports a structured approach in which programmers are gradually exposed to more hardware details in an iterative process. During this process, the compiler gives programmers detailed performance feedback about the programs to increase their understanding of the performance they obtain. This approach leverages the strengths of both the programmer and compiler: the programmer has application knowledge that the compiler lacks, and the compiler does not have to be as conservative in providing feedback as it has to be for automatic transformations. Using our methodology, the programmer and compiler work together to produce high-performance code.

To show the process of optimizing many-core programs with this methodology, we presented Many-Core Levels, our programming system that supports our methodology. MCL contains a hardware description language and a programming language that are tightly integrated and help the compiler to produce detailed, hardware-specific performance feedback with a strong relation to the code that programmers wrote.

We gave a thorough example of the optimization process to explain how programmer and compiler work together to produce high-performance code. We explained how MCL automatically translates between abstraction-levels, how we analyze data-reuse, cache behavior and how we encode complex architecture-specific rules to give feedback. We evaluated our approach with several well-known, widely varying (compute-bound, bandwidth-bound, regular, and irregular) many-core programs on two different architectures: a GPU and a Xeon Phi. We demonstrate that our methodology provides programmers a well-defined hardware interface that gives them a trade-off in portability against performance. We show that it is possible to obtain substantial speed-ups in almost all cases with the important benefit that programmers not only gain performance but also understand the performance they obtain.

REFERENCES

1. URL https://github.com/pieterhijma/mcl, last accessed: August 2014.
2. Buck I, Foley T, Horn D, Sugerman J, Fatahalian K, Houston M, Hanrahan P. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Trans. Graph.* Aug 2004; **23**(3):777–786.
3. Dubach C, Cheng P, Rabbah R, Bacon DF, Fink SJ. Compiling a High-Level Language for GPUs. *Proc. of the 33rd ACM SIGPLAN conf. on Programming Language Design and Implementation*, PLDI '12, ACM: New York, NY, USA, 2012; 1–12.
4. Valiant LG. A Bridging Model for Parallel Computation. *Commun. ACM* Aug 1990; **33**(8):103–111.
5. Hou Q, Zhou K, Guo B. BSGP: Bulk-Synchronous GPU Programming. *ACM Trans. Graph.* August 2008; **27**:19:1–19:12.
6. Jeffers J, Reinders J. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013. ISBN: 9780124104143.
7. Blelloch GE. Programming Parallel Algorithms. *Commun. ACM* 1996; **39**(3):85–97.
8. Bergstrom L, Reppy J. Nested Data-Parallelism on the GPU. *Proc. of the 17th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '12, ACM: New York, NY, USA, 2012; 247–258.
9. Catanzaro B, Garland M, Keutzer K. Copperhead: Compiling an Embedded Data Parallel Language. *Proc. of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, 2011; 47–56.
10. Newburn CJ, So B, Liu Z, McCool M, Ghuloum A, Toit SD, Wang ZG, Du ZH, Chen Y, Wu G, *et al.*. Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. *Proc. of the 9th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*, CGO '11, 2011; 224–235.
11. Cunningham D, Bordawekar R, Saraswat V. GPU Programming in a High Level Language: Compiling X10 to CUDA. *Proc. of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, ACM: New York, NY, USA, 2011; 8:1–8:10.
12. Guo J, Thiyagalingam J, Scholz SB. Breaking the GPU Programming Barrier with the Auto-Parallelising SAC Compiler. *Proc. of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, ACM: New York, NY, USA, 2011; 15–24.
13. Larsen B. Simple Optimizations for an Applicative Array Language for Graphics Processors. *Proc. of the sixth workshop on Declarative Aspects of Multicore Programming*, DAMP '11, ACM: New York, NY, USA, 2011; 25–34.
14. Lee S, Grover V, Keller G, Chakravarty MM. GPU Kernels as Data-Parallel Array Computations in Haskell. *Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods (EPHAM 2009)*, 2009.
15. Chakravarty MM, Keller G, Lee S, McDonell TL, Grover V. Accelerating Haskell Array Codes with Multicore GPUs. *Proc. of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, 2011; 3–14.
16. Claessen K, Sheeran M, Svensson BJ. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. *Proc. of the 7th workshop on Declarative Aspects and Applications of Multicore Programming*, DAMP '12, ACM: New York, NY, USA, 2012; 21–30.
17. Mainland G, Morrisett G. Nikola: Embedding Compiled GPU Functions in Haskell. *SIGPLAN Not.* Sep 2010; **45**(11):67–78.
18. Yang Y, Xiang P, Kong J, Zhou H. A GPGPU Compiler for Memory Optimization and Parallelism Management. *Proc. of the 2010 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '10, 2010; 86–97.
19. Wang C, Kang K, Zhu M, Deng Y. A Polyhedral Modeling Based Source-to-Source Code Optimization Framework for GPGPU. *2012 IEEE 26th Int.l Parallel and Distributed Processing Symp. Workshops & PhD Forum*, 2012; 1964–1970.
20. Buono D, Danelutto M, Lametti S, Torquati M. Parallel Patterns for General Purpose Many-Core. *21st Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, 2013; 131–139.
21. Majeed M, Dastgeer U, Kessler C. Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters. *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA-2013), Las Vegas, USA, July 2013*, 2013.
22. Nugteren C, Corporaal H. Introducing 'Bones': A Parallelizing Source-to-source Compiler Based on Algorithmic Skeletons. *Proc. of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, ACM: New York, NY, USA, 2012; 1–10.
23. Matsuzaki K, Kakehi K, Iwasaki H, Hu Z, Akashi Y. A fusion-embedded skeleton library. *Euro-Par 2004 Parallel Processing*, *Lecture Notes in Computer Science*, vol. 3149, Danelutto M, Vanneschi M, Laforenza D (eds.). Springer Berlin Heidelberg, 2004; 644–653.
24. Aldinucci M, Gorlatch S, Lengauer C, Pelagatti S. Towards Parallel Programming by Transformation: The FAN Skeleton Framework. *Parallel Algorithms and Applications* 2001; **16**(2):87–121.
25. Sato S, Iwasaki H. A Skeletal Parallel Framework with Fusion Optimizer for GPGPU Programming. *Programming Languages and Systems*, *Lecture Notes in Computer Science*, vol. 5904, Hu Z (ed.). Springer Berlin Heidelberg, 2009; 79–94.
26. Chafi H, Sujeeth AK, Brown KJ, Lee H, Atreya AR, Olukotun K. A Domain-Specific Approach to Heterogeneous Parallelism. *Proc. of the 16th ACM symp. on Principles and Practice of Parallel Programming*, PPoPP '11, 2011; 35–46.

27. Cartey L, Lyngsø R, de Moor O. Synthesising Graphics Card Programs from DSLs. *Proc. of the 33rd ACM SIGPLAN conf. on Programming Language Design and Implementation*, PLDI '12, 2012; 121–132.
28. Nickolls J, Buck I, Garland M, Skadron K. Scalable Parallel Programming with CUDA. *Queue* 2008; **6**(2):40–53.
29. Stone J, Gohara D, Shi G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering* 2010; **12**(3):66–73.
30. Han TD, Abdelrahman TS. hiCUDA: High-Level GPGPU Programming. *IEEE Trans. Parallel Distrib. Syst.* Jan 2011; **22**(1):78–90.
31. Lee S, Eigenmann R. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. *Proc. of the 2010 ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '10, IEEE Computer Society: Washington, DC, USA, 2010; 1–11.
32. Lee S, Min SJ, Eigenmann R. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. *Proc. of the 14th ACM SIGPLAN symp. on Principles and Practice of Parallel Programming*, PPoPP '09, ACM: New York, NY, USA, 2009; 101–110.
33. Wienke S, Springer P, Terboven C, an Mey D. OpenACC - First Experiences with Real-World Applications. *Euro-Par 2012 Parallel Processing*, *Lecture Notes in Computer Science*, vol. 7484, Kaklamanis C, Papatheodorou T, Spirakis P (eds.). Springer Berlin / Heidelberg, 2012; 859–870.
34. Lee S, Vetter JS. Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. *Proc. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, IEEE Computer Society Press: Los Alamitos, CA, USA, 2012; 23:1–23:11.
35. Lee S, Vetter JS. Openarc: Open accelerator research compiler for directive-based, efficient heterogeneous computing. *Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing*, HPDC '14, ACM: New York, NY, USA, 2014; 115–120.
36. Lopez-Novoa U, Mendiburu A, Miguel-Alonso J. A Survey of Performance Modeling and Simulation Techniques for Accelerator-based Computing. *IEEE Trans. on Parallel and Distributed Systems* 2014; **PP**(99):1–1, doi: 10.1109/TPDS.2014.2308216.
37. Burtscher M, Kim BD, Diamond J, McCalpin J, Koesterke L, Browne J. PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. *Proc. of the 2010 ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '10, IEEE Computer Society: Washington, DC, USA, 2010; 1–11.
38. Rane A, Sardeshpande S, Browne J. Poster: Determining Code Segments That Can Benefit from Execution on GPUs. *Proc. of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, ACM: New York, NY, USA, 2011; 55–56.
39. Rane A, Browne J, Koesterke L. PerfExpert and MACPO: Which code segments should (not) be ported to MIC. *TACC-Intel Highly Parallel Computing Symposium*, 2012.
40. Fialho L, Browne J. Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems. *Supercomputing*, *Lecture Notes in Computer Science*, vol. 8488, Kunkel J, Ludwig T, Meuer H (eds.). Springer International Publishing, 2014; 261–277.
41. Bell N, Hoberock J. Thrust: A Productivity-Oriented Library for CUDA. *GPU Computing Gems*. Morgan Kaufmann Publishers, 2011; 359–371.
42. Fatahalian K, Knight TJ, Houston M, Erez M, Horn DR, Leem L, Park JY, Ren M, Aiken A, Dally WJ, *et al.*. Sequoia: Programming the Memory Hierarchy. *Proc. of the ACM/IEEE SC 2006 Conference*, 2006.
43. Spafford KL, Vetter JS. Aspen: A Domain Specific Language for Performance Modeling. *Proc. of the Int.l Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012; 84:1–84:11.
44. Ammar RA. Hierarchical Performance Modeling and Analysis of Distributed Software Systems. *Handbook of Parallel Computing: Models, Algorithms and Applications*, Rajasekaran S, Reif J (eds.). 1 edn., chap. 12, Chapman & Hall, 2007. ISBN: 9781584886235.
45. URL https://github.com/pieterhijma/mcl/tree/master/doc, last accessed: August 2014.
46. Klint P, van der Storm T, Vinju J. EASY Meta-programming with Rascal. *Generative and Transformational Techniques in Software Engineering III*, *Lecture Notes in Computer Science*, vol. 6491, Fernandes, João and Lämmel, Ralf and Visser, Joost and Saraiva, João (ed.). Springer Berlin / Heidelberg, 2011; 222–289, doi: 10.1007/978-3-642-18023-1_6. URL http://dx.doi.org/10.1007/978-3-642-18023-1_6.
47. van Werkhoven B, Maassen J, Bal HE, Seinstra FJ. Optimizing convolution operations on gpus using adaptive tiling. *Future Generation Computer Systems* 2014; **30**:14 – 26.
48. Fang J, Varbanescu AL, Sips H. Identifying the key features of intel xeon phi: A comparative approach. *Technical Report*, Delft University of Technology 2013.
49. Grauer-Gray S, Xu L, Searles R, Ayalasomayajula S, Cavazos J. Auto-tuning a high-level language targeted to gpu codes. *Innovative Parallel Computing (InPar), 2012*, 2012; 1–10.
50. Liu X, Smelyanskiy M, Chow E, Dubey P. Efficient sparse matrix-vector multiplication on x86-based many-core processors. *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, 2013; 273–282.
51. Ancourt C, Irigoin F. Scanning Polyhedra with DO Loops. *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, ACM: New York, NY, USA, 1991; 39–50, doi: 10.1145/109625.109631. URL http://doi.acm.org/10.1145/109625.109631.
52. Williams S, Waterman A, Patterson D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* Apr 2009; **52**:65–76, doi:http://doi.acm.org/10.1145/1498765.1498785.
53. Ramos S, Hoefler T. Modeling Communication in Cache-Coherent SMP Systems: A Case-study with Xeon Phi. *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, ACM: New York, NY, USA, 2013; 97–108, doi:10.1145/2462902.2462916. URL http://doi.acm.org/10.1145/2462902.2462916.

A. TRANSLATING AN MCPL PROGRAM TO LOWER LEVEL OF ABSTRACTION

This appendix describes how an MCPL module, represented by a list of functions is translated to hardware description $target$, given a hierarchy of hardware descriptions. We have added type annotations to the pseudo code below to make the code more readable. The type list[T] is a list of type T, a list expression is $[1, 2, 3]$. A tuple[T, S] is a tuple with types T and S, a tuple expression is $< 1, 2 >$. The first and second field of a tuple can be reached with $t.first$ and $t.second$ where t is a tuple.

Throughout the following algorithms the type ParGroupMapping is an important type which is a list of tuples, where each tuple contains two fields with the first field being a ParGroup and the second field a list of ParGroups: list[tuple[ParGroup, list[ParGroup]]]. The type ParGroupMapping will be used as an alias for this type.

A ParGroup contains a ParUnit, and a ParUnit may contain a ParGroup. We consider the outer scope of a $\mathrm{parallelism}$ hierarchy a ParGroup as well, the top ParGroup $parallelism$. All ParGroups and ParUnits can contain MemorySpaces.

A.1. The top-level functions

The top-level function takes as input a list of functions and a string representing the hardware description to which we want to translate. Since a hardware description always knows its parent in the hierarchy, the function construct a path from hardware description $perfect$ to $target$.

```
Require: each hardware description in $hwds$ (except for "perfect") has defined its parent in the
         hierarchy
Ensure: each function that has a path to $target$ is translated to $target$
1: function TRANSLATEFUNCTIONS(list[Function] $fs$, str $target$, list[HWDesc] $hwds$)
2:     list[HWDesc] $path \leftarrow$ FINDPATH("perfect", $target$, $hwds$)
3:     for all Function $f$ in $fs$ do
4:         TRANSLATEFUNCTION($f$, $target$, $path$)
5:     end for
6: end function
```

The following function translates one function iteratively to $target$:

```
Ensure: if function is on the $path$ from "perfect" to $target$, then it is translated to $target$
1: function TRANSLATEFUNCTION(Function $f$, str $target$, list[HWDesc] $path$)
2:     for int $i \leftarrow 0$; $i < path.size - 1$; $i \leftarrow i + 1$ do
3:         if $f.hwdesc = path[i]$ then
4:             TRANSLATE($f$, $path[i + 1]$)
5:         end if
6:     end for
7: end function
```

The following function translates a function, but only one step in the hierarchy. An executing ParUnit is the innermost ParUnit in a parallelism hierarchy, the unit that executes the instructions. In MCPL other ParUnits are not allowed to execute code. An executing ParGroup is a ParGroup that contains the executing ParUnit.

```
Require: $hwdescTo$ has as direct parent $f.hwdesc$
1: function TRANSLATE(Function $f$, HWDesc $hwdescTo$)
2:     HWDesc $hwdescFrom \leftarrow f.hwdesc$
3:     ParUnit $executingParUnitFrom \leftarrow$ GETEXECUTINGPARUNIT($hwdescFrom$)
4:     ParUnit $executingParUnitTo \leftarrow$ GETEXECUTINGPARUNIT($hwdescTo$)
5:     ParGroup $parGroupTo \leftarrow$ GETPARGROUP($executingParUnitTo$)
6:     ParGroup $parGroupFrom \leftarrow$ GETPARGROUP($executingParUnitFrom$)

7:     ParGroupMapping $m \leftarrow [< parGroupFrom, [parGroupTo] >]$
```

```
 8:     $m \leftarrow$ FINDEQUIVALENTPARGROUPS($m$)
          ▷ Sec. A.2

 9:     TRANSLATEDECLARATIONS($f$, $m$)
10:     TRANSLATEBARRIERS($f$, $m$)
          ▷ Sec. A.3

11:     TRANSLATEFOREACHSTATS($f$, $m$)
          ▷ Sec. A.4
12: end function
```

A.2. Finding equivalent ParGroups

The following function finds equivalence between ParGroups. For each ParGroup the compiler needs to know how to translate it to potential more ParGroups. Therefore the function needs a mapping from a ParGroup to lists of ParGroups. As this may occur at different levels in the parallelism hierarchy, the compiler maintains a list of those mappings, for each level an element in the list. A mapping is represented by a tuple with the first element denoting the "from" ParGroup and the second element the list of ParGroups "to". Below we explain the algorithm in more detail.

```
Require: $m$ contains at least a mapping from executing ParGroups
 1: function FINDEQUIVALENTPARGROUPS(ParGroupMapping $m$)
 2:     ParGroup $from \leftarrow m[0].first$
 3:     list[ParGroup] $to \leftarrow m[0].second$
 4:     if $from = parallelism \wedge to[0] = parallelism$ then
 5:         return $m$
 6:     else if $from = parallelism \vee to[0] = parallelism$ then
 7:         ERROR("not equivalent")
 8:     end if

 9:     ParGroup $parentFrom \leftarrow$ GETPARENTPARGROUP($from$)
10:     ParGroup $parentTo \leftarrow$ GETPARENTPARGROUP($to[0]$)
11:     if NRUNITS($from$) $=$ NRUNITS($to$) then
12:         return FINDEQUIVALENTPARGROUPS($[< parentFrom, [parentTo] >] + m$)
13:     else if NRUNITS($from$) $<$ NRUNITS($to$) then
14:         ERROR("from less than to")
15:     else if NRUNITS($from$) $>$ NRUNITS($to$) then
16:         if $parentTo = parallelism$ then
17:             $m \leftarrow [< parentFrom, [parentTo] >] + m$
18:         else
19:             $m[0].second \leftarrow [parentTo] + to$
20:         end if
21:         return FINDEQUIVALENTPARGROUPS($m$)
22:     end if
23: end function
```

Line 4 is the stop condition that states that the function stop if it has reached the top of the parallelism hierarchy. Both $from$ and to must have reached the $parallelism$ to be a valid hardware description. In the conditions in lines 11-22 the flow of control is based on the number of units in $from$ and to. As to is a list, the number of units is the product of the ParGroup sizes contained in the list. Everything is less than $unlimited$ except $unlimited$ itself which is equal to $unlimited$.

If the number of units in $from$ and to are equal, the function prepends the parents to the mapping and continues recursively. In HDL it is not allowed to have less units of parallelism in $from$ than in to (line 13 and 14). If the number of parallelism units in $from$ is greater than in to, the compiler has to add the parent of $to[0]$ to the mapping. However, if the parent is $parallelism$, the function can assume that the parent of $from$ is also $parallelism$. If this is not true, then the stop-condition will detect the error and the hardware description is not valid.

A.3. Translating memory spaces

Both declarations and barrier statements contain memory-space expressions. For simplicity we do not handle declarations that use the default memory-space. Primitive constant declarations do not have memory spaces and are not considered. The functions TRANSLATEDECLARATIONS and TRANSLATEBARRIERS are now shown. They visit the function and call TRANSLATEDECL and TRANSLATEBARRIER for each declaration and barrier in the function respectively.

```
 1: function TRANSLATEDECL(Decl $d$, ParGroupMapping $m$)
 2:     if $d$ is not primitive and constant then
 3:         MemorySpace $from \leftarrow$ GETMEMORYSPACE($d$)
 4:         MemorySpace $to \leftarrow$ FINDEQUIVALENTMEMORYSPACE($from$, $m$)
 5:         $d \leftarrow$ SETMEMORYSPACE($d$, $to$)
 6:     end if
 7: end function
```

```
 1: function TRANSLATEBARRIER(Barrier $b$, ParGroupMapping $m$)
 2:     MemorySpace $from \leftarrow$ GETMEMORYSPACE($b$)
 3:     MemorySpace $to \leftarrow$ FINDEQUIVALENTMEMORYSPACE($from$, $m$)
 4:     $b \leftarrow$ SETMEMORYSPACE($b$, $to$)
 5: end function
```

The function below requires that memory spaces do not disappear in moving from a higher level of abstraction to a lower level of abstraction. Therefore, this function defines the limitation on designing parallelism hierarchies. It is also required that memory spaces that are defined in ParUnits have a representative memory-space in a lowest-level ParUnit, and memory spaces in ParGroups in the highest-level ParGroup. We explain the algorithm below.

```
Require: In the mapping there should be a representative memory space for $ms$.
 1: function FINDEQUIVALENTMEMORYSPACE(MemorySpace $ms$, ParGroupMapping $m$)
 2:     $< level, inUnit > \leftarrow$ FINDMEMORYSPACE($ms$, $m$)
          ▷ $ms$ is in $m[level].first$
          ▷ $inUnit$ denotes whether $ms$ was found in a ParUnit or not
 3:     list[ParGroup] $pgsTo \leftarrow m[level].second$
 4:     if $inUnit$ then
 5:         ParGroup $pg \leftarrow$ LAST($pgsTo$)
 6:         ParUnit $pu \leftarrow$ GETPARUNIT($pg$)
 7:         return FINDEQUIVALENTMEMORYSPACE($ms$, $pu.memorySpaces$)
 8:     else
 9:         ParGroup $pg \leftarrow pgsTo[0]$
10:         return FINDEQUIVALENTMEMORYSPACE($ms$, $pg.memorySpaces$)
11:     end if
12:     ERROR("no matching memory-space")
13: end function
```

On line 2, the $level$ in the mapping for the memory space is found. It is also detected whether the memory-space is in a ParUnit or a ParGroup. The function then looks up the equivalent memory-space in the mapping, taking into account whether ms was found in a ParUnit or ParGroup. In the first case, the function retrieves the memory from the last, the lowest-level ParGroup. If ms was found in the ParGroup, then the function retrieves it from the first ParGroup.

The following function finds the right memory-space with a preference to match whether a memory space is read-only or not. If there is no match, then it returns a memory-space that is not read-only. It is an error if ms is not read-only and there are only read-only memory-spaces in mss.

```
function FINDEQUIVALENTMEMORYSPACE(MemorySpace $ms$, list[MemorySpace] $mss$)
    for all MemorySpace $msTo$ in $mss$ do
        if $(ms.readonly \wedge msTo.readonly) \vee (\neg ms.readonly \wedge \neg msTo.readonly)$ then
            return $msTo$
        end if
    end for
    for all MemorySpace $msTo$ in $mss$ do
        if $ms.readonly$ then
            return $msTo$
        end if
    end for
    ERROR("no equivalent memory space")
end function
```

A.4. Translating ForEach statements

This section discusses how ForEach statements are translated into ForEach statements of a lower-level abstraction. The following function translates a list of statements in a list of statements where each ForEach has been translated to the lower-level abstraction. Other supporting statements will also be generated.

Ensure: Each ForEach in $stats$ has been translated, including the nested ForEach statements.

```
1: function TRANSLATEFOREACHSTATS(list[Stat] $stats$, ParGroupMapping $m$)
2:     list[Stat] $newStats \leftarrow []$
3:     for all Stat $s$ in $stats$ do
4:         if $s$ is ForEach then
5:             $newStats \leftarrow newStats +$ TRANSLATEFOREACHSTAT($s$, $m$)
6:         else
7:             $newStats \leftarrow newStats + [s]$
8:         end if
9:     end for
10:    return $newStats$
11: end function
```

The following function is called for ForEach statements. If the ForEach is not split into multiple ForEach statements, then it is a simple translation.

Require: s is a statement with a ForEach

Ensure: s and all its inner ForEach statements are translated into a list of Stats with ForEach, dimension, and indexing Stats.

```
1: function TRANSLATEFOREACHSTAT(Stat $s$, ParGroupMapping $m$)
2:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
         ▷ $m[currentLevel].first = s.foreach.parGroup$
3:     if $m[currentLevel].second.size = 1$ then
4:         return TRANSLATEFOREACHSIMPLE($s$, $m$)
5:     else
6:         return TRANSLATEFOREACHADVANCED($s$, $m$)
7:     end if
8: end function
```

A simple translation of ForEach statements just modifies the ParGroup of the ForEach statement. It continues by calling TRANSLATEFOREACHSTATS on each inner ForEach statement.

Require: The list of ParGroups $m[currentLevel].second$ has only one element.

Ensure: A list of statements is returned.

```
1: function TRANSLATEFOREACHSIMPLE(Stat $s$, ParGroupMapping $m$)
2:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
3:     $s.foreach.parGroup \leftarrow m[currentLevel].second[0]$
4:     $s.foreach.stats \leftarrow$ TRANSLATEFOREACHSTATS($s.foreach.stats$, $m$)
5:     return [s]
6: end function
```

The above function automatically handles multiple dimensions within the same ParGroup. This is not the case for the function below. First, the function has to find out how many dimensions the ForEach has. It then collect the $innerStats$ for the last dimension with which it will continue.

The function generates three kinds of statements. The $dimensionStats$ compute the dimensions of each ForEach statement, the $foreachStats$ contain the translated ForEach Statements, and the $indexingStats$ contain statements that translate the index of the old ForEach into indices for the new ForEeach statements. The for-loop on line 9 collects all three statements for each dimension, starting with the innermost. Variable $pgsTo$ will be updated to reflect which ParGroups still need to be handled.

On line 16, the inner foreach-statements that do not have $s.foreach.parGroup$ are translated. The statement on line 17 adds the indexing statements to the inner statements. On lines 18-21 the nesting of statements is organized.

Ensure: A list of statements is returned.

```
1: function TRANSLATEFOREACHADVANCED(Stat $s$, ParGroupMapping $m$)
2:     int $nrDimension \leftarrow$ GETNRDIMENSIONS($s.foreach$, $s.foreach.parGroup$)
         ▷ # dimensions in the same ParGroup
3:     list[Stat] $innerStats \leftarrow$ GETSTATSDIMENSION($nrDimension - 1$, $s.foreach$)
4:     list[Stat] $dimensionStats \leftarrow []$
5:     list[Stat] $foreachStats \leftarrow []$
6:     list[Stat] $indexingStats \leftarrow []$
7:     int $currentLevel \leftarrow$ GETLEVEL($s.foreach.parGroup$, $m$)
8:     list[ParGroup] $pgsTo \leftarrow m[currentLevel].second$
9:     for int $dim \leftarrow nrDimension - 1$; $dim \geq 0$; $dim \leftarrow dim - 1$ do
10:        Stat $feDim \leftarrow$ GETFOREACHDIMENSION($dim$)
11:        $< ds, fes, is, pgsTo > \leftarrow$ CREATEFOREACH($feDim$, $pgsTo$, $m$)
12:        $dimensionStats \leftarrow dimensionStats + ds$
13:        $foreachStats \leftarrow foreachStats + fes$
14:        $indexingStats \leftarrow indexingStats + is$
15:    end for

16:    $innerStats \leftarrow$ TRANSLATEFOREACHSTATS($innerStats$, $m$)

17:    $innerStats \leftarrow indexingStats + innerStats$

18:    for all Stat $s$ in $foreachStats$ do
19:        $s.foreach.stats \leftarrow innerStats$
20:        $innerStats \leftarrow [s]$
21:    end for
22:    return $dimensionStats + innerStats$
23: end function
```

The CREATEFOREACH function creates all three kinds of statements for the ForEach statements that have to be created for the ParGroups in $pgsTo$. Line 5 shows the simple case. If there is only one ParGroup in $pgsTo$, the existing ForEach in s obtains the new ParGroup. Otherwise, the function keeps track of $dimensionVars$ and $indexingVars$ of which the former list contains the variables that will be used in the ForEach statements to indicate the size and the latter the variables that are used to create indices for the ForEach statements.

The loop on line 13 treats the ParGroups in reversed order and creates dimension, indexing, and foreach statements. The indexing statement that expresses the old indexing variable in terms of the new dimensions and indices can only be computed when all indexing variables and dimension variables are known (line 23).

```
1: **function** CREATEFOREACH(Stat $s$, list[ParGroup] $pgsTo$, ParGroupMapping $m$)
2:     list[Stat] $dimensionStats \leftarrow []$
3:     list[Stat] $foreachStats \leftarrow []$
4:     list[Stat] $indexingStats \leftarrow []$

5:     **if** $pgsTo.size = 1$ **then**
6:         $s.foreach.parGroup \leftarrow pgsTo[0]$
7:         $foreachStats \leftarrow [s]$
8:         **return** $< dimensionStats, foreachStats, indexingStats, pgsTo >$
9:     **else**
10:        list[Var] $dimensionVars \leftarrow []$
11:        list[Var] $indexingVars \leftarrow []$
12:        list[ParGroup] $pgsReversed \leftarrow$ REVERSE($pgsTo$)

13:        **for all** ParGroup $pg$ in $pgsReversed$ **do**
14:            Exp $dimensionExp \leftarrow$ CREATEDIMENSIONEXP($dimensionVars$, $pg$,
                 $s.foreach.nrIterations$)
15:            Var $dimensionVar \leftarrow$ CREATEVAR
16:            Var $indexingVar \leftarrow$ CREATEVAR

17:            Stat $dimensionStat \leftarrow$ CREATEASSIGNSTAT( $dimensionVar$, $dimensionExp$)
18:            Stat $foreachStat \leftarrow$ CREATEFOREACHSTAT( $indexingVar$, $dimensionVar$, $pg$)

19:            $dimensionVars \leftarrow dimensionVar + dimensionVars$
20:            $indexingVars \leftarrow dimensionVar + dimensionVars$
21:            $dimensionStats \leftarrow dimensionStats + dimensionStat$
22:            $foreachStats \leftarrow foreachStats + foreachStat$

23:            **if** $pg = pgsReversed.last$ **then**
24:                Stat $indexingStat \leftarrow$ CREATEINDEXINGSTAT($s.foreach.indexingVar$,
                     $indexingVars$, $dimensionVars$)
25:                $indexingStats \leftarrow indexingStats + indexingStat$
26:                $pgsTo \leftarrow [pg]$
27:            **end if**
28:        **end for**
29:        **return** $< dimensionStats, foreachStats, indexingStats, pgsTo >$
30:    **end if**
31: **end function**
```

Function CREATEDIMENSIONEXP creates an expression for the number of parallelism units of a ForEach loop. The function uses the value from the hardware description on line 3, if there are no dimension variables or if the number of units of the ParGroup is definite. This means that a ParGroup has as property nr_units (denoting the exact number of units there has to be) instead of max_nr_units.

```
1: **function** CREATEDIMENSIONEXP(list[Var] $dimensionVars$, ParGroup $pg$, Exp
           $nrIterations$)
2:     **if** $dimensionVars.size = 0 \lor pg.nrUnitsDefinite$ **then**
```

```
3:          return CREATENRUNITSEXP($pg$)
4:      else
5:          Exp $productVars \leftarrow$ MULVARS($dimensionVars$)
        ▷      $productVars$ is 1 if $dimensionVars.size = 0$
6:          return DIV($nrIterations$, $productVars$)
7:      end if
8:  end function
```

The function below creates a statement that defines how the old index is computed from the newly generated index and dimension variables. The first dimension variable is not part of the computation.

```
1:  function CREATEINDEXINGSTAT(Var $oldIndexingVar$, list[Var] $indexingVars$, list[Var]
            $dimensionVars$)
2:      $dimensionVars \leftarrow$TAIL($dimensionVars$)
3:      Exp $e \leftarrow$CREATEINDEXINGEXP($indexingVars$, $dimensionVars$)
4:      return CREATEASSIGNSTAT($oldIndexingVar$, $e$)
5:  end function
```

This function recursively creates an indexing expression from the existing dimension and indexing variables. The $size$ expression is computed by multiplying all dimension variables.

```
1:  function CREATEINDEXINGEXP(list[Var] $indexingVars$, list[Var] $dimensionVars$)
2:      if $indexingVars.size = 1$ then
3:          return EXP($indexingVars[0]$)
4:      end if
5:      Exp $size \leftarrow$MULVARS($dimensionVars$)
6:      Exp $indexingVar \leftarrow indexingVars[0]$
7:      $indexingVars \leftarrow$TAIL($indexingVars$)
8:      $dimensionVars \leftarrow$TAIL($dimensionVars$)
9:      Exp $lowerDim \leftarrow$CREATEINDEXINGEXP($indexingVars$, $dimensionVars$)
10:     return ADD(MUL($indexingVar$, $size$), $lowerDim$)
11: end function
```