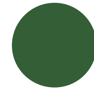
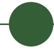
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN





GPU programming made easier

Master's thesis

by

Jacob Jepsen

Department of Computer Science

University of Copenhagen

Advisor: Jyrki Katajainen

Submitted: April 2014

Abstract

We present a framework that aids the programmer in the development of GPU-executable code. We implement a catalogue of common optimizations specific to the GPU architecture. Through the framework, the programmer can semi-automatically apply the optimizations to a computationally-intensive code section and generate an equivalent GPU-executable code section. Based on our experiments, the generated code can be up to one order of magnitude faster than the code from equivalent frameworks and optimized CPU code, and it can attain close to 25% of peak performance of the GPU. We also found that many of the transformations can be performed automatically, which makes our framework usable for both novices and experts in GPU programming. Finally, we contribute with our experiences in creating such frameworks.

Contents

Abstract **i**

1 Introduction **1**
1.1 Related work . . . 4
1.2 Acknowledgements . . . 5

2 Background **6**
2.1 The GPU architecture . . . 6
2.2 The OpenCL programming model . . . 8
2.3 What does semi-automatic mean? . . . 13
2.4 Our approach to a semi-automatic framework . . . 14

3 Compilation: Front end and code generation **18**
3.1 Lexing . . . 19
3.2 Parsing . . . 20
3.3 Internal representation . . . 23
3.4 Code generation . . . 27

4 Generation of host code and kernel code **28**
4.1 Overview of the host code . . . 28
4.2 Generating the kernel code . . . 30
4.3 Generating the `AllocateBuffers` function . . . 32
4.4 Generating the `SetArguments` function . . . 33
4.5 Generating the `InvokeKernel` function . . . 33
4.6 Generating the `RunMain` function . . . 35

5 Transformations **38**
5.1 Defining arguments . . . 39

5.2 Memory coalescing 39
5.3 Placing reusable data in registers 41
5.4 Placing shared data in local memory 44

6 Pattern-matching rules 50
6.1 The TRANSPOSITION transformation 51
6.2 The HOISTTOREG and HOISTTOREGLOOP transformations . 51
6.3 The TILEINLOCAL transformation 52
6.4 Discussion 52

7 Use of the framework 54

8 Performance experiments 57
8.1 Sample programs 57
8.2 Systems under investigation 59
8.3 Benchmark results 60

9 Concluding remarks 64
9.1 Advantages and Disadvantages 64
9.2 Future work 66
9.3 Summary 66

References 68

Appendix A: Paper submitted for PSTI 2014 71

Chapter 1

Introduction

We present a framework for speeding up the process of writing high-performance software for a heterogeneous set of computer architectures. Our focus will be on reducing the development time, while at the same time ensuring high performance.

Multi-core *central processing units* (CPUs) have been common merchandise in desktops since around 2007 and a multi-processing standard, OpenMP [25], was already established for creating programs that execute in parallel across the cores of a CPU. Simultaneously, graphical processing units (GPUs) were made programmable for general-purpose applications, most notably through the CUDA [22] programming model by NVIDIA. In 2009, the Open Computing Language (OpenCL) [16] appeared which added portability, that is, it allowed one to create programs that could execute on both CPUs and GPUs.

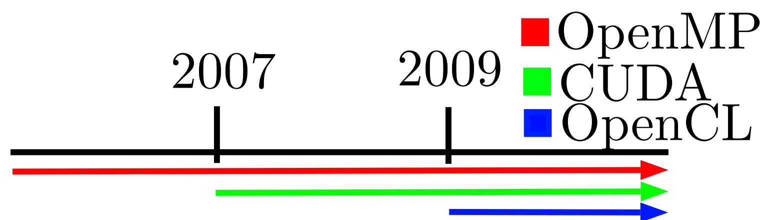


Figure 1.1: Timeline of programming models.

The main advantage of the GPU over the CPU in 2007 was the superior number of floating-point operations per second that one could achieve from a

GPU. However, this only applied to single-precision floating-point numbers, and many applications in high-performance computing use double-precision floating-point numbers for accuracy reasons. GPUs currently achieve gigaflops in double precision beyond that of CPUs, while keeping energy consumption low. They are also resonably priced and the architecture has become easier to program. In brief, GPUs are the platform of choice for many high-performance applications.

While OpenMP manages the tedious details of creating a multi-threaded program, programming of GPUs is still explicit: Programs are written in a low-level C-like language and errors are more the rule than the exception. Thus, it is time-consuming to rewrite a program to run on the GPU. Recently, an OpenMP-like standard, OpenACC [24], has been proposed as an option to make GPU programs easier and faster to write.

The developed standards are good for parallelizing execution of a section of code on the CPU or the GPU, but they do not address optimizations specific to the hardware.

Consider for example an eight-core CPU that uses a 256-bit wide single-instruction multiple-data (SIMD) instruction set, such as the Advanced Vector Extensions (AVX) [11], and a two-level cache system. We discuss two strategies for optimizing an application on such a CPU; we call the first one *parallelization* and the second one *hardware-specific optimization*. When considering parallelization, the maximum theoretical speedup gained through parallelizing a program with OpenMP on our CPU is a factor of eight, since there are eight cores.

Alternatively, one can make efficient use of the hardware components inside a core. For our example CPU, hardware-specific optimizations include usage of the SIMD instructions, which for double-precision floating-point numbers would yield a maximum speedup of four, and better use of the cache which may yield a maximum speedup of two. These hardware-specific optimizations are not always applicable, but they are common in high-performance software. By performing both of these hardware-specific optimizations, we obtain a factor-of-eight speedup.

Each of the two optimization strategies result in a factor-of-eight speedup. One strategy does not exclude the other, so we can combine them to get a factor-of-64 speedup in total. A similar example could be made with hardware-specific optimizations for the GPU. One important difference is that, while the CPU hardware-specific optimizations—such as those mentioned above—should almost always be performed when applicable, it is not the case for GPU hardware-specific optimizations. The goal of the framework presented in this thesis is to make it easy for the programmer to

- transform a section of code into a program runnable on GPUs (parallelization).
- perform optimizations specific to the GPU hardware (hardware-specific optimization).

The outcome is a way of programming that is less time-consuming than hand-coding the program, and because we focus on using the OpenCL API, the resulting programs will be portable and highly efficient. Although it is less time-consuming, we still need an experienced programmer with deep knowledge of the GPU hardware to perform the hardware-specific optimizations.

We define a *transformation* as any rewriting of the source code which results in semantically-identical source code. Therefore, we also denote optimizations as transformations. The framework is based on the idea that a certain set of transformations, S, can be reused in many of the programs that a programmer optimizes. Furthermore, many of the transformations can be performed automatically. For any given program, the programmer will only perform a subset of the transformations in S.

We have implemented a catalogue of transformations in the framework. The programmer then analyses the source code in order to determine which transformation from the catalogue should be performed next and then instructs the framework to perform this transformation, which is done automatically. An example of a sequence of transformations, $T = \{t_1, t_2, t_3, t_4\}$, could be

t_1**:** Parallelize a section of code on the GPU.

t_2**:** Place one of the arrays in the constant memory segment.

t_3**:** Optimize access to another array using the local memory segment.

t_4**:** Save a reused data element in a register instead of loading it from global memory.

For a given program it might not be immediately clear which sequence of transformations will yield the fastest program execution. Determining automatically the optimal sequence has proved itself difficult. Therefore, we leave this task to the programmer who will have to do experimentation on which sequence is the fastest. This is where our framework saves the programmer time, by providing an interactive semi-automatic methodology so that the programmer does not have to hand-code his way through each iteration of experimentation.

Our findings are that:

- Code generated with our framework are up to one order of magnitude faster than code generated by similar frameworks and CPU-executable code.
- For some programs, the code we generate attain close to 25% of the peak performance of the GPU.
- Many transformations can be performed automatically instead of semi-automatically which makes the framework usable for both novices and experts in GPU programming.

Furthermore, we contribute with our experiences in creating such a framework.

The roadmap through this thesis is as follows: In Chapter 2 we cover relevant parts of the GPU architecture and the OpenCL programming model as well as our definition of the term semi-automatic. In Chapter 3 we present the parts of the front end and back end that are needed to parse a piece of code into our internal representation and a module that handles code generation from this representation. We explain the construction of the GPU code in Chapter 4. We go through source-code transformations and pattern matching rules in Chapters 5 and 6. In Chapter 7 we describe how to use our framework and in Chapter 8 we evaluate the performance of the generated code. We conclude the work in Chapter 9. We wrote a paper describing the tool and its capabilities. The paper, which can be found in Appendix A, was submitted to the Fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014).

1.1 Related work

The most desirable approach to utilizing the GPU hardware is a fully-automatic compiler, but this approach is often hindered by unresolvable data dependences and inadequate analyses, which therefore leads to slow programs. Another approach is to use high-level domain-specific languages such as Harlan [10], but this requires us to rewrite the code into this language, which is a time-consuming task. Then there is the library approach, such as the Lapack-like MAGMA library [18] which delivers high-performance matrix-algebra subroutines. This works well if the program uses any of these subroutines, otherwise it does not help.

Approaches similar to ours include the semi-automatic frameworks developed in the 90's which helped the programmer to create parallel applications for the 90's multi-core machines. One such framework is the ParaScope

Editor [14] which helps the programmer with the transformation of a sequential program into a parallel program. The ParaScope Editor performs data dependency analysis and many other analyses on the source code and displays the results to the programmer. The programmer can then easier determine the next transformation to perform from a catalogue of transformations which the framework performs automatically. We found no contemporary frameworks like this or our own.

The idea of the ParaScope Editor differs from ours on two points: First, our approach performs no analyses, so the programmer must determine which transformation to perform next. While analyses would be helpful to the programmer, we wanted to spend our time elsewhere. Secondly, the aim of the ParaScope Editor is the transformation of a sequential program into a parallel program, usually driven by performing transformations that makes a program free of data dependencies that prohibit parallel execution. In our approach, we assume that there are no data dependences that prohibit parallel execution of the program. One could therefore imagine a combination of the two approaches: Use the ParaScope Editor first to minimize data dependences, then use our framework to create an efficient GPU program.

1.2 Acknowledgements

I am grateful to Jyrki Katajainen for excellent guidance, Stefan Sommer for providing a test platform and a sample program, Fabian Gieseke for providing a sample program, and the Munich Centre of Advanced Computing[1] for providing a test platform.

[1] `http://www.mac.tum.de/wiki/index.php/Home`

CHAPTER 2

Background

In this chapter we cover GPU architecture, a programming model for the GPU, and we discuss the term semi-automatic.

2.1 The GPU architecture

In this section we give a brief overview of the GK110 GPU architecture [21]. We only give details on the parts that relate to the transformations that we perform. Many of the components described is this section can be found in other GPU architectures, but they may be called something different.

A GPU is composed of several smaller components, one of which is called a *streaming multiprocessor* (SMX). This component is where the computations take place. It consists of several cores, a register file, local memory, and an L1 cache, see Figure 2.1.

The registers of the register file are distributed among the cores, such that each core can access a set of the registers which are private to that core. The local memory and L1 cache are shared by the cores. The size of the register file limits the amount of registers that each core can utilize.

Multiple threads are used when a parallel program is executed on the GPU. Each of these threads execute in one of the cores. Furthermore, threads are organized in groups of 32, which are called *warps*. Each thread in a warp executes the same instruction. This form of parallelism is called single-instruction-multiple-threads (SIMT).

The warp size has several implications. First, it is important to organize the code such that the 32 threads do not execute different instructions.

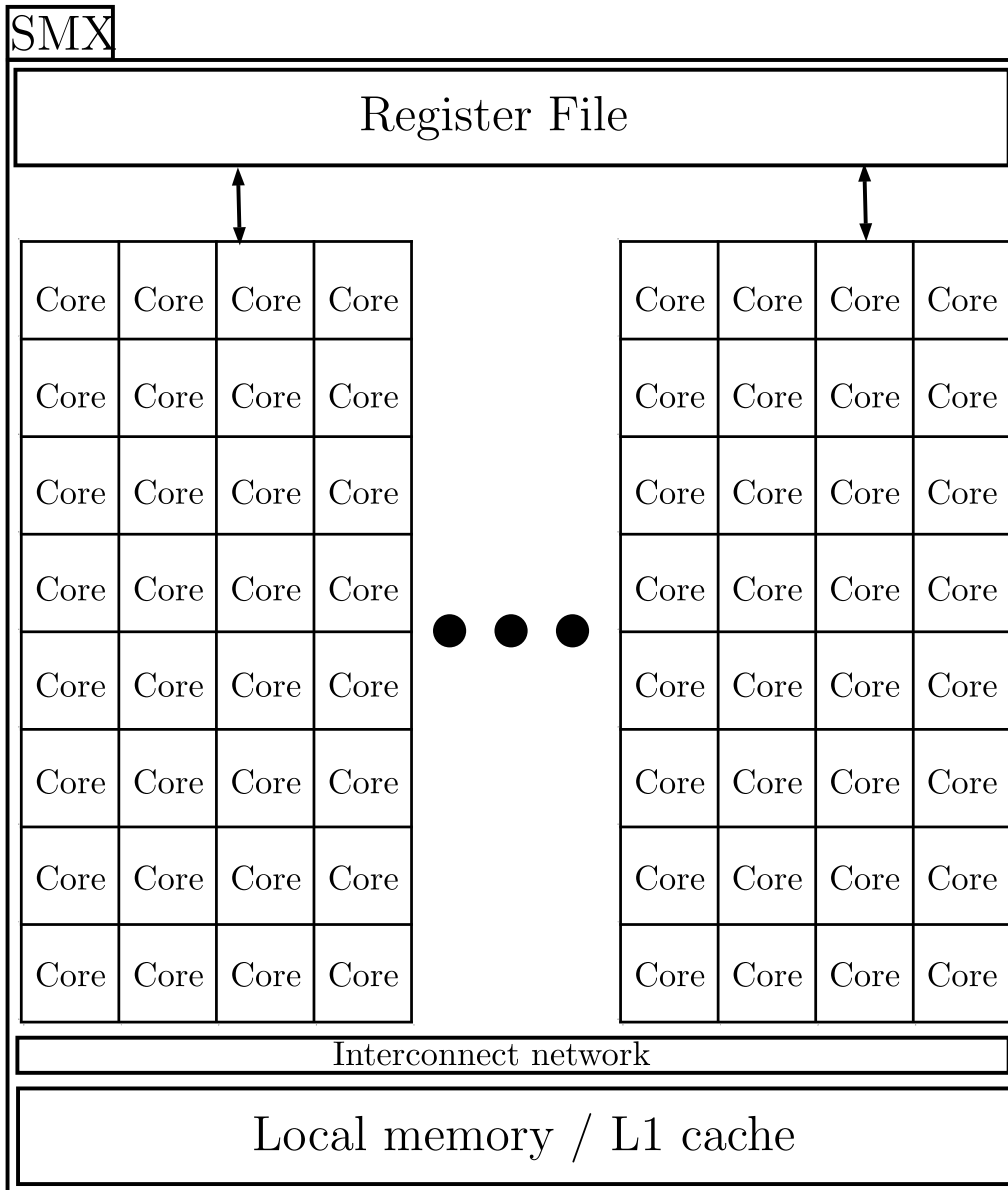


Figure 2.1: Hardware overview of a streaming multiprocessor.

For example, consider code with divergence where the first five threads and the last 27 threads execute two different pieces of code. Secondly, programming languages for the GPU often require the programmer to divide the parallel work into groups and the size of these groups should be a multiple of the warp size.

The GPU is composed of several SMXs, each of which is connected to

a random-access *global memory*, which functions as the main memory of the GPU. Data used during the execution of a program must be transferred to the global memory before the execution begins.

A *latency hiding mechanism* is used to hide the memory latency to the global memory. Warps, that are waiting for data from the global memory, are moved away from the cores and warps that are not waiting start executing. In this way, each SMX has a set of warps, some of which are executing and some of which are waiting.

2.2 The OpenCL programming model

We consider the OpenCL standard [16] as an API that augments an existing programming language, the *host language*, such as C/C++, with functionality to express parallel execution on heterogeneous hardware. The OpenCL standard also defines a language, the *device language*, which is a subset of C99 with extensions to formulate data parallelism and to specify address spaces. For more details on OpenCL programming see [8].

The code written in the host language, the *host code*, always executes on the CPU, the *host*, while the code written in the device language is called the *kernel code* or simply the *kernel*. The kernel may execute on any processor, called the *device*, for which there exists an OpenCL implementation. Such processors include, but not limited to, CPUs from Intel, AMD and IBM, and GPUs from NVIDIA and AMD. The host code and the kernel constitute together the *OpenCL program*, which has the desirable property that it is portable across any hardware architecture having an OpenCL implementation.

The idea of the OpenCL programming model is to offload a computationally-intensive code section to a device that provides faster execution of the code section than the host. Instead, the host code is responsible for three phases: a start-up phase, invoking the kernel and then a shutdown phase, as shown in Fig. 2.2. In the start-up phase we do the following things:

- Allocate a device.
- Allocate memory on the device to hold the data used in the kernel.
- Compile the kernel code.
- Set the arguments for the kernel code.

Then the host enters the phase where it invokes the kernel for execution on the device. This phase also manages data transfer between the host and the device.

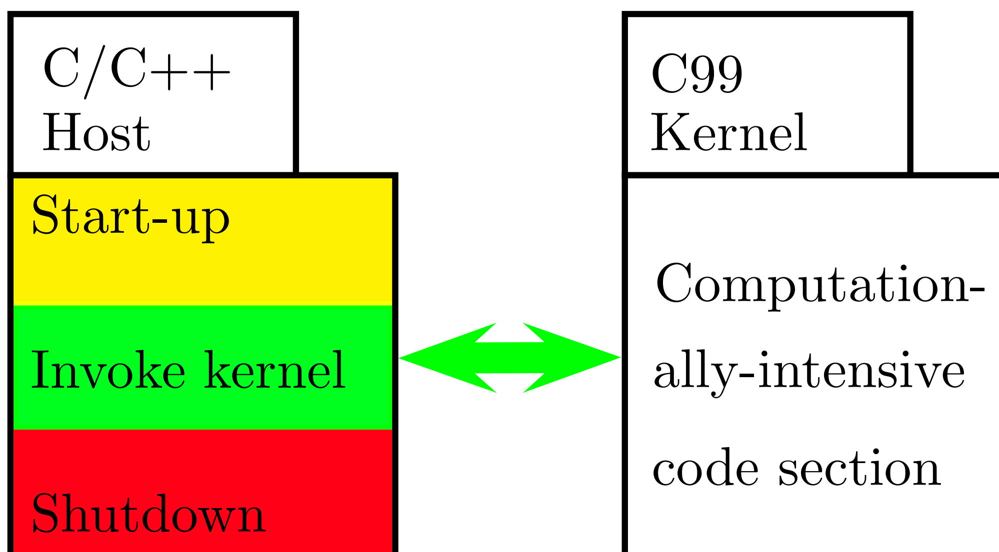


Figure 2.2: On the left we have the C/C++ host code, which manages the three phases: start-up, invoking the kernel and shutdown. The kernel, written in the C99 device language, contains the computationally-intensive code section that is executed on the device.

When the device finishes execution, we enter the shutdown phase which handles deallocation of the device memory, kernel and device, essentially we free all resources that we reserved in the start-up phase.

We now turn to the execution model used in OpenCL. A *parallel loop* is a loop where each iteration of the loop can be executed independently such that the outcome is the same as in the sequential execution of the loop. We only treat `for` loops.

A *nested loop* is a loop that is placed inside the body of another loop. A nested loop may also contain nested loops which are called doubly-nested loops. In general, a loop can be nested any number of times. A *loop nest* is a loop which contains a set of nested loops. In Listing 2.1 we give two examples of how nested loops may occur. We use the loop index to distinguish the loops from each other. In the top example, `j`-loop is nested inside `i`-loop, and `k`-loop is doubly-nested inside the `j`- and `i`-loops. In the bottom example, the `j`-loop is nested inside `i`-loop as is the `l`-loop. The `k`-loop is nested inside `l`- and `i`-loops.

A *perfect loop nest* is a sequence of nested loops where each of the outer loops may only contain a loop in its body, except for the innermost loop which may also contain statements other than loops. A perfect loop nest of *size* m means that there are $m-1$ outer loops plus the innermost

```
  for (size_t i = 0; i < NTEST; i++) {
      for (size_t j = 0; j < NTRAIN; j++) {
          float d = 0.0;
          for (size_t k = 0; k < dim; k++) {
              float tmp = test_patterns[i][k]
                        - train_patterns[j][k];
              d += tmp * tmp;
          }
          dist_matrix[j][i] = d;
      }
  }

for (size_t i = 0; i < NTEST; i++) {
    for (size_t j = 0; j < NTRAIN; j++) {
            ...
        }
    }
    for (size_t l = 0; l < NTRAIN; j++) {
        for (size_t k = 0; k < dim; k++) {
            ...
        }
    }
}
```

Listing 2.1: Two examples of loop nests.

loop in the perfect loop nest. The top example in Listing 2.1 shows a perfect loop nest of size 2 consisting of the `i`- and `j`-loops, and the bottom example shows one of size 1 consisting of `i`-loop. In the latter example the outermost and innermost loop of the perfect loop nest is the same.

In OpenCL, a transformation, which assigns threads the task of executing one of the loop iterations of a parallel loop, takes place. In the following we define several terms, to be needed later, which are part of the OpenCL terminology.

A *work item* is the OpenCL way of describing that which is executed by a thread. Work items are organized in a *grid*. Every work item in the grid is executed in parallel, and hence the size of the grid determines the amount of parallel work. The grid has one, two, or three dimensions, where the product of the length of each of the dimensions is the total amount of parallel work. Each work item has a *global thread identifier*, which is used to distinguish them from one another. The global thread identifier is a tuple which has the same number of elements as the number of dimensions of the grid. The global thread identifier corresponds to the Cartesian coordinate

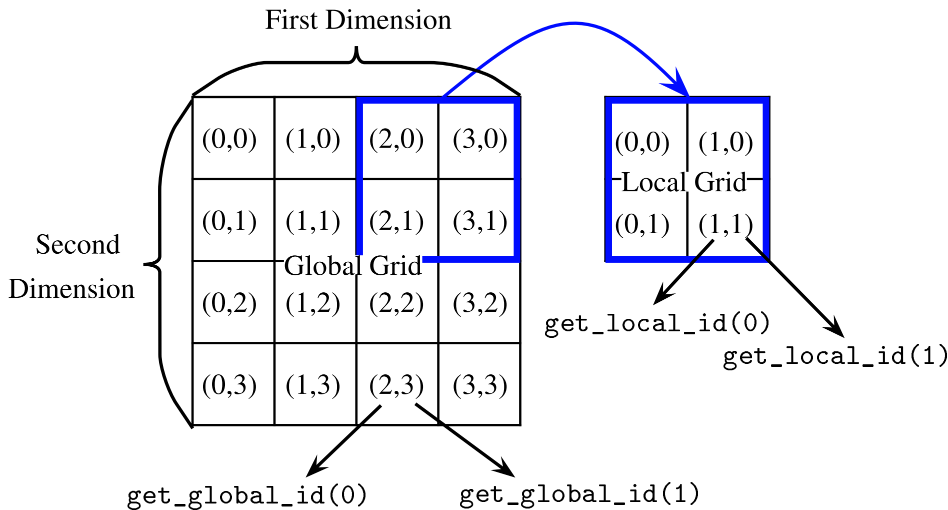


Figure 2.3: An example of a grid, the indexes of the work items in the grid and the functions that return the index.

of the work item in the grid. Special functions are available in OpenCL for getting the thread identifiers.

The parallelization of one, two, or three loops translates directly into using a one-, two- or three-dimensional grid in the execution model. For example, if we want to execute two loops, each of size 100, in parallel, this translates into using a 100-by-100 grid with 10000 work items.

Inside the grid, the work items are furthermore divided into *local grids* or *local work groups*, whose size, the *local work-group size*, has the same number of dimensions as the grid. The division is needed to partition the work among the processors of the GPU. If we make it too small, we have a high degree of parallelism but also high scheduling overhead. If we make it too large, we have a low degree of parallelism and the latency hiding mechanism of the GPU will perform poorly.

In Figure 2.3 we show an example of a 4-by-4 grid. Each position in the grid is identified by a 2-tuple. The first element of the tuple is used as the thread identifier in the first dimension of the grid and the second element for the second dimension. We have defined a local work group which has the size 2-by-2. A important limitation is that the grid size must be divisable by the local work-group size.

The global thread identifier is a tuple whose elements can be accessed using the `get_global_id(dim)` function. The argument `dim`, which can be `0`, `1`, or `2`, specifies which element of this tuple is returned. The function `get_local_id(dim)` does the same but for local thread identifiers. We use

Original C code

```
for (size_t i = 0; i < NTEST; i++) {
    for (size_t j = 0; j < NTRAIN; j++) {
        float d = 0.0;
        for (size_t k = 0; k < dim; k++) {
            float tmp = test_patterns[i][k]
                      - train_patterns[j][k];
            d += tmp * tmp;
        }
        dist_matrix[j][i] = d;
    }
}
```

OpenCL C kernel code

```
float d = 0.0;
for (unsigned k = 0; k < dim; k++) {
    float tmp = test_patterns[get_global_id(1)][k]
              - train_patterns[get_global_id(0)][k];
    d += tmp * tmp;
}
dist_matrix[get_global_id(0)][get_global_id(1)] = d;
```

Listing 2.2: An example of how the original loop nest is transformed into kernel code.

the terms *global thread-identifier function* and *local thread-identifier function* when referring to these functions.

The form of parallelism used in the OpenCL programming model is called *data parallelism*, since each thread executes the same code, which in some sense performs a operation, but each thread performs the operation on different elements of data. This has some similarities to vector instructions, but vector instructions can only be performed for a limited set of arithmetical instructions, whereas a thread in the OpenCL execution model may execute any C statement defined in the device language specification.

In order to transform a perfect loop nest of size m in C into kernel code, we remove the two outermost loops if $m \geq 2$ and the outermost loop if $m = 1$. We replace the loop indices of the removed loops by calls to the `get_global_id(dim)` function.

In Listing 2.2 we show such a transformation using the loop nest of a k-neighbour nearest algorithm. The original code is a perfect loop nest of size 2 and both of the outermost loops are parallelizable. In the kernel code, we have removed these two loops and replaced the indices of the two loops with the global thread-identifier function. We replace `i` with `get_global_id(1)`

and `j` with `get_global_id(0)`. Note that we display the code with two-dimensional array references for readability and brevity.

A clever aspect of the OpenCL programming model is the ability to compile kernel code at run time. This means that we can optimize the kernel at run time for any set of program parameters before executing it.

2.3 What does semi-automatic mean?

Compilers perform a series of transformations automatically without interacting with the programmer. The use of a compiler, from the programmer's view, can be simplified to the following:

1. The programmer gives the source code as input to the compiler.
2. The compiler performs a series of transformations and generates an executable.

A *semi-automatic* compiler-like program performs transformations automatically, but requires interaction with the programmer. Therefore, we refer to these programs as *tools* or *frameworks*. An example of the use of a framework is the following:

1. Give the source code as input.
2. Instruct the framework to perform a transformation automatically.
3. Give the new source code as input.
4. The compiler performs a series of transformations and generates an executable.

Steps 2 and 3 can be iterated by the programmer for a number of times. In this thesis our concern is frameworks that perform source-code transformations that the compiler cannot do.

The term semi-automatic is used in different ways in the literature. For example, in the creation of a multi-threaded version of Geant4, Dong et al. [4] first change the parser to insert keywords in front of variables in the source code. Then they create a tracer to identify shared and read-only variables. They use this information to automatically transform the sequential code into multi-threaded code. In short, they interactively apply different tools to automate the time-consuming aspects of the transformation process.

Zima et al. [29] describe a tool for semi-automatic parallelization based on a component performing classical analyses such as dependency analysis, and whenever the tool has insufficient information, the programmer is

prompted for it. When all the needed information is obtained, the transformations yielding a parallel program are performed automatically. A similar approach, also based on performing analyses, is used by Vandierendonck et al. [28]. In this work the programmer is required to help the compiler by writing annotations to define thread-level parallelism, and to highlight semantic information that analyses cannot find.

Felber [6] describes a different kind of semi-automatic transformations. The programmer specifies "rewriting-rules", i.e. application-specific rules that decide how the decomposition of the work load is handled. Their approach then automatically instruments the Java byte-code with constructs that enable a Java application to execute in parallel across multiple nodes.

Based on the above examples, we summarize a common theme among frameworks for semi-automatic transformations as follows:

- They are designed for time-consuming transformations that can be automated.
- They work on existing source code written in a common (sequential) programming language.
- The programmer works with the framework interactively in order to decide what to do next.

The above frameworks focus on the process of parallelizing a program by means of multi-treading or by distribution across multiple nodes. Our aim is the transformation of programs that are already parallel into programs that utilize the GPU hardware and do so efficiently. The latter is accomplished by performing transformations that optimize the code for specific hardware features of the GPU.

2.4 Our approach to a semi-automatic framework

We focus on applications written in a low-level language such as C/C++. Although we do not need to restrict ourselves to C/C++, it is a language that is close to our target language, which makes the transformations less extensive.

Different classes of code exist. Some code sections have a high degree of parallelism, but must be executed sequentially because of data dependencies, see Fig. 2.4. Other code sections also have data dependencies, but these do not prohibit parallel execution of the program. In the former case, statement

reordering transformations such as loop interchange and loop distribution and transformations to make variables private can be performed to permit parallel execution.

We leave it to the programmer to choose programs that have no data dependencies that prohibit parallel execution and enough parallelism to make GPU execution practical, i.e. faster than CPU execution.

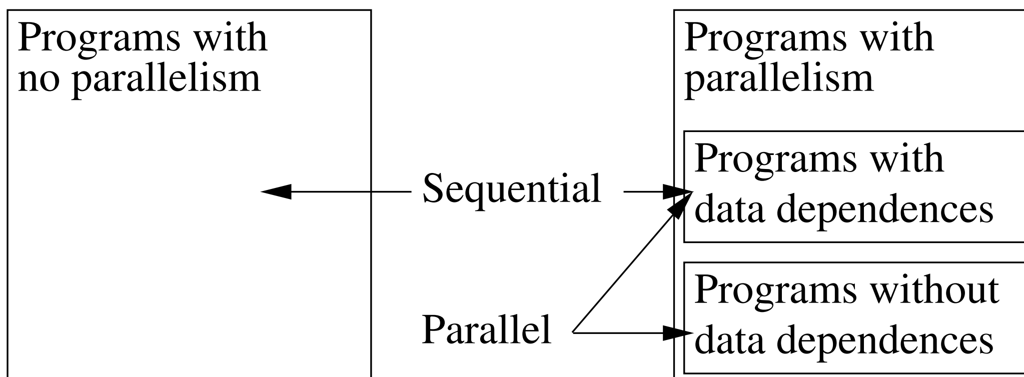


Figure 2.4: We group programs into three groups: *programs with no parallelism*, which must be executed sequentially, *programs with parallelism but without data dependencies*, which can be executed in parallel, and lastly *programs with parallelism and data dependencies*, which sometimes can be executed in parallel and at other times certain transformations must be performed before parallel execution is possible.

We expect the programmer to perform the following tasks before using our framework:

1. Find the code section to be executed on the GPU.

2. If necessary, perform transformations on the code section that will allow for parallel execution of the code section.

When parallel execution of a particular code section is possible, the programmer starts using our framework. The first transformation that must be performed is the generation of the host code used in the three phases shown in Fig. 2.2. This code is referred to as the *boilerplate* code, and it is the backbone for executing the code section on the device.

The programmer subsequently analyses the kernel code and decides, from a catalogue of transformations, which transformation to perform, cf. Fig. 2.5. The programmer then analyses the newly transformed kernel code in order to choose another transformation to instruct the framework to apply.

The programmer continues like this until no more useful transformations from the catalogue can be applied.

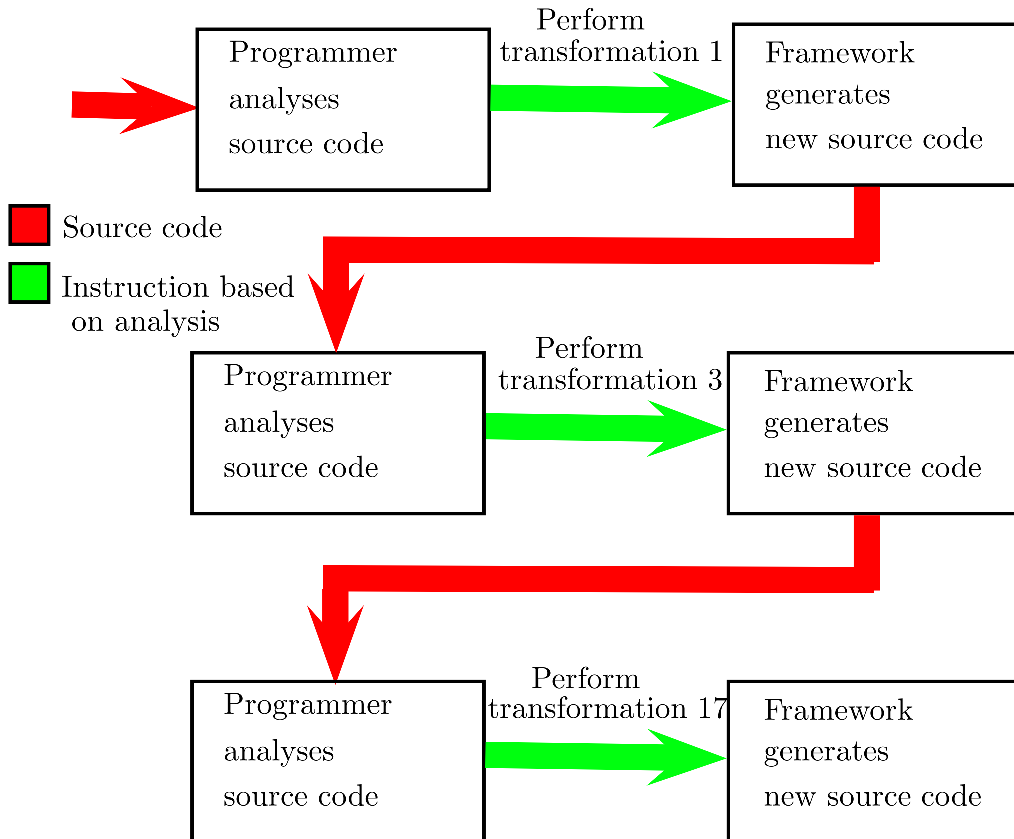


Figure 2.5: An example of the work-flow in our framework. The programmer analyses the source code and decides to instruct the framework to perform transformation 1. The programmer then analyses the resulting source code and decides to perform transformation 3. Finally, the programmer instructs the framework to perform transformation 17.

At this point the generated source code is available to the programmer, so that the programmer can do further transformations by hand. We summarize below the steps that our approach relies on.

1. Find a computationally-intensive code section
2. Perform the necessary transformations by hand to permit for parallel execution.

3. Generate boilerplate code.
4. Iteratively choose which transformations to perform on the kernel code.
5. Perform further transformations by hand if needed.
6. Finally, compile the code with the host-language compiler of choice to get an executable.

We choose to leave the first two steps to the programmer, since it has proved difficult to automate these two steps, and we think that they will not be too time-consuming for the programmer to do, and because the focus in this thesis is on steps 3–4.

Chapter 3

Compilation: Front end and code generation

In this chapter we describe the main components of the front-end module: The lexer, the parser, and the abstract syntax tree (AST). We use this module to parse the loop nest that the programmer wants to execute on GPU hardware. We also present part of the back end namely the code generation.

The module is written using the Python Lex-Yacc (PLY) module [3] to parse a subset of the C programming language [15]. In order to save time, we chose to implement the subset of C that we needed in order to parse a set of example programs. The PLY module allows us to create a context-free grammar by using LR parsing [1].

The goal of the implemented parsing facilities is to create the abstract syntax tree that we need to perform source-code transformations. Other parts of the traditional front end such as type checking is not implemented and non-C syntax may be parsed, hence the programmer must provide a program with grammatically correct types and syntax. An overview of the parts of the typical compiler that we implement in our framework can be seen in Figure 3.1. We postpone the presentation of transformations and pattern matching to Chapters 5 and 6.

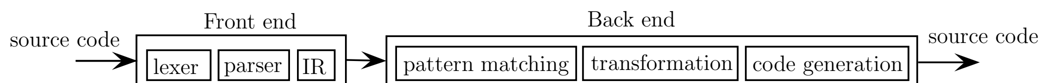


Figure 3.1: An overview of the compiler phases that we implement in our framework.

Keyword tokens					
CHAR	DOUBLE	FLOAT	INT	LONG	SHORT
SIGNED	UNSIGNED	VOID	SIZE_T	FOR	RETURN

Table 3.1: The lexer tokens for the C keywords that we can parse. The regular expression for these is the same as the token name in lower case.

In Section 3.1 we present the lexer and the tokens that we use. Section 3.2 explains how we combine these tokens to form parsing rules. We cover the internal representation of the source code in Section 3.3. Finally, in Section 3.4 we summarize the C code-generation procedure. Parts of the module were based on the PyCParser [5].

3.1 Lexing

As the first step in creating an internal representation for the source code, we break the source code down into the smallest possible components that are still valid C keywords or symbols, known as *tokens*. We make use of regular expressions to represent the pieces of C source code that we convert into tokens.

The lexer tokenizes the source code using the tokens in Tables 3.1 and 3.2. The latter presents a subset of the tokens that the lexer accepts, but most of the arithmetical, bitwise, logical and equality operations as well as increments and delimiters are tokenized. Each token has a corresponding regular expression for the piece of source code that the lexer tokenizes. The regular expression for each keyword token in Table 3.1 is the keyword in lower case. Some tokens deviate from the full C syntax, for example, we do not allow floating-point numbers with e or f, such as 1e-9 and 1.0f, and strings may only contain a subset of the usual C characters, see line 4 of Table 3.2.

In order to define a token using the PLY module, we create a variable starting with `t_`, which signifies that it is a token, and ending with the token name. Then we set this variable equal to the regular expression corresponding to the piece of source that the token represents. If we, for example, want to add the PLUS token to the lexer, we write:

```
t_PLUS = r'\+'
```

All the other tokens are created using a similar syntax.

The lexer automatically matches a piece of source code with the longest token that can represent that piece of source code. For example `+=` is lexed to the PLUSEQUALS token and not to the two tokens PLUS and EQUALS.

Token	Regular expression
ID	`[a-zA-Z_][a-zA-Z0-9_]*`
FLOAT	`([0-9]*\.[0-9]+)|([0-9]+\.)`
INTEGER	`0|([1-9][0-9]*)`
STRING	`\"[a-zA-Z0-9_+.,:; \t!=<>"#@$%&/{}()[]?]*\"`
PLUS	`\+`
TIMES	`*`
AND	`&`
LOGAND	`&&`
LT	`<`
EQ	`==`
EQUALS	`=`
PLUSEQUALS	`\+=`
PLUSPLUS	`\+\+`
LPAREN	`\(`
LBRACKET	`\[`
LBRACE	`\{`
COMMA	`,`
SEMI	`;`

Table 3.2: A subset of the lexer tokens and the corresponding piece of source code, given as a regular expression, that they represent.

3.2 Parsing

A single token is usually not valid C syntax. In this section we present how to combine the tokens using a set of rules that ensure that we are parsing the C code correctly.

A *parsing rule* in PLY is an unambiguous grammar specified with a syntax similar to the Backus-Naur form (BNF). Although the syntax is not the same as the original BNF, we refer to this syntax as the BNF notation. A parsing rule starts with a name followed by a colon and a sequence of terminals and non-terminals signifying what the name may be expanded to. Adding a vertical bar and another sequence of terminals and non-terminals defines a choice in the parsing rule between the two sequences.

As we have implemented many parsing rules in the parser, we only present a few parsing rules at the source code level in order to describe the overall structure of the parser.

To implement a parsing rule we define a function declaration with a name starting with `p_`, signifying that this function is a parsing rule, and

```
def p_native_type(p):
    """ native_type : VOID
                    | SIZE_T
                    | CHAR
                    | SHORT
                    | INT
                    | LONG
                    | FLOAT
                    | DOUBLE
                    | SIGNED
                    | UNSIGNED
    """
    p[0] = p[1]
```

```
def p_type(p):
    """ type : native_type
             | native_type TIMES """
    p[0] = [p[1]] if len(p) == 2 \
           else [p[1]] + [p[2]]

def p_identifier(p):
    """ identifier : ID """
    p[0] = Id(p[1])

class Id(Node):
    def __init__(self, name):
        self.name = name
```

Listing 3.1: Three examples of how to define a parsing rule. These rules are for parsing types and identifiers. A parsing rule is defined using a function whose name starts with `p_`. The first comment gives the parsing rule in Backus-Naur form (BNF). The rest of the function saves the important parts of the parsed C code in the AST.

ending with the name of the parsing rule that we are creating. See for example the function `p_identifier` in Listing 3.1. As the first statement of the function we place a comment, known as the *docstring* in Python. The docstring contains a grammar rule in the BNF as described above. The `p_native_type` function is an example of how to write a parsing rule, which handles the tokens that the lexer has identified as native types. The last part of the function controls what we do with the components of the grammar rule. We wish to build the AST.

Each component of the parsing rule can be accessed through the list `p` of size n using the following mapping:

```
""" Comp_0 : Comp_1 Comp_2 ... Comp_{n-1} """
p = [Comp_0, Comp_1, Comp_2, ..., Comp_{n-1}]
```

Assigning `p[0]` to a value means that the parsing rule may expand, when used as a terminal in another parsing rule, to that value.

A *node* in the AST is a class with member data to represent the syntax consumed by a parsing rule. We do not always want to save the syntax consumed in a parsing rule as a node in the AST, but instead pass it to another parsing rule which then save it in a node in the AST. A node class which just saves the components of a parsing rule as attributes would not be useful in itself. Therefore the node class must be a subclass of the `Node` class which adds extra functionality that enables AST traversal.

```
def p_for_loop(p):
    """ for_loop : FOR LPAREN assignment_expression SEMI binop SEMI
                                 increment RPAREN compound
    """
    p[0] = ForLoop(p[3], p[5], p[7], p[9])

class ForLoop(Node):
    def __init__(self, init, cond, inc, compound):
        self.init = init
        self.cond = cond
        self.inc = inc
        self.compound = compound
```

Listing 3.2: The parsing rule and internal representation of a `for` loop.

In the `p_native_type` function we are saving the token, since no valid C syntax consists of just a type. Since a type may additionally be a pointer we create a list for the type in the `p_type` function. The `p_identifier` function in Listing 3.1 wraps an identifier in an `Id` node, which holds the name of the identifier as a string.

In Listing 3.2 we give a more elaborate example. The grammar rule for a `for` loop starts with the `for` keyword followed by a starting parentheses, a single initialization statement, a semicolon, a binary operation which is the condition, a semicolon, an increment, such as `i++` or `j+=4`, a closing parentheses, and finally the compound which is a list of statements to be executed inside the loop. The `ForLoop` class wraps the `for`-loop components into a node in the AST. While we usually use lower case nouns connected with underscores as names for the parsing rules, the names of the AST node classes are constructed by putting together nouns, often abbreviated, with the first letter in capital.

Unlike a C `for` loop, we can only parse `for` loops with a single initialization statement and a single increment statement. Limitations such as these are not required by our framework, but it reduces the amount of work needed in the parser implementation and it is not syntax that we make use of.

We created parsing rules and classes similar to the `for`-loop example for the other C statements. These parsing rules consist of, but not limited to, comments, function declarations, function calls, assignments, binary and unary operators, array references, variable declarations, and constants. In Listing 3.3 we present a subset of the full grammar that we have implemented in the framework.

Notable constructs for which we did not implement parsing rules in-

```
""" arg_params : typeid COMMA arg_params
               | typeid
               | identifier
               | binop
               | empty
"""
""" assignment_expression : typeid assignment_operator expr
                          | identifier assignment_operator expr
                          | array_reference assignment_operator expr
"""
""" constant : INT_CONST | FLOAT_CONST | STRING_LITERAL """
""" binop : LPAREN binop_expression RPAREN | binop_expression """
""" array_reference : identifier subscript_list """
""" term : identifier
         | constant
         | array_reference
         | function_call
         | unary_expression
"""
""" function_declaration : typeid arglist SEMI
                         | typeid arglist compound
                         | function_call SEMI
"""
""" declaration : typeid SEMI"""
""" typeid : type identifier"""
```

Listing 3.3: A subset of the grammar that we can handle in our framework.

clude structs, `if-then` statements, qualifiers, and `while` loops.

3.3 Internal representation

The parser builds an AST which functions as the internal representation of the source code. The AST has a top node which is a class containing references to other nodes, *children nodes*, which again contain references to other children nodes, or when we reach the leaf level, the nodes contain terminal symbols. Since each node in the AST is a subclass of the `Node` class, it is required to define the `children` function which returns a list of children nodes. The implementation of this function is straightforward: We add the attributes of the node to a list and return the list.

A simple visual representation of such a tree is a nested sequence of lists or tuples which is not reader-friendly. Therefore, we have defined a printing function which places each node on a line and indents this line according to how deep the node is nested in the AST.

The names of the nodes should make sense to those familiar with the C programming language. For example, the `BinOp` node is the internal representation of a binary operation of the form `lval op rval`, while the `ArrayRef` node represent a reference to an array such as `A[i]`.

For example, lines 3-6 of the code in Listing 3.4 has the internal representation shown in Listing 3.5, where we have excluded the header of the loop for brevity.

Each node in the internal representation corresponds to one of the classes that we created in Section 3.2 in order to save the components of a parsing rule. In Listing 3.5 we can also see the type and value of a variable of a node. For example, the first assignment in the AST, lines 2-5, has the operator `=`, the left-hand side, `lval`, is a `TypeId` with the type `float` and the identifier of the variable is `sum`. The right-hand side, `rval`, is a `Constant`. So this corresponds to the statement `float sum = 0.0`.

The indentation signifies the nesting depth of a node in the AST. Take for example the assignment that we just discussed. It is contained in the compound of the second loop. Therefore, it has five parents (not shown) in our internal representation, so its indentation level is six.

To traverse an AST we make use of the *visitor pattern* [7]. We define a generic visitor which performs a pre-order traversal of the AST. A *visitor* is a user-defined class with *visit functions* for one or more nodes of the internal representation. The visit function, for example for an `Assignment` node, defines what should be done when the visitor-pattern traversal encounters an `Assignment` node. Depending on what the programmer wants, the visit function may do a number of different things such as saving values present in the assignment, changing values in the assignment, or replace parts of the assignment with another node.

The visitor pattern traverses the AST and when a node is visited for which the programmer has defined a visitor, then the visit function defined in this visitor is automatically called. The visitor must be a subclass of the `NodeVisitor` class, which contains the visitor pattern, in order to function in this way. If we want to collect some data with the visitor, then we add attributes to the visitor class.

In Listing 3.6 we present a visitor that finds all unique identifiers in an AST. This may be useful for several things such as finding all identifiers in a loop nest or as a sub-visitor in another visitor. One example of this is to use this visitor on an AST for an array reference in order to find the unique identifiers in the subscript of the array reference.

The `Ids` class has one visitor function which is executed when an `Id` node is encountered in the AST. The function accumulates a set of identifier names.

```
1 for (unsigned i = 0; i < hA; i++) {
2     for (unsigned j = 0; j < wB; j++) {
3         float sum = 0.0;
4         for (unsigned k = 0; k < wA; k++) {
5             sum += A[i * wA + k] * B[j + k * wB];
6         }
7         C[wB * i + j] = sum;
8     }
9 }
```

Listing 3.4: A loop nest that expresses a matrix-matrix multiplication. The three matrices, A, B, and C have the dimensions hA $\times$ wA, hB $\times$ wB, and hC $\times$ wC.

```
1  FileAST <top>:
2            Assignment <stmt[0]>: op==
3              TypeId <lval>: type=['float']
4                Id <name>: name=sum
5              Constant <rval>: value=0
6            ForLoop <stmt[1]>:
7                Assignment <stmt[0]>: op=+=
8                  Id <lval>: name=sum
9                  BinOp <rval>: op=*
10                   ArrayRef <lval>:
11                     Id <name>: name=A
12                     BinOp <subscript 0>: op=+
13                       BinOp <lval>: op=*
14                         Id <lval>: name=i
15                         Id <rval>: name=wA
16                       Id <rval>: name=k
17                   ArrayRef <rval>:
18                     Id <name>: name=B
19                     BinOp <subscript 0>: op=+
20                       Id <lval>: name=j
21                       BinOp <rval>: op=*
22                         Id <lval>: name=k
23                         Id <rval>: name=wB
```

Listing 3.5: The internal representation of lines 3-6 of the code in Listing 3.4. The loop header of the third loop is excluded for brevity. The names corresponds to the nodes of the AST. Nesting depth is shown using indentation.

In Listing 3.7 we show a more elaborate example. This visitor is called on all `for` loops in the code and it gathers the following data: A list of all loop indices, a mapping from a loop index to its starting value, a mapping from a loop index to its ending value, and a mapping from the loop index to

```python
class Ids(NodeVisitor):
    """ Finds all unique identifiers"""
    def __init__(self):
        self.ids = set()

    def visit_Id(self, node):
        self.ids.add(node.name)
```

Listing 3.6: A visitor that finds all unique identifiers. The identifiers can be read through the member set `ids`.

the corresponding `ForLoop` node.

Here, we are using the `Ids` visitor to find the identifiers in the initialization statement which is the loop index. Lines `9-11` show how to use a visitor: You initiate the visitor, run the visit function on the AST of choice and read the return data using dot notation.

```python
class LoopIndices(NodeVisitor):
    def __init__(self):
        self.index = list()
        self.end = dict()
        self.start = dict()
        self.Loops = dict()
    def visit_ForLoop(self, node):
        self.Loops[node.init.lval.name.name] = node
        IdVis = Ids()
        IdVis.visit(node.init)
        ids = list(IdVis.ids)
        self.index.extend(ids)
        self.visit(node.compound)
        self.end[ids[0]] = (node.cond.rval.name)
        self.start[ids[0]] = (node.init.rval.value)
```

Listing 3.7: A visitor that finds all loop indices, the start and end values of the indices and creates a mapping from a loop index to the `ForLoop` AST node.

We make several assumption on the structure of the `for` loop, for instance, that the right-hand side of the condition is a variable name. The visitor could easily be extended to work for other cases as well.

To perform transformations, we needed something that would easily allow us to read different kinds of data from the AST as well as making local changes to the AST such as inserting some statements or to replace an identifier. We found that the visitor pattern was a good fit and we are able to do the things that we want by writing a few lines of code. We did not try

to implement other methods that make changes to the AST.

In addition to the AST, we create several global data structures containing direct access to various data that we use when we generate code and perform transformations and pattern matching.

3.4 Code generation

Once we have performed a set of transformations on the AST we print the corresponding C code in order to compile and execute the transformed program. To print, we make use of the visitor pattern and for each node in the internal representation we create a class that generates the string of C code that the node represent. For most nodes this is a straightforward process of first getting the strings of the children nodes and then combining these strings with the appropriate keywords and symbols.

Three example classes are given in Listing 3.8. The C code for an identifier is simply the variable name. In an array reference, we first get the array name and add the subscript encapsulated in brackets.

```
1 def visit_Id(self, n):
2     return n.name
3 def visit_ArrayRef(self, n):
4     s = self.visit(n.name)
5     for arg in n.subscript:
6         s += '[' + \
7         self.visit(arg) \
8            + ']'
9     return s
```

```
def visit_ForLoop(self, n):
    init = self.visit(n.init)
    cond = self.visit(n.cond)
    inc = self.visit(n.inc)
    self.indent_level += 2
    compound = self.visit(n.compound)
    self.indent_level -= 2
    return 'for (' + init + ' ' + cond \
         + '; ' + inc + ')' + compound
```

Listing 3.8: Visitors for the generation of C code corresponding to the internal representation of an identifier, an array reference and a `for` loop.

To turn a `for`-loop node into a piece of C code, we first get the string for the initialization, condition and increment statements. We do this through the use of the visitor pattern as shown in lines `2-4` on the right-hand side of Listing 3.8. Before we get the string for the compound we add an indentation to make the code more readable. Finally, we concatenate the strings and add the `for` keyword and the appropriate extra symbols.

CHAPTER 4

Generation of host code and kernel code

To execute a loop nest on the GPU, we need to generate the host code and the kernel code, cf. Section 2.2. The kernel code is a representation of the loop nest we wish to execute on the GPU. The host code is responsible for setting up the data structures that we need in order to execute the kernel code, as well as setting the arguments for the kernel code and invoking the kernel code.

In Section 4.1 we give an introduction to the different sections of host code that we generate. In Section 4.2 we explain how we generate the kernel code, and in Section 4.3 we cover how the code for allocating buffers is generated. Then, in Sections 4.4 and 4.5 we present how the code for setting the kernel arguments and invoking the kernel code is generated. Finally, in Section 4.6 we go through the function used by the user to start the GPU execution of a loop nest.

4.1 Overview of the host code

In the host code, we set up a series of data structures that are mandatory in order to execute kernel code on the GPU. A *buffer* is a chunk of memory on the device. In many regards, allocating a buffer is just the OpenCL equivalent of allocating a chunk of memory in C with the `malloc` function, and we need to create these buffers in order to store data on the GPU. We always allocate two chunks of memory for the same piece of data: one chunk in the main memory and one chunk in the device memory, where the latter

mirrors the former.

The OpenCL API defines several functions which can be used to set up data structures, manage data transfer and to invoke the kernel code. We make use of those described below.

- `clCreateBuffer`: Allocates a buffer. The arguments are a GPU context, a flag denoting the access mode of the buffer, namely, whether it is write only, read only, or read and write, the size of the buffer, the host pointer which can be `NULL`, and a reference to where the error code is saved. The return value is a buffer object.
- `clSetKernelArg`: Sets an argument to a kernel function. The arguments are the kernel function, the position of the argument in the kernel function, the size of the argument, and a reference to the argument. The return value is an error code.
- `clEnqueueNDRangeKernel`: Invokes a kernel function. The arguments are a command queue for the GPU, the kernel function, the number of dimensions of the grid, an offset signifying where the numbering of the thread identifiers start, the grid size, the local work-group size, and the last three arguments which concern the use of OpenCL events, which we do not make use of in the framework. The return value is an error code.
- `clFinish`: Synchronizes the host with the device. The argument is a command queue for the GPU. The return value is an error code.
- `clEnqueueReadBuffer`: Reads data from a buffer to a memory chunk on the host. The arguments are a command queue for the GPU, a buffer, a boolean flag denoting if the read is blocking, an offset in the buffer, the size of the data, a pointer to a memory chunk. The last three arguments concerns the use of OpenCL events. The return value is an error code.

The host code that we generate is divided into five functions which have a specific task to perform:

- **GetKernelCode**: Returns a string representation of the kernel code.
- **AllocateBuffers**: Calculates the size of the buffers needed by the kernel code and allocates space for these buffers.
- **SetArguments**: Sets the arguments for the kernel code. The arguments include the buffers used and the values needed in the kernel code.

- **InvokeKernel**: Invokes the kernel code on the GPU, and transfers the data that we need from the GPU.
- **RunMain**: Allocates the GPU and compiles the kernel code, then calls the four functions above.

In each function we are using functions from the OpenCL API to accomplish the given task. The code for allocating a GPU and compiling kernel code is general and can be reused across multiple GPU programs. Hence, we do not generate this code, but keep it in a file which we include in the host code.

In addition to the aforementioned functions, we create a set of global variables which contain information about the dimensions of the arrays, the sizes of the arrays, the pointers to the host-side data, the buffers, and the kernel code.

The code is generated by creating AST nodes, discussed in Section 3.3, and putting these nodes together to form an AST. Then we print the AST to a file using our code-generation module.

In the following we present several visitors used to create the data structures that we need in order to generate the host code as well as visitors that rewrite certain parts of the loop nest. We mainly use figures or text to explain these visitors as we think they are more understandable than the source code.

4.2 Generating the kernel code

Our framework takes as input a loop nest. We use this loop nest as the basis for the kernel code and, hence, every statement in this loop nest must conform to the standard of the device language [16].

The loop nest may have several outer loops which are parallel. As a first step, we determine how many of the outer loops we parallelize. We require that at least one loop, namely the outermost loop, can be executed in parallel. Our framework only support parallelization of either one or two outer loops. In general we assume that the two outermost loops, if such a pair exists, can be run in parallel, but we also provide a mechanism for specifying that only the outermost loop can be run in parallel. If there are two or more outer loops, then we parallelize the two outermost loops, otherwise we parallelize the outermost loop.

We create the kernel code by making a copy of the original loop nest and then removing the loops that we parallelize on the GPU. The kernel code cannot be given as a loop nest, but must be encapsulated in a function, so

Original loop nest

```
float *A;
float *B;
float *C;
unsigned wA;
unsigned hA;
unsigned wB;
for (unsigned i = 0; i < hA; i++) {
  for (unsigned j = 0; j < wB; j++) {
    float sum = 0;
    for (unsigned k = 0; k < wA; k++) {
      sum += A[i][k] * B[j][k];
    }
    C[i][j] = sum;
  }
}
```

Generated kernel function

```
__kernel void MatMul(
  __global float *A,
  __global float *B,
  __global float *C,
  unsigned wA,
  unsigned hst_ptrA_dim1,
  unsigned hst_ptrB_dim1,
  unsigned hst_ptrC_dim1) {
    float sum = 0;
    for (unsigned k = 0; k < wA; k++) {
      sum += A[get_global_id(1)][k]
                        * B[get_global_id(0)][k];
    }
    C[get_global_id(1)][get_global_id(0)] = sum;
  }
```

Listing 4.1: An example of a loop nest and the corresponding kernel function. The arguments to the kernel function are the variables which are not declared in the original loop nest as well as the length of the innermost dimension of the arrays. The types of these variables must be declared before the loop nest.

we find the arguments for this function using the visitor pattern. We refer to this function as the kernel function, which we use interchangeably with kernel code.

First, we create a visitor that visits all `Id` nodes and returns the set of identifiers. Then we create a visitor which visits all `TypeId` nodes, which represents a statement which give the type and the identifier of a variable, such as the left-hand side of `float sum = 0.0`. The identifiers in the set returned by this visitor are considered local to the loop nest. The set of arguments for the kernel function are the set of all identifiers minus the set of local identifiers, and the length of the innermost dimension of every two-dimentional array. We need the latter argument for the calculation of the array subscripts.

We require the type of each argument, but we cannot identify the types from the loop nest. Therefore, we require the user to declare the types in front of the loop nest, see Listing 4.1, in which we show an input loop nest and the corresponding kernel function. If the type of an identifier is not given, then we cannot proceed and we raise an error.

For any pointer argument of the kernel function we add a qualifier which tells the run-time system where on the GPU to place the data pointed to by the pointer. By default we place all data in the global memory segment by adding the `__global` qualifier.

The device language only supports one-dimensional arrays. Therefore, we create a visitor which rewrites the array subscripts to a one-dimensional

subscript (not shown in Listing 4.1). We do this by multiplying the outer subscript with the innermost dimension of the array and add the inner subscript.

Variables in the header of the loops that we parallelize are not added as arguments since these loops are deleted from the kernel code. Finally, we create a visitor which replaces the loop indices of the loops, which we deleted, with the `get_global_id` function.

We create the `GetKernelCode` function and hard-code the kernel code as the return value using the `stringstream` class [2]. This means that we first create a `stringstream` object and then add each line of kernel code using the `<<` operator.

4.3 Generating the `AllocateBuffers` function

We start by determining the set of buffers that we need to allocate. We iterate through the arguments of the kernel function and add each pointer argument to the set. Before allocating the buffers we calculate their size and save it in a global variable. We generate statements where the left-hand side is the identifier of the global variable and the right-hand side is the product of the dimensions of the array and the size of the data type, see lines 3–4 in Listing 4.2 for an example.

To allocate a buffer we use the `clCreateBuffer` function. To find the flag argument, we examine how the arrays are accessed in the kernel code. First we create two sets: Arrays that we write to and arrays that we read from. We create a visitor which adds the identifier of an array reference to the write set if that array reference is the left-hand side of an assignment, and to the read set if the array reference appears somewhere on the right-hand side, for example in a binary operation, in a function argument, or simply by itself, or possibly inside the subscript of an array reference appearing on the left-hand side of an assignment. Depending on whether an array is in both the read and write sets, only the write set, or only the read set, the corresponding buffer will use the read-and-write flag, the write flag or the read flag.

For any buffer with a read access mode we add a `CL_MEM_USE_HOST_PTR` flag which tells the run-time system to copy the data from the host memory to the device memory when execution of the kernel code begins.

A portion of the `AllocateBuffers` function for the matrix-multiplication loop nest can be seen in Listing 4.2. In addition to allocating a buffer for each array in the loop nest, we check the error code in order to ensure that the allocation was successful. The buffer objects are saved in the global

```
1 void AllocateBuffers() {
2   hst_ptrA_mem_size = hst_ptrA_dim2 * (hst_ptrA_dim1 * sizeof(float));
3   hst_ptrC_mem_size = hst_ptrC_dim2 * (hst_ptrC_dim1 * sizeof(float));
4   // More size calculations
5   cl_int oclErrNum = CL_SUCCESS;
6   dev_ptrA = clCreateBuffer(context,
7                             CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
8                             hst_ptrA_mem_size, hst_ptrA, &oclErrNum);
9   oclCheckErr(oclErrNum, "clCreateBuffer dev_ptrA");
10  dev_ptrC = clCreateBuffer(context,
11                            CL_MEM_WRITE_ONLY,
12                            hst_ptrC_mem_size, NULL, &oclErrNum);
13  oclCheckErr(oclErrNum, "clCreateBuffer dev_ptrC");
14  // More buffer allocations
15 }
```

Listing 4.2: The `AllocateBuffers` function for the matrix-multiplication loop nest. In lines 3–4 we calculate the sizes of the buffers before allocating them in lines 7–12. The memory access mode of each buffer was found using a visitor on the original loop nest.

variables `dev_ptrA` and `dev_ptrC`.

4.4 Generating the `SetArguments` function

For each argument to the kernel function we generate a corresponding call to the `clSetKernelArg` function, see Listing 4.3, which shows a portion of the `SetArguments` function for the matrix-multiplication loop nest. At the end we check that all arguments are set successfully.

4.5 Generating the `InvokeKernel` function

In order to invoke some kernel code we need to define the grid size. For the parallelization of two loops, we use a two-dimensional grid where the lengths of the dimensions are equal to the number of iterations of the two loops. As the offset argument we use the starting value of the two loops that we parallelize.

For example, in the matrix-multiplication loop nest (see Listing 4.1) we parallelize the two outer loops with the loop indices `i` and `j`. The number of iterations of the two loops are `hA` and `wB`, so the grid size of the kernel code is `wB`-by-`hA`, where the first dimension is iterated first. We observe that the offset is zero in both dimensions. The grid size is divided into two-dimensional

```
1 void SetArguments() {
2   cl_int oclErrNum = CL_SUCCESS;
3   int counter = 0;
4   oclErrNum |= clSetKernelArg(MatMulKernel, counter++,
5                  sizeof(cl_mem), (void *) &dev_ptrA);
6   oclErrNum |= clSetKernelArg(MatMulKernel, counter++,
7                  sizeof(unsigned), (void *) &hst_ptrB_dim1);
8   oclErrNum |= clSetKernelArg(MatMulKernel, counter++,
9                  sizeof(unsigned), (void *) &wA);
10  // Set more arguments
11  oclCheckErr(oclErrNum, "clSetKernelArg");
12 }
```

Listing 4.3: The `SetArguments` function for the matrix-multiplication loop nest. Lines 5–6 set the first argument for the kernel function, lines 7–8 set the second one and so on.

groups, the local work groups, and we give these groups the default size of 16-by-16.

In Listing 4.4, we show an extract of the `InvokeKernel` function for the matrix-multiplication loop nest. In lines 3–5 we set the global and local work sizes before invoking the kernel code. Then we synchronize with the GPU. Next, we read back data from the buffers which are in the write set, as described in Section 4.3. This approach may be more conservative than needed, but we do not know which arrays will be used after the kernel code has finished executing. Therefore, we assume that all arrays which were written to will be used, even though it may be that in some programs only a subset of these arrays is actually used.

After adding something to the command queue of a GPU, it is important to synchronize the GPU with the host, since the execution would otherwise continue concurrently. For example, if the kernel code is still executing on the GPU, then we want to wait until the code has finished, so that we transfer back the correct data. As always we check that the calls to the OpenCL functions are successful. If they are not, we stop the program.

Although one of the goals of the framework is to generate code without errors, the user is still able to raise certain types of errors, such as transferring more data to the GPU than what the GPU has memory for. Errors such as these are checked at run-time and the proper error response is returned to the user.

```
1 void Exec() {
2   cl_int oclErrNum = CL_SUCCESS;
3   size_t global_offset[] = {0, 0};
4   size_t grid_size[] = {wB, hA};
5   size_t local_worksize[] = {16, 16};
6   oclErrNum = clEnqueueNDRangeKernel(command_queue, MatMulKernel, 2,
7           global_offset, grid_size, local_worksize,
8           0, NULL, NULL);
9   oclCheckErr(oclErrNum, "clEnqueueNDRangeKernel");
10  oclErrNum = clFinish(command_queue);
11  oclCheckErr(oclErrNum, "clFinish");
12  oclErrNum = clEnqueueReadBuffer(command_queue, dev_ptrC, CL_TRUE,
13                          0, hst_ptrC_mem_size, hst_ptrC,
14                          0, NULL, NULL);
15  oclCheckErr(oclErrNum, "clEnqueueReadBuffer");
16  oclErrNum = clFinish(command_queue);
17  oclCheckErr(oclErrNum, "clFinish");
18 }
```

Listing 4.4: The `InvokeKernel` function for the matrix-multiplication loop nest. Lines 3–5 set the offset, grid size, and local work-group size. The rest of the code invokes the kernel code and tranfers back the data of the buffers in the write set.

4.6 Generating the `RunMain` function

A *context* is an OpenCL data structure which allows us to invoke a kernel on the GPU. In the `RunMain` function we begin by allocating a context for the GPU on which we wish to execute the kernel code. For this purpose we have created a general function, `StartUpGPU`, which we do not generate, but simply include in the host code from another file. The `StartUpGPU` function queries the computer to see if it has any available GPUs, and if it does, it allocates a context for one of them. Then we set up a command queue which is an additional structure that we need in order to invoke the kernel code on the GPU.

Next, we compile the kernel function that we wish to execute on the GPU. We have created the `compileKernel` function to do this and it is also a generic function which we include in the host code. As arguments it takes the name of the kernel function, the kernel code, and a reference to a kernel object where the compiled kernel is saved.

Then we call the `AllocateBuffers`, `SetArguments`, and `InvokeKernel` functions, see Listing 4.5.

The `RunMain` function takes as arguments the global variables of the

```
void RunMain(
  float * arg_A, size_t arg_hst_ptrA_dim1, size_t arg_hst_ptrA_dim2,
  float * arg_C, size_t arg_hst_ptrC_dim1, size_t arg_hst_ptrC_dim2,
  float * arg_B, size_t arg_hst_ptrB_dim1, size_t arg_hst_ptrB_dim2,
  unsigned arg_wB, unsigned arg_wA, unsigned arg_hA
  ) {
  hst_ptrA = arg_A;
  hst_ptrA_dim1 = arg_hst_ptrA_dim1;
  hst_ptrA_dim2 = arg_hst_ptrA_dim2;
  wB = arg_wB;
  // Save the rest of the arguments
  StartUpGPU();
  CompileKernel("MatMul", GetKernelCode(), &MatMulKernel);
  AllocateBuffers();
  SetArguments();
  InvokeKernel();
}
```

Listing 4.5: The RunMain function for the matrix-multiplication loop nest. Lines 7–11 save all the arguments in global variables. The rest of the code allocates a GPU, compiles the kernel code, allocates buffers, sets the kernel function arguments, and invokes the kernel code.

loop nest that we found in Section 4.1 as well as the dimensions of the arrays. This is all that we need in order to set up the code for GPU execution. We have designed the host code in this way, because then the user only needs to run one function from his code, namely the RunMain function, and the user is therefore not bothered with the details of allocating a GPU and buffers and so on.

An example of the RunMain function for the matrix-multiplication loop nest is shown in Listing 4.5. Lines 7–11 saves all the arguments of the function in global variables so that they are available in all functions. Then we call the functions that we have described above.

To initiate the GPU execution of the original loop nest, the user needs to include the host code and then replace the loop nest in the user's code with a call to the RunMain function. While the user might execute the original loop nest any number of times, the RunMain function can only be called once. This is not a necessary restriction, but for our purposes we only need to run the kernel code once. The extra code that we need to allow for multiple calls to the RunMain function can easily be generated. An important element in this is that we enable kernel function arguments to be changed in between two invocations.

In this chapter we explained how we generated the necessary parts of

the host code that we need to execute a piece of code on GPU hardware. There are many ways to set up the host code, for example by using different functions from the OpenCL API, but above we have explained the set up that we have chosen. We have split the set up into sections which correspond to a number of steps that you need to go through in order to carry out GPU execution of a piece of code. Our goal with the split is to provide good readability as well as a simple software architecture, which is useful, for example, if one wants to merge two GPU programs.

Chapter 5

Transformations

In this chapter we present the source-code transformations which optimize the code for the GPU hardware. A *profitable* transformation speeds up the code by any factor greater than one. Some programs may not be executed on a GPU because they, for example, use more memory than what is available on the GPU, or if the number of iterations of the loops that we parallelize is not a multiple of the local work-group size. There are other transformations which are not profitable, but are useful for enabling programs to be executed on a GPU. We have only focused on profitable transformations.

In general, the transformations are aimed at any GPU with a corresponding OpenCL implementation, but the transformations are mostly intended for NVIDIA GPUs, specifically for the Kepler GK110 architecture [21]. Due to similarities in the architectures, the transformations should in theory be profitable on a recent AMD GPU architecture, or at least, not unprofitable.

Some of the transformations involve rewriting both the host code and the kernel code. Our transformations are applicable to one-dimensional or two-dimensional arrays. A transformation will not be performed on, for example, a three-dimensional array even if it would be profitable to do so.

We explain the transformations in text instead of using the source code and give some examples which show the code before and after the transformation. The implementation of the transformations is carried out by jumping to the corresponding place of interest in the AST and performing a set of local changes to existing nodes, or inserting additional statements into the AST.

In Section 5.1 we explain how we define certain kernel function arguments as constants, while we in Section 5.2 describe how we perform array transposition. Then, in Sections 5.3 and 5.4 we present the transformations for placing reusable data in registers and shared data in the local memory. The goal of the transformations in the last two sections is to move data as close as possible to the processing units, where close means with minimum latency.

5.1 Defining arguments

To *define* a variable means that we propagate the value of the variable into the kernel code at compile time. This is similar to using the `#define` statement in C instead of using a constant global variable. The non-pointer arguments of the kernel function can be defined when we compile the kernel code. This provides more information to the OpenCL compiler which allows it to perform more optimizations, such as loop unrolling [20].

We call the transformation DefineArg. To perform it, we iterate over the kernel function arguments and for each argument which is not a pointer, we delete it from the list of arguments. Then, we generate host code which adds a string containing the compiler option to define the argument that was removed. The string generated for each argument is accumulated in a global variable, which is passed to the OpenCL compiler.

This transformation is similar to constant propagation [13] which can be found in many contemporary compilers. These compilers usually perform the transformation offline. Since we are able to compile the kernel code at run time, we can also define variables that was initialized from command-line arguments or files.

For some programs, a kernel function argument can change between two kernel invocations, which means that we cannot define this argument at compile time. We provide a function which allows the user to exclude an argument when this transformation is performed.

The transformation is not specific to any hardware as such, but we include it as it usually gives a noticable speedup. The running time of this transformation is linear in the number of kernel function arguments.

5.2 Memory coalescing

Kernel codes exhibit several common memory access patterns. One of them is for each thread to access a different element of data from the global memory

in the same instruction. If these elements are not located consecutively in the global memory, a performance penalty occurs. The reason for this is that, on the GPU, a memory access is an instruction which accesses several consecutive memory locations. In the case of a memory read, a small chunk of data is read for each instruction, much like when a CPU reads an element of data, it brings the cache line containing the element to the CPU cache. If we do not use all the elements of a chunk, then we are essentially wasting memory bandwidth. The worst case is when none of the data elements that the threads access are in the same chunk of data. Then each thread will read one chunk and use only one element from this chunk. This is called *uncoalesced memory access*.

Let us look at an example. Say we have four threads in the local work group and that the memory is accessed in chunks which contain four data elements, that is, four 32-bit values. These numbers are not from any real GPU. If the threads execute an instruction, such that each thread accesses one value and if the values are all in the same chunk, then a single memory transfer for that chunk is issued. If the values are located in two different chunks, then two memory transfers are issued. In the worst case the number of issued memory transfers is equal to the number of threads.

Not only are we wasting bandwidth, but the memory transfers are performed sequentially resulting in a latency penalty as well.

In order to obtain coalesced memory access we interchange the two subscripts of a two-dimensional array. The effect of this is a transposition of the memory access pattern, which means that we must also transpose the data in the array, much like when one transposes a matrix.

This is related to what is known in litterature as the array of structs (AoS) to struct of arrays (SoA) transformation. When the data is layed out as an AoS, the stride between the same member of two consecutive structs is the size of the struct. To obtain coalesced memory access, we can convert to the SoA layout where the stride is one. This transformation is usually performed by the programmer, since it, presumably, is difficult to analyse if it allowed and if it is profitable in the general case.

The Transposition transformation takes a list of arrays as its argument. In Chapter 6 we explain how to create this list. For each array in the list we swap the dimensions of the array. We create, in the host code, a new array of the same size as the original. Then, if the array is not in the write-only set, we add code which copies the data from the original array to a new array in the transposed layout. Finally, we set the new array as the source to the corresponding kernel function argument.

If the array is in the write set, then after the data is transferred back from the GPU we transpose it, in order to give the data the layout that the

rest of the user's code expects. The running time of the transformation is linear in the length of the list given as argument.

This transformation is not a GPU-specific transformation, because it should also be profitable on the CPU architecture, where it rearranges the data to enable vectorization and to improve data locality.

This transformation uses extra memory, because we need a chunk of memory that is of the same size as the original. For some programs this extra memory usage may be too large to perform this transformation. This can be avoided by invoking a kernel function on the GPU which transposes the array in place.

5.3 Placing reusable data in registers

In some situations we are reading the same data from the global memory in each iteration of a `for` loop. The transformation to be described is about reading the data once, before the loop, saving it in a variable, and then reading this variable where we read from the global memory before. This transformation has some similarities to loop hoisting, also called loop-invariant code motion [20], which moves loop-invariant code, for example a computation, outside the loop.

The transformation can be performed on the GPU because it, unlike the CPU, has a large amount of registers per thread. On the Kepler GK110 architecture each thread can use a maximum of 255 registers. Some of these registers are not being used in the kernel code, so we can utilize them by performing this transformation. This transformation only changes the kernel code.

The transformation works when we are reading the same data from the global memory inside one loop, or inside two loops. If the global memory reads are inside only one loop, then we perform the HoistToReg transformation. We proceed by creating a variable outside the loop. We assign to the variable the value which is read from the global memory. Then we replace the global memory read inside the loop with a read to that variable.

In Listing 5.1 we see a portion of some kernel code from an N-body simulation. In lines 5–7 of the original kernel code we see that we are reading the same data, the position and mass of a body, from the global memory in each iteration of the `j`-loop. In the transformed kernel code, we start by creating an assignment where the left-hand side is a temporary variable and as the right-hand side we use a copy of the array reference to the global memory location that we read. These assignments are placed at the beginning of the kernel code before any `for` loops. We do this for each array reference that

Original kernel code

```
1  __kernel void NBody(
2          __global float *Mas,
3          __global float *Pos,
4          __global float *Forces
5    ) {
6    for (unsigned j = 0; j < N; j++) {
7        float a_x = Pos[0][get_global_id(0)];
8        float a_y = Pos[1][get_global_id(0)];
9        float a_m = Mas[get_global_id(0)];
10       ...
11   }
12   ...
13 }
```

Transformed kernel code

```
__kernel void NBody(
        __global float *Mas,
        __global float *Pos,
        __global float *Forces
  ) {
  float Mas0_reg = Mas[get_global_id(0)];
  float Pos0_reg = Pos[0][get_global_id(0)];
  float Pos1_reg = Pos[1][get_global_id(0)];
  for (unsigned j = 0; j < N; j++) {
      float a_x = Pos0_reg;
      float a_y = Pos1_reg;
      float a_m = Mas0_reg;
      ...
  }
  ...
}
```

Listing 5.1: Example of the transformation which saves reusable data in registers. In lines 7–9 of the original kernel code we read the same data from the global memory in each iteration of the `j`-loop. In the transformed kernel code we read the data from the global memory once in the three lines just before the beginning of the `j`-loop. Then we reuse this data inside the `j`-loop.

Transformation 1 HoistToRegLoop

Input: A list, A, of 2-tuples containing an array reference and a loop.

1: **for** $(ref, loop)$ in A
2: Allocate a temporary array with length equal to the number of iterations of loop.
3: Copy *loop* in front of the other loops. Only copy once for every distinct loop.
4: Create assignment from *ref* to the temporary array inside the new loop.
5: Replace *ref* with reference to the temporary array inside the other loops.
6: **end for**

---

we perform the transformation for. Finally, we replace the right-hand side of the assignments inside the loop with the temporary variables.

In the case that the global memory reads are inside two loops we perform a slightly different transformation, which we call HoistToRegLoop, see Transformation 1 for a detailed description. First, we allocate a temporary array, inside the kernel code, with a length equal to the number of iterations of the second loop. Then we copy the second loop in front of the other loops and place in it an assignment where the left-hand side is a refer-

ence to the new array with the loop index of the second loop as the subscript. The right-hand side of the assignment is a copy of the reference to the global memory location that we read. The old array references to global memory inside the two loops are replaced by the corresponding references to the new arrays.

In Listing 5.2 we show an example of the second type of the transformation. First, in lines 6–8 of the transformed kernel code, we generate the three allocations of the temporary arrays `X_reg`, `Y_reg`, and `Z_reg`. Then we copy the `d`-loop from line 7 of the original kernel code and place it in front of the other two loops. Inside the new `d`-loop we read the values from the global memory and save them in the temporary arrays. Then, in lines 16–18 of the transformed kernel code we replace the right-hand side of the assignments with a reference to the corresponding temporary array.

This transformation is not profitable when it causes the kernel code to use more registers than what is available. This can happen for example if the second loop has a very large number of iterations. In Chapter 6 we present condition that we check to determine whether we can do the transformation at all, and if so, if it is profitable.

The transformation is performed on array references. It takes as argument a dictionary with keys equal to the identifiers of the arrays. We number each reference to the same array with a number corresponding to the order in which they appear in the kernel code. The values of the dictionary are lists of the indices of the references which we perform the transformation on. For example, when we performed the transformation in Listing 5.2 we gave the dictionary `{'X' : [0], 'Y' : [0], 'Z' : [0]}` as the argument. The loop that we copy in Transformation 1 can easily be found by using this dictionary. In Chapter 6 we describe how to find this dictionary without input from the user. The running time of this transformation is linear in the sum of the lengths of the lists in the dictionary.

In this section we have described a transformation which causes reused data to be moved closer to the processing units *once* and then reused from there. On the CPU, the same is accomplished transparently through the cache design of the CPU, but this has the downside that the data may be evicted and then it has to be reloaded.

Many contemporary compilers perform loop hoisting, including the LLVM compiler infrastructure [17] that the NVIDIA OpenCL compiler is based on. Much, but not all loop-invariant code is performing a computation, which may be hoisted without using extra registers. We only perform it on reads from global memory, in which case it uses several extra registers and the compiler may not be able to analyse if this is profitable.

Original kernel code

```
1 __kernel void Laplace(
2         __global double *X,
3         __global double *Y,
4         __global double *Z,
5         ...) {
6   for (unsigned j = 0; j < storagesize; j++) {
7       for (unsigned d = 0; d < dim; d++) {
8           double X_d = X[d][get_global_id(0)];
9           double Y_d = Y[d][get_global_id(0)];
10          double Z_d = Z[d][get_global_id(0)];
11          ...
12      }
13   ...
14   }
15 }
```

Transformed kernel code

```
1 __kernel void Laplace(
2         __global double *X,
3         __global double *Y,
4         __global double *Z,
5         ...) {
6   double X_reg[dim];
7   double Y_reg[dim];
8   double Z_reg[dim];
9   for (unsigned d = 0; d < dim; d++) {
10      X_reg[d] = X[d][get_global_id(0)];
11      Y_reg[d] = Y[d][get_global_id(0)];
12      Z_reg[d] = Z[d][get_global_id(0)];
13  }
14  for (unsigned j = 0; j < storagesize; j++) {
15      for (unsigned d = 0; d < dim; d++) {
16          double X_d = X_reg[d];
17          double Y_d = Y_reg[d];
18          double Z_d = Z_reg[d];
19          ...
20      }
21  ...
22  }
23 }
```

Listing 5.2: A transformation which enables reuse of data that is read from the global memory inside a loop. In lines 6–8 we allocate the arrays in which we subsequently store the data from the global memory in lines 9–13. Then, in lines 16–18 of the transformed kernel code, we replace the global array references from lines 8–10 of the original kernel code with references to the arrays we allocated in lines 6–8 of the transformed kernel code.

5.4 Placing shared data in local memory

The GPU has a local memory segment which is shared between all threads in a local work group. It is useful when the threads need to communicate

with each other, or if all threads need to read the same data element, that is, if we have some data which is shared between the threads. The local memory is located close to the processing units of the GPU and it is therefore significantly faster to access than the global memory. This transformation is similar to loop tiling [13] which is used to optimize the code for the cache of CPUs.

In the following we present two transformations which exploit the fact that the value needed by one thread was already read from the global memory by another thread. The two transformations follow the same overall scheme: Each thread reads one value from the global memory into local memory, and then each thread reads from the local memory the values that it needs. Hence, we are basically replacing multiple reads from the global memory with one read from the global memory and multiple reads from the local memory.

The first transformation, which we call TileInLocal, takes as argument a dictionary like the one we described in Section 5.3. Hence, it has the same running time as well.

Transformation 2 gives an abstract description of how we rewrite the code. It starts by allocating space in the local memory for an array with a length equal to the local work-group size and of the same type as the array given in the argument. We change the increment of the corresponding loop to the length of the first dimension of the local work group. Then, inside the loop, we create an inner loop and move the body of the outer loop inside the inner loop. The iteration count of the inner loop is equal to the length of the first dimension of the local work group.

At the start of the outer loop, we read data from the global memory into the local arrays, and add a barrier to synchronize the threads. In the inner loop, we replace reads to the global memory with reads to the local memory. After the inner loop we add another barrier. These barriers are important, otherwise a thread may overwrite some data before it was used by the thread that needed it.

Since we have changed the increment of the outer loop and added a new inner loop, some of the subscripts of the other arrays inside the inner loop may be incorrect. We replace any occurence of the outer loop index with the outer loop index plus the inner loop index.

In Listing 5.3 we show an example of this transformation in action. At the top we show a graphical depiction of the execution after the transformation is performed. We perform the transformation on the matrices `A` and `B`, and the local work-group size is 16-by-16. The rows of `A` is divided into groups of size 16. Each of these groups are divided into 16-by-16 sized tiles. The same is done for `B`, but for the columns. Each of these tiles are then loaded to the local memory and a matrix multiplication of these tiles is

Transformation 2 TileInLocal

Input: A list, A, of 2-tuples containing an array reference and a loop.
Input: The length, len, of the first dimension of the local work-group size.

1: **for** $(ref, loop)$ in A
2: Allocate an array of size $len \cdot len$ in local memory.
3: Change increment of *loop* to *len*.
4: Create an inner loop, whose number of iterations is *len* and increment is one, move the body of *loop* into this loop, and place the inner loop inside *loop*. Only create once for every distinct loop.
5: Assign to the local array the data from global memory that will be used inside the inner loop.
6: Add barriers just before and just after the inner loop.
7: Replace *ref* with a reference to the local array inside the inner loop.
8: Rewrite other expressions which uses the loop index of *loop* inside the inner loop.
9: **end for**

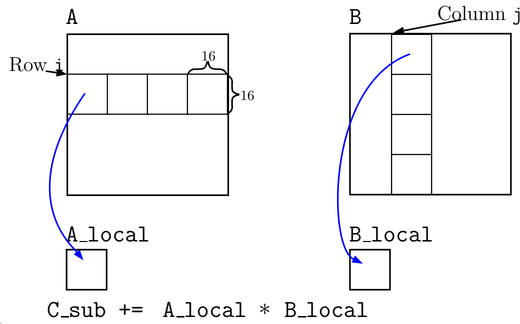
performed and the result is accumulated in `C_sub`. When all tiles have been multiplied we save the `C_sub` tile in the `C` matrix.

Code-wise, we proceed as follow: In lines 4–5 of the transformed kernel code we allocate the corresponding local arrays. In lines 13–15 we add the inner loop, which has 16 iterations, around the body of the outer loop, that is, line 6 of the original kernel code.

Next, we add the assignments in lines 8–11 which reads values from the global memory into the local memory. The right-hand side is a copy of the reference to global memory found in line 6 of the original kernel code. Then we add the local thread identifier to the subscript which contains the loop index `k`. The argument to the local thread-identifier function is the dimension which is not used in the other subscript of the array reference. For example, array `A` has a global thread-identifier function with `1` as the argument in its first subscript. Then we must use the local thread-identifier function with the argument `0` in the second subscript.

The left-hand sides of the assignments are references to the local arrays, where the first subscript is the local thread-identifier function with the same argument which is used in the thread-identifier function of the first subscript of the array reference on the right-hand side of the assignments. The second subscript is handled in an analogous manner.

Now we turn to line 14 in the transformed code. We replace the global array references with the local array references, and the global thread identifiers with the local thread identifiers. In the subscripts, we replace the outer



Original kernel code

```
1  __kernel void MatMul(
2    __global float *A, __global float *B, __global float *C
3    ) {
4    float C_sub = 0;
5    for (unsigned k = 0; k < wA; k++) {
6        C_sub += A[get_global_id(1)][k] * B[k][get_global_id(1)];
7    }
8    C[get_global_id(1)][get_global_id(0)] = C_sub;
9  }
```

Transformed kernel code

```
1  __kernel void MatMul(
2    __global float *A, __global float *B, __global float *C
3    ) {
4    __local float A_local[16][16];
5    __local float B_local[16][16];
6    float C_sub = 0;
7    for (unsigned k = 0; k < wA; k+=16) {
8        A_local[get_local_id(1)][get_local_id(0)] =
9                                  A[get_global_id(1)][k + get_local_id(0)];
10       B_local[get_local_id(1)][get_local_id(0)] =
11                                 B[k + get_local_id(1)][get_global_id(0)];
12       barrier(CLK_LOCAL_MEM_FENCE);
13       for (unsigned kk = 0; kk < 16; kk++) {
14           C_sub += A_local[get_local_id(1)][kk] * B_local[kk][get_local_id(0)];
15       }
16       barrier(CLK_LOCAL_MEM_FENCE);
17   }
18   C[get_global_id(1)][get_global_id(0)] = C_sub;
19 }
```

Listing 5.3: An example of a transformation which replaces reads from the global memory with reads from the local memory. The illustration shows how we divide the matrices intro 16–by–16 sized tiles. The tiles are moved into the local memory and a matrix multiplication of the tiles is performed and the result is accumulated in the `C_sub` tile. This is reflected in the transformation of the kernel code.

loop index `k` with the inner loop index `kk`.

There are several conditions which need to be met before we can

perform this transformation. For example, this transformation only works for two-dimensional local work groups where each dimension has the same length. In Chapter 6 we describe what these conditions are, how to check them, and how to find the arguments for this transformation.

A *stencil computation* updates each point in a grid based on a weighting of its neighbours in the grid. The second type of transformation in this section involves using the local memory to optimize a specific stencil computation, which, in a two-dimensional grid, updates each point using all its four non-diagonal neighbours, or possibly a subset thereof, in the grid.

The transformation, which we call TILEINLOCALSTENCIL, takes as arguments an array and which of the neighbours in the grid are included in the stencil. We first allocate space in the local memory for an array with a length equal to the length of the dimensions of the local work group plus room for potential neighbours. For each neighbour that we include in the first dimension of the grid we add one to the length of the first dimension of the local array and similarly for each neighbour in the second dimension. This extension of the length of dimensions of the local array is sometimes called a *halo* or a *ghost boundary*.

Next, we add one assignment of data from the global memory into the local array per neighbour included in the stencil. The data read is the data needed by the local work group to perform a stencil computation on a tile of the grid. A barrier follows to synchronize the threads. Lastly, we replace array references to the global memory in the stencil with array references to the local memory.

In Listing 5.4 we show the transformation in action on the two-dimensional five-point Jacobi stencil. In lines 4–7 of the transformed kernel code we add a local memory allocation and we assign the local thread identifiers with an offset, to account for the halo, to `li` and `lj` which now function as the local thread identifiers. We use the local work-group size, which is 16-by-16, as the basis for the allocation of the local array, and since the stencil include two neighbours in each dimension, this is extended to a 18-by-18 sized tile.

We add the assignments in lines 7–10 where the right-hand side is a copy of the array references found in lines 6–9 of the original kernel code. The left-hand side is a copy of the right-hand side where we replace the array identifier with the local array identifier and the global thread identifier with the local thread identifiers. These changes are reflected in lines 14–17 of the transformed code.

The transformation works for stencils in a two-dimensional grid which include any of the non-diagonal neighbours. We do not provide any analysis for this transformation and the user therefore has to enable this transformation.

Original kernel code

```
__kernel void Jacobi(
  __global float *B, __global float *X2, __global float *X1
  ) {
  X2[get_global_id(1)][get_global_id(0)] = -0.25 *
    (B[get_global_id(1)][get_global_id(0)]
    - (X1[get_global_id(1) - 1][get_global_id(0)]
                             + X1[get_global_id(1) + 1][get_global_id(0)])
    - (X1[get_global_id(1)][get_global_id(0) - 1]
                             + X1[get_global_id(1)][get_global_id(0) + 1]));
}
```

Transformed kernel code

```
__kernel void Jacobi(
  __global float *B, __global float *X2, __global float *X1
  ) {
  __local float X1_local[18][18];
  unsigned li = get_local_id(1) + 1;
  unsigned lj = get_local_id(0) + 1;
  X1_local[li - 1][lj] = X1[get_global_id(1) - 1][get_global_id(0)];
  X1_local[li + 1][lj] = X1[get_global_id(1) + 1][get_global_id(0)];
  X1_local[li][lj - 1] = X1[get_global_id(1)][get_global_id(0) - 1];
  X1_local[li][lj + 1] = X1[get_global_id(1)][get_global_id(0) + 1];
  barrier(CLK_LOCAL_MEM_FENCE);
  X2[get_global_id(1)][get_global_id(0)] = -0.25 *
    (B[get_global_id(1)][get_global_id(0)]
    - (X1_local[li - 1][lj]
                             + X1_local[li + 1][lj])
    - (X1_local[li][lj - 1]
                             + X1_local[li][lj + 1]));
}
```

Listing 5.4: An example of a transformation which utilizes the local memory in stencil computations. In the transformed code we first allocate a chunk of the local memory with an added halo. Next, we add an offset to the local thread identifiers which is due to the halo. In lines 7–10 we read the data from the global memory into the local memory. We then read the data from that location in lines 14–17.

Loop tiling could have been used to optimize both of the two situations that we have described in this section for the CPU. Unfortunately the transformation is generally not performed by contemporary compilers with default options, except for maybe some of the FORTRAN compilers.

Chapter 6

Pattern-matching rules

In this chapter we present pattern-matching rules that enable the framework to perform some of the transformations in Chapter 5 automatically. We search the kernel code for patterns, and for each one that we find, we check that the conditions of the transformation are met, and perform it if they are.

We do not guarentee that our pattern-matching rules will work for any given program, but we try to make the rules as thorough as possible. We think that this will make the pattern-matching rules work well in practice, which makes it usable for many of the programs that the user may have.

The user is encouraged to skim the resulting code of the transformation step to look for any obvious errors, or simply check that the code gives the expected output for a number of inputs. We note that this is not a nice feature of the framework, in particular for users new to GPU programming.

For each transformation we list the conditions that must be met before we can perform the transformation. In general, the Tranposition, local memory, HoistToRegLoop transformations only work on two-dimensional arrays, and this is something that we check for. The HoistToReg transformation works for an array reference with any number of dimensions, but the subscript of the reference cannot contain any loop indices. From a bird's eye view, the pattern matching works by iterating over the array references and from the code, we create the arguments that the transformations need and then the transformations are performed. The running time is linear in the number of array references found in the code.

In Sections 6.1, 6.2, and 6.3 we present pattern matching for the Transposition, HoistToReg, HoistToRegLoop, and TileInLocal

For 1D:

```
A[get_global_id(0)][d]
```

For 2D:

```
A[get_global_id(0)][get_global_id(1)]
```

Figure 6.1: Examples of patterns where we would perform TRANSPOSITION. `d` is a constant or a loop index.

transformations which we covered in Chapter 5. In Section 6.4 we summarize our thoughts on the pattern matching required in order to generate high-performance GPU code.

6.1 The Transposition transformation

We divide the pattern-matching rule into two cases: one where one loop is parallelized and the other where two loops are parallelized, that is, 1D- and 2D-parallelization. For 1D, the pattern is a two-dimentional array reference having the global thread identifier in the outermost subscript, and something that is not in the innermost subscript, see Fig. 6.1 for an example. For 2D, the pattern is a two-dimentional array reference having the first element of global thread identifier in the outermost subscript and the second element in the innermost subscript. We do not check any conditions, because it is always safe to transpose the data.

6.2 The HoistToReg and HoistToRegLoop transformations

For the purpose of this pattern-matching rule we have created a dictionary which maps each array reference to a list of the loops that it appears in. We iterate this dictionary and, if the particular array reference is in the write-only set, we do nothing. Otherwise, we check the following two conditions: (a) the reference is inside one loop and its subscripts does not contain the loop index, and (b) the reference is inside two loops and the loop index of the outermost loop is not in the subscripts of the reference. If (a) is true, we perform the HOISTTOREG transformation. If (b) is true, we perform the HOISTTOREGLOOP transformation.

We cannot always perform this transformation, because although it is usually profitable to use some of the GPU registers for this transformation,

```
for (unsigned k = 0; k < N; k++) {
      ... = A[get_global_id(1)][k];
      ... = B[k][get_global_id(0)];
}
```

Figure 6.2: Examples of patterns where we would perform TileInLocal.

it would be unwise to use all of them for it. It would be difficult to estimate exactly how many registers can be used, as it would require some kind of cost model.

Instead, we provide a heuristical limit of 20 registers. The number of registers that we need to perform the transformation can be read from the kernel code and if this number is below 20 then we perform the transformation. The number of registers used may depend on parameters that are only known at runtime. If this transformation is applicable, then we generate two version of the kernel code, one with the transformation and one without it. We check at runtime if the version with the transformation uses less than 20 registers and use that version if this was true. Otherwise we use the other version of the kernel code.

6.3 The TileInLocal transformation

The pattern is an array reference with two subscripts where one of them contain a loop index and the other a global thread identifier, see Fig. 6.2 for examples. The conditions, that need to be met, are:

- The grid must have two dimensions.
- The loop index must have a stride of one.
- The length of the range of the loop index must be divisible by the length of the first dimension of the local work-group size.

The last condition is checked at run-time just as we did in the previous subsection.

6.4 Discussion

We have described pattern-matching rules, which we think will work well in practice, for the transformations. This makes the framework more automatic

and user-friendly. The users can turn off any one of the transformations if they discover that it is not profitable. We did not provide the analysis for the TILEINLOCALSTENCIL transformation, because we did not have enough programs to test if the pattern matching would work well in practice.

There are still two things that the users can set manually, if they choose to, in order to get some better performance, namely the local work-group size and if the grid should be one- or two-dimensional. We provide default values which we think will give good performance in most cases. How to choose these values could be automated if we added an automatic tuning module to our framework.

We have identified three important components needed for generating high-performance OpenCL code for GPUs, and in general for other devices as well. First, we analyse the code at compile time. Secondly, we perform some extra analysis at runtime. Finally, tuning of additional parameters is needed.

We can think of two reasons why the C compiler and OpenCL compiler do not perform these analyses, and hence why a framework like ours is needed. First, the two compilers are separated from each other, and many of the transformations require changes to the host and kernel code. Secondly, the OpenCL compiler have detailed information about the architecture that it is compiling for and how many resources are needed by each thread of execution, for example how many registers are used. However, it lacks an important piece of informantion, namely, how many threads will be used to execute the kernel code. These values are known when we initiate the global and local work sizes in the host code, and hence the values are unknown at compile time.

We have been avoiding the last issue by choosing default values which we know by experience will work well for the GPU architecture. For an arbitrary architecture we would not know how to choose the default values and we would have to find them using tuning or choose some values based on heuristics, but this is arguably not an ideal situation.

CHAPTER 7

Use of the framework

Next, we give a brief overview of how to use the framework in practice. Note that what we have built is a prototype of a framework and we have therefore not spent much time on making the framework user friendly.

First, we describe what one needs to do before using the framework. Then, we present how to parse the code and generate the boilerplate code. Finally, we go through the commands that perform the transformations and pattern matching.

The user needs to copy the loop nest of interest to a separate *input file*, and add declarations of the types of the variables that are not declared in the loop nest. If any user-defined functions are used in the loop nest, then these functions need to be copied to a separate file as well. The user must then `include` this file at the top of the input file.

In Listing 7.1 we show an example of what this separate file with the loop nest looks like for the matrix-multiplication program. In the first six lines we declare the types of the variables not declared in the loop. The rest is a copy of the loop nest performing $A * B = C$ from the original program. We save this code in a file called `MatMulFor.cpp`.

We now describe the commands that the user needs to run in order to make use of the framework. We have defined a helper function, `LexAndParse`, which parses the loop nest and creates the AST. The `DataStructures` class runs a set of visitors on the AST, and generates the global data structures that we need for the transformations, pattern matching, and code generation.

```
unsigned wA; unsigned hA; unsigned wB;
float * A; float * B; float * C;
for (unsigned i = 0; i < hA; i++) {
  for (unsigned j = 0; j < wB; j++) {
    float sum = 0;
    for (unsigned k = 0; k < wA; k++) {
      sum += A[i * wA + k] * B[j + k * wB];
    }
    C[wB * i + j] = sum;
  }
}
```

Listing 7.1: Example input which gets parsed in the framework. It contains the types of the variables not declared and a loop nest which calculates $A * B = C$, where each operand is a matrix.

```
def matmul():
    name = 'MatMulFor.cpp'
    ast = LexAndParse(name)
    ds = DataStructures(ast)

    tf = Transformation(ds)
    pat = PatternMatching(ds, tf)

    pat.Transpose()
    pat.DefineArg()
    pat.HoistToReg()
    pat.TileInLocal()

    Generate(pat, ast)
```

Listing 7.2: Example usage of our framework. First, we parse the loop nest found in the `MatMulFor.cpp` file and initiate the needed data structures. Then we initiate the classes for the transformations and the pattern matching. Lastly, we perform the pattern matching and generate the code.

We have created two classes, `Transformation` and `PatternMatching` through which the user can call various functions to perform transformations and pattern matching. Lastly, we have created a helper function, `Generate`, which generates the boilerplate and kernel code. An additional member function of the `Transformation` class, `ParOneLoop()`, is available in order to choose to only parallelize the outermost loop in the loop nest.

In Listing 7.2 we give an example of how to use these functions and classes. Recall that our framework is written in python, so this function is also written in python. The code in Listing 7.1 is located in a file named `MatMulFor.cpp`. We parse this file, perform transformations on the AST, and generate the code which is saved in a file named `boilerplate.cpp`. This file

must be included in the user's original program which allows the user to run the GPU-executable code via a call to `RunMain`, cf. Section 4.6.

Chapter 8

Performance experiments

In this chapter we evaluate our framework based on the achieved performance of six real-world sample programs. The evaluation has three parts: We compare the performance against (a) the theoretical performance of our test hardware, (b) other frameworks with comparative capabilities, and (c) the performance of a price-equivalent CPU.

First, we give a description of the sample programs and how they were benchmarked. Then, we present the hardware platform and the frameworks that we compare against. Finally, we discuss the benchmark results.

8.1 Sample programs

The sample programs used in the benchmarks are:

MatMul: This program multiplies two N-by-N matrices. In theory, this program is compute-bound because it performs $\mathcal{O}(N^3)$ floating-point operations, and reads $\mathcal{O}(N^2)$ floating-point values.

Our framework performs two transformations profitable to matrix-matrix multiplication and we therefore do not expect it to able to compete with hand-optimized versions from software libraries. Nevertheless, we include it as it is a typical subroutine in high-performance software. We benchmark this program with $N = 12\,544$, which was the maximum we could use before we reached the memory limit of the GPU.

Jacobi: This program performs one iteration of the five-point two-dimen-

sional Jacobi stencil computation. This is a so-called smoother used in computational science, for example when solving partial differential equations. We use an N-by-N grid. The maximum problem size we could fit in the memory of the GPU was reached for $N = 16\,384$. This program is memory-bound.

Squared Euclid: This program calculates the squared Euclidean distance between all pairs taken from N objects with 16 dimensions each. The program has many uses, for example as a subroutine in the classical machine-learning algorithm k-nearest neighbour. We benchmarked the program using the input size $N = 16\,384$, which was the maximum problem size we could fit in the memory of the GPU. The program is memory-bound.

NBody: This program is the force calculation in a simulation N bodies, where we used the standard all-to-all $\mathcal{O}(N^2)$ algorithm. This is a classic method for simulating interactions between objects. We use the input size $N = 1\,081\,600$ and the program is compute-bound.

Laplace: This program is a subroutine in a solver of the Black-Scholes partial differential equation with k underlyings [9]. It has a similar structure to a matrix-vector multiplication, but instead of multiplying, we apply a much more compute-heavy operator. Hence, the program is compute-bound. Its running time is $\mathcal{O}(N^2)$ and we use the input size $N = 215\,296$ and $k = 5$.

Gaussian kernels: This program is used in image registration. It calculates the first, second, and third order derivatives of Gaussian kernels [26] in a number of points, N. It has $\mathcal{O}(N^2)$ running time and appears to be compute-bound. We benchmarked this program for input size $N = 4\,608$ which was the maximum we could use.

In general, the algorithms underlying the benchmarked programs are not important, because our framework performs hardware-specific transformations. We executed each program ten times and computed the average running time. We only tested each program with one input size which we think would give the program enough parallelism to saturate the processing units of the GPU. However, for some of our programs we were limited by the memory size of the GPU and we could not be sure of full saturation.

The kernel code was compiled with the NVIDIA OpenCL compiler available in CUDA 5.5 and the Intel OpenCL compiler version 3.0 using the `-cl-relaxed-math` flag. The host code was compiled with gcc 4.7 with the

`-O3` flag. We used the PGI compiler version 14.1 to compile the OpenACC code.

8.2 Systems under investigation

The GPU-side benchmarks were performed on a NVIDIA K20 GPU which has a peak performance of 3.52 TFlop/s and 1.17 TFlop/s in single and double precision, respectively [23]. For the CPU benchmark, we use a machine with two Intel Xeon E5-2670 processors clocked at 2.6 GHz [12]. Together, they have a peak performance of 664 GFlop/s and 332 GFlop/s in single and double precision, respectively.

This CPU has the Advanced Vector Extensions (AVX) [11], which extends the instruction set with so-called single-instruction-multiple-data (SIMD) instructions. These instructions allow one to execute one arithmetical operation, such as addition, multiplication, etc., on multiple independent sets of data, using the same amount of cycles as a normal arithmetical operation. In case of single-precision floating-point numbers, the SIMD instructions can operate on eight sets of data, and for double-precision floating point numbers on four sets. For programs where SIMD instructions are applicable, we can in theory get a speedup of factor four or eight.

When selecting competitors for our framework, we require that they should have approximately the same characteristics as ours. That means that the frameworks should (a) take as input some piece of source code, i.e. a loop nest, from some common programming language, and (b) generate GPU-executable code for that piece of code. Point (a) excludes parallel languages that can generate GPU-executable code. While these languages are useful to people who are writing new parallel programs, we do not think that they will be of much use when parallelizing parts of existing programs, because it is necessary to rewrite the entire program in the parallel language.

We were not able to find many other frameworks, in fact, we only found one, namely the OpenACC API [24]. We use the implementation by PGI [27]. One can parallelize a loop nest with OpenACC by adding so-called *pragmas*, i.e. directives that provide additional information to the compiler, before the `for`-loops in a loop nest. The pragmas contain information about parallelization of the loop nest and data transfer between the host and the GPU. The compiler generates GPU-executable code for the loop nest and incorporates the data-transfer patterns given.

After reading the documentation it is not clear whether the PGI compiler performs hardware-specific optimizations. However, it does provide compiler flags, which we made use of, that enable compilation of the code for

a specific GPU architecture. The PGI compiler generates CUDA code, but it does not allow compilation of the kernel code at run-time. This may prevent some optimizations, because values of command-line arguments cannot be propagated into the kernel code.

We also compared the performance of the GPU execution with that of the CPU execution of the programs. We ran the programs sequentially on the CPU, but they were 2-3 orders of magnitude slower than the GPU execution of the code and we do not think that this comparison would be particularly informative.

We looked for ways to optimize the code for the CPU architecture. For example, OpenMP has the `parallel` pragma to distribute the work across the cores and the `simd` pragma to use the SIMD capabilities of the CPU. However, several of our programs would require the TRANSPOSITION transformation, before the SIMD capabilities could be used. Instead, we expand our framework to generate OpenCL code which utilizes the cores and the SIMD capabilities of the CPU.

8.3 Benchmark results

The code was set up to initialize all data structures pertaining to the GPU execution before the loop nests were executed on the GPU. Data copy from the host to the GPU is included in the running time, while data copy from the GPU to the host is not. For many of the programs, we do not want to copy the data back, but instead continue with another computation on the GPU which uses that data. Take for example the N-body simulation which updates the positions of the bodies in each time step. Instead of copying the data back and forth in each time step, we keep the data on the GPU and copy it back when all time steps have completed.

The code generated by the PGI compiler for **Laplace** and **Gaussian kernels** calculated the wrong values, and we therefore do not compare against these two programs in our performance results. The problem seems to be with privatizing static arrays that are allocated somewhere inside the loop nest, e.g. through statements such as `float temp[3];`. Despite our best efforts to inform the compiler through pragmas that these arrays should be privatized, we did not get it to work. We have already discussed aspects of our framework that may cause it to produce incorrect results. In general we think that there is a certain fragility in these kinds of frameworks, perhaps because there is just so many things to consider and get right for the framework creators.

We compiled the code both with and without the transformations from

	MatMul	**Jacobi**	**Squared Euclid**	**NBody**	**Laplace**	**Gaussian kernels**
GPU OPTIMIZED to GPU BASIC	3.1	1	55.7	3.4	3.6	1.7
GPU BASIC to PGI	0.9	1.9	4.6	2.2	–	–
GPU OPTIMIZED to PGI	2.8	1.9	257.4	7.5	–	–

Table 8.1: Speedup in the execution time of the code generated by the different frameworks.

our framework, denoted GPU OPTIMIZED and GPU BASIC, and with the PGI compiler. In Table 8.1 we give the relative speedup of the running times of the six sample programs when comparing these three different compilations. In the first row we see that our transformations yield high speedups for all but **Jacobi** and **Gaussian kernels**. This is most likely due to the already good cache behaviour of **Jacobi**. We did not expect a high speed up for **Gaussian kernels**, since we could only perform the DEFINEARG transformation. We perform the HOISTTOREGLOOP transformation on **Squared Euclid**, which saves much reusable data in registers, and since it is a memory-bound program, this gives a large speedup.

When we compare GPU BASIC to PGI, we see that GPU BASIC is slightly slower for **MatMul**, but faster for the other programs. This suggests that the PGI compiler is not performing hardware-specific transformations. We already now know that GPU OPTIMIZED is going to be faster than PGI, and row three shows exactly how much faster. The speedups are mixed with some programs having low single-digit speedups, others having high single-digit speedups and some even with triple-digit speedups.

Next, we compare the achieved performance to the theoretical peak performance. We counted the number of floating-point operations for each program and divided that number by the running time. For all but **Laplace**, we use the peak performance for single-precision floating-point numbers, because **Laplace** uses double-precision floating-point numbers.

In Table 8.2 we give the performance in giga floating-point operations per second (GFlop/s) and what percentage of the peak performance was reached. We observe that for **Squared Euclid**, **NBody**, and **Laplace** we achieve 18-25% of peak performance, which we think is reasonably good. There are many reasons why we cannot reach the peak performance for a given program. First, programs may contain integer and comparison operations, which are not floating-point operations. Naturally, these operations

	MatMul	**Jacobi**	**Squared Euclid**	**NBody**	**Laplace**	**Gaussian kernels**
Performance [GFlop/s]	205	4	611	872	245	104
% of peak performance	5.8	0.1	18	25	21	3

Table 8.2: The measured performance of the code generated by our framework for the six programs.

take time to compute, but they are not included in our GFlop/s calculations. Secondly, the peak performance calculation is based on the fact that the GPU can perform an addition and a multiplication in the same instruction, a so-called fused multiply-add (FMA) instruction. However, many programs do not have an instruction mix of sheer FMAs, but instead they have a mix of additions, multiplications, and FMAs. Even when a program has only FMAs, for example the **MatMul**, NVIDIA claims that the program can only reach around 80% of the peak performance [21]. This was for matrix-multiplication with double-precision numbers. It is hard to say if the same level of efficiency can be reached with single-precision numbers.

For **MatMul**, **Jacobi**, and **Gaussian kernels**, we observe that their performance is far from the peak performance. For **MatMul** and **Jacobi**, we think that the programs are memory-bound, despite our transformations which reduce the amount of data transfers from global memory. **Jacobi**, in particular, is very memory-bound and requires non-trivial transformations to increase the performance. Maruyama et al. [19] hand-optimize a three-dimensional seven-point stencil for their GPU which has 25% percent more bandwidth than the ours and they achieve a performance of around 200 GFlop/s.

We are less sure on why **Gaussian kernels** performs poorly. It does have some uncoalesced memory accesses that our optimizations could not resolve, and it has many operations such as square roots, divisions and powers, each of which is counted as one floating-point operation, but they take a lot more cycles to execute than an addition or multiplication. We are not sure if this justifies the measured performance.

We would like to point out that we, in another context, have created a hand-optimized version of **Laplace** which had a performance of 450 GFlop/s, which shows that we are within a factor of two from the hand-optimized code.

We benchmarked two CPU versions of the programs: one with our transformations, and one without, denoted CPU Optimized and CPU Basic respectively. In Table 8.3 we compare CPU Optimized to CPU Basic

	MatMul	**Jacobi**	**Squared Euclid**	**NBody**	**Laplace**	**Gaussian kernels**
CPU Optimized to CPU Basic	6.8	0.7	1.1	1.1	1.1	15.6
GPU Optimized to CPU Optimized	3.3	0.6	36.1	10.9	6.5	1.8

Table 8.3: Speedup in the execution time of the code generated by our framework for the CPU and GPU.

as well as GPU Optimized to CPU Optimized.

For **Jacobi**, **Squared Euclid**, **NBody**, and **Laplace**, we observe that our transformations have no impact. In **MatMul** we are performing the TileInLocal transformation, which reduces the memory transfers from the global memory even on CPUs. We also perform the TileInLocal transformation for **Jacobi**, but here it seems that it is faster to execute the program without the transformation and let the cache do its work. In **Gaussian kernels** the only transformation we perform is to define the kernel function arguments. To our surprise this give a considerable speedup. Either HoistToReg or HoistToRegLoop is performed on **Squared Euclid**, **NBody**, and **Laplace**, and as we expected, this has no effect at all, most likely because it results in register spilling.

When we compare GPU Optimized to GPU Optimized, we see that we are slower for **Jacobi**, which is probably due to better caching behaviour in the CPU code. We get high speedups for **Squared Euclid**, **NBody** and **Laplace** which were the programs that were closest to the peak performance of the GPU.

CHAPTER 9

Concluding remarks

In this final chapter we contemplate the advantages and disadvantages of the approach taken, we give directions for future work, and we summarize one more time the main features of the framework.

9.1 Advantages and Disadvantages

In Chapter 8 we saw that a programmer could make use of our framework to gain significant speedups in the running time of data-parallel programs when comparing to other frameworks and CPU execution. However, throughout this work, we discovered several shortcommings of our approach.

It is difficult to make a framework that take *any* code section as input, for example because there are many special cases of code structuring to consider. Our framework and the PGI compiler have limitations on the code sections that they take as input. Therefore, these frameworks are fragile, because the user do not know if the framework will work for a given code section. We saw that the PGI compiler generates incorrect code when it encounters code structuring that it cannot handle. We think it is better to let the user know that something is wrong through error messages, and abort code generation.

Our framework performs transformations that improve the performance for a single kernel invokation. Often, programs invoke the kernel multiple times with slightly different arguments. Several transformations exists that merge two kernel invokations, for example to enable more data reuse. We could extend the framework to include these types of transformations,

but we think it would require more complex pattern matching that what we presented.

As we added more transformantions to the framework, it got harder to determine if it is profitable to perform a transformation, because we have to consider the order in which the transformations could be performed and if a different subset of transformations would be more profitable. We would like to highlight the difficulty of determining such things and the reason for this seems to be that there, for a given program and hardware, *exists no direct mapping to the set of transformations that one needs to perform to get optimally performing code.* One could reflect on ”whose fault this is”. On the one hand, the hardware manufacturers are adding increasing complexity to the hardware in order to speed up unoptimized programs. On the other hand, software manufacturers should create better programming models for generating code which fits the hardware. In order to generate truly optimal code, it seems that one needs to manufacture the hardware and a programming model in conjunction with each other.

Instead, programmers experiment with different transformations to obtain code that is often suboptimal. In our work, as we implemented few transformations, we performed little experimentation with different sets of transformations. However, we see some problems with this manual approach. First, the programmer is required to know the function of each transformation in order to instruct the framework to perform it. Second, even if the programmer knows each transformation, it may prove difficult to put together ”good” sets of transformations. These two things also puts limitations on the number of persons that can make use of the semi-automatic capabilities of our framework, which is not desirable.

If one is content with suboptimal code, and does not know much about GPU programming, then one could still make use of our framework to generate the GPU related code and even perform a few transformations automatically. We saw a good example of this in Section 8.3 where most of the transformations were performed automatically. In general, we think that there is a nice spread of transformations that can be performed automatically and transformations that can be performed semi-automatically. In this way, our framework is all-round because it accomodates the needs of novices as well as experts in GPU programming.

We also considered how many transformations need to be implemented to obtain high-performance code. In practice, we think that this is in the range 10–20, depending on how complex and how low-level the included transformations are. We see at least one problem with just implementating a fixed set of transformations. Some transformations may not be profitable for a particular architecture, which is something we saw evidence of in our

benchmark results. This means that we need to implement a few architecture-specific transformations for each architecture that we want our framework to generate high-performance code for. Hence, we think that one should make a framework that generates code for only one architecture, but even this may be hard because some GPUs currently receive small architectural changes in each new generation. This can be seen, for example, in the Fermi and Kepler architectures from NVIDIA.

Not only would it be time-consuming for the framework creator to implement all of these transformations, but sometimes it may be hard to know which transformations are actually needed to optimize a program for the complex GPU architecture, a common issue which is almost a research field in its own right.

9.2 Future work

Since the input to our framework is a parallel piece of code, it would be easy to extend the framework to generate code which runs in parallel on multiple GPUs in the same machine and in different machines connected with a network. We would need to create a mechanism that makes a static decomposition of the work load.

We could extend our framework with more transformations, and make it more user friendly. We think that it may be more profitable to design a model which abstracts what our transformations do. If we look aside from the DefineArg and Transposition transformations, then, more abstractly, our transformations are about the reuse of data. With a model of data usage patterns, we could find data that can be reused and reorder the instructions to optimize the ratio between the amount of reused data and the amount of parallel work.

9.3 Summary

We presented a methodology that enables a programmer to take advantage of the computing resources of a GPU. We created a parser that accepts as input a loop nest for which the programmer wants to generate GPU-executable code. We then created an internal representation of the loop nest which made it easy to perform transformations on the code. We made a module which generated the boilerplate and kernel code needed to execute the loop nest on a GPU. Then we implemented several transformations to speed up the running time of the generated code, and pattern matching rules

to perform some of these transformation automatically.

When we benchmarked six sample programs, we observed that significant improvements in time-to-solution were obtained when comparing to other frameworks and to CPU-executable code. For some programs the code ran one order of magnitude faster in both cases. Furthermore, we found that the generated code could attain close to 25% of the peak performance of the GPU for some of the programs. For others, we concluded that additional transformations would be needed to obtain higher performance.

We contributed with our experiences in creating a framework that generates GPU-executable code. We discovered that transformations can be grouped into three sets: transformations that can be performed (a) without checking any conditions, (b) by checking conditions at compile time, and (c) by checking conditions at compile time and at run time. Also, many of the transformations could, with little effort, be performed automatically instead of semi-automatically, which makes our framework usable for both novices and experts.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Boston (1986).

[2] cplusplus.com (Editor), C++ stringstream class, Website accessible at `http://www.cplusplus.com/reference/sstream/stringstream/` (Jan. 2014).

[3] D. Beazley (Editor), Python Lex-Yacc (PLY), Website accessible at `http://www.dabeaz.com/ply/` (Dec. 2013).

[4] X. Dong, G. Cooperman, and J. Apostolakis, Multithreaded Geant4: Semi-automatic transformation into scalable thread-parallel software, *Euro-Par 2010*, *Lecture Notes in Computer Science* **6272**, Springer, Berlin/Heidelberg (2010), 287–303.

[5] E. Bendersky (Editor), Python C parser, Website accessible at `https://github.com/eliben/pycparser` (Dec. 2013).

[6] P. Felber, Semi-automatic parallelization of Java applications, *CoopIS/-DOA/ODBASE*, *Lecture Notes in Computer Science* **2888**, Springer, Berlin/Heidelberg (2003), 1369–1383.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston (1995).

[8] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL — Revised OpenCL 1.2 Edition*, Morgan Kaufmann, Waltham (2013).

[9] A. Heinecke, S. Schraufstetter, and H.-J. Bungartz, A highly parallel Black-Scholes solver based on adaptive sparse grids, *IJCM 2012* **89(9)** (2012), 1212–1238.

[10] E. Holk, W. E. Byrd, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine, Declarative parallel programming for GPUs, *PARCO 2011*, *Advances in Parallel Computing* **22**, IOS Press, Amsterdam (2011), 297–304.

[11] Intel (Editor), Introduction to Intel advanced vector extensions, Website accessible at `http://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions` (Mar. 2014).

[12] Intel (Editor), Xeon processor E5-2670 specification, Website accessible at http://ark.intel.com/products/64595/ (Mar. 2014).

[13] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, San Francisco (2002).

[14] K. Kennedy, K. S. McKinley, and C. W. Tseng, Interactive parallel programming using the ParaScope editor, *IEEE Trans. Parallel Distrib. Syst.* **2**, 3 (1991), 329–341.

[15] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice-Hall, Englewood Cliffs (1988).

[16] Khronos Group (Editor), OpenCL specification v1.2r19, Website accessible at http://www.khronos.org/registry/cl/ (Nov. 2012).

[17] C. Lattner and V. Adve, LLVM: A compilation framework for lifelong program analysis and transformation, *CGO 2004*, IEEE Computer Society, Washington (2004), 75–88.

[18] MAGMA (Editor), Matrix algebra on GPU and multicore architectures, Website accessible at http://icl.cs.utk.edu/magma/ (Nov. 2013).

[19] N. Maruyama and T. Aoki, Optimizing stencil computations for NVIDIA Kepler GPUs, *HiStencils 2014* (2014).

[20] S. S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco (1997).

[21] NVIDIA, NVIDIA's next generation CUDA compute architecture: Kepler GK110 (Whitepaper), Worldwide Web Document (2012). Available at http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[22] NVIDIA, Compute unified device architecture C programming guide, Worldwide Web Document (2013). Available at http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[23] NVIDIA (Editor), Tesla GPUs, Website accessible at http://www.nvidia.com/object/tesla-servers.html (Feb. 2014).

[24] OpenACC, The OpenACC application programming interface, Worldwide Web Document (2011). Available at http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf.

[25] OpenMP (Editor), OpenMP API specification, Website accessible at http://openmp.org/wp/ (Mar. 2014).

[26] S. Sommer, M. Nielsen, S. Darkner, and X. Pennec, Higher order kernels and locally affine lddmm registration, *SIIMS 2013* **6(1)** (2013), 341–367.

[27] The Portland Group, inc. (Editor), PGI accelerator compilers with OpenACC directives, Website accessible at http://www.pgroup.com/resources/accel.htm (Feb. 2014).

[28] H. Vandierendonck, S. Rul, and K. De Bosschere, The paralax infrastructure: Automatic parallelization with a helping hand, *PACT 2010*, ACM, New York (2010), 389–400.

[29] H. P. Zima, H.-J. Bast, and M. Gerndt, SUPERB: A tool for semi-automatic MIMD/SIMD parallelization, *Parallel Comput.* **6**, 1 (1988), 1–18

Appendix A: Paper submitted for PSTI 2014

I submitted the paper below for the Fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014). It is a small workshop held in connection with the 43rd International Conference on Parallel Processing (ICPP 2014), which will be held in Minneapolis, USA, during September 9-12, 2014.

Semi-automatic tool to ease the creation and optimization of GPU programs

Jacob Jepsen
Department of Computer Science
University of Copenhagen
Universitetsparken 5, Copenhagen, Denmark
jepsen@diku.dk

***Abstract*—We present a tool that reduces the development time of GPU-executable code. We implement a catalogue of common optimizations specific to the GPU architecture. Through the tool, the programmer can semi-automatically transform a computationally-intensive code section into GPU-executable form and apply optimizations thereto. Based on experiments, the code generated by the tool can be 3-256X faster than code generated by an OpenACC compiler, 4-37X faster than optimized CPU code, and attain up to 25% of peak performance of the GPU. We found that by using pattern-matching rules, many of the transformations can be performed automatically, which makes the tool usable for both novices and experts in GPU programming.**

I. Introduction

The high performance and low energy consumption of graphical processing units (GPUs) make them a preferred hardware platform for high-performance software. An obstacle for many GPU software manufacturers is the high development time, which stems from complicated programming models. Two similar and well-known models are OpenCL [1] and NVIDIA's CUDA [2], which enable the programmer to express GPU-executable code in a low-level C-like language. Programming in this language is error-prone, and hence, time-consuming.

Two important ways of improving the performance of software is to transform the code to execute in parallel using multiple processing units, and to perform hardware-specific optimizations that, for example, make more efficient use of the memory hierarchy. Parallelization can be done by using the OpenMP API [3] for CPUs and the similar OpenACC API [4] for GPUs, but the APIs promise little or no hardware-specific optimizations. For a given code section, it is difficult to automatically find out which hardware-specific optimizations should be applied, and hence, the programmer must analyse the code to find the optimizations. Consequently, programmers of GPU software need deep knowlegde of the underlying execution model and hardware architecture. We think that many hardware-specific optimizations are so regular that they can be performed automatically. Also, several optimizations are so common that they are used frequently in the optimization process of many different kinds of computationally-intensive code.

A GPU-executable program is often divided into two parts: one part runs on the CPU, the other on the GPU. Two separated compilers are used to generate the executable. It would be desirable to incorporate optimizations into these compilers, but one problem is that many optimizations require changes to both parts of the program. The other is that information needed to perform optimizations may be unknown at compile time. Fortunately, some programming models allow one to compile the GPU part of the program at run-time. One idea is to develop a system that performs as many optimizations as possible to both parts of the code at compile time. If some optimizations require information unknown at compile time, then the system inserts statements that read this information at run-time, and compiles the most optimal GPU code based on that.

We present a tool that takes advantage of the frequency of use and regularity of hardware-specific optimizations by implementing a catalogue of optimizations which the programmer can perform semi-automatically. The tool takes as input a section of computationally-intensive code and generates corresponding OpenCL code. Then, the programmer analyses the code and instructs the tool to perform relevant hardware-specific optimizations. Such optimizations include memory coalescing, use of local memory, and hoisting reused data to registers.

We use the tool to speed up five test programs. We evaluate the performance and compare it to that of OpenACC code and optimized CPU code. The main contributions of this paper are:

- A tool that eases GPU programming through a semi-automatic methodology that reduces the development time of high-performance software.
- Formalization of pattern-matching rules which enable the optimizations to be performed automatically. This makes the tool usable for both novices and experts.
- Experimentation which shows that the generated code can be 3-256X faster than OpenACC code and 4-37X faster than optimized CPU code. Three out of five programs attain close to 25% of the peak performance of the GPU.

The roadmap of the paper is as follows: We review related work in Section II. In Section III we cover relevant parts of the GPU architecture and the OpenCL programming model. In Section IV we explain how to use the tool. We describe source-code transformations and pattern-matching rules in Sections V and VI. In Section VII we evaluate the performance of the generated code. We conclude the work in Section VIII.

II. Related work

The ParaScope Editor [5] analysed code and displayed relevant information to the programmer, in order to help him

decide how to make the code parallel. A catalogue of transformations, that are useful for converting sequential programs to parallel programs, was available. The framework by Lee et al. [6] automatically transforms an OpenMP program into a GPU program. In contrast to the former, our tool only works on programs that are already made parallel. The latter framework does not focus on making hardware-specific optimizations which is one of the the main focuses in this work. To our knowledge, this is the first tool to semi-automatically perform optimizations specific to the GPU hardware.

III. The OpenCL programming model

We view the OpenCL standard [1] as an API that extends an existing programming language, the *host language*, such as C/C++, with functionality to express parallel execution on heterogeneous hardware. The standard defines a language, the *device language*, which is a subset of C99 with extensions to formulate data parallelism and specify address spaces. For more details on OpenCL programming, see [7].

The code written in the host language, the *host code*, executes on the CPU, the *host*, and the code written in the device language is called the *kernel code*. The kernel code may execute on any processor, called the *device*, for which there exists an OpenCL implementation. Such processors include, but not limited to, CPUs from Intel, AMD and IBM, and GPUs from NVIDIA and AMD. The host and kernel code constitute together the *OpenCL program*, which has the desirable property that it is portable across hardware architectures having an OpenCL implementation.

The idea of the OpenCL programming model is to offload a section of computationally-intensive code to a device that provides fast execution. We define three phases that the host code is responsible for: a start-up phase, a kernel code invocation phase, and a shutdown phase. In the start-up phase we do the following:

- Allocate a device.
- Allocate memory on the device to hold the data used by the kernel code.
- Compile the kernel code.
- Set the arguments for the kernel code.

Then the host enters the phase where it invokes the kernel code for execution on the device. This phase also manages data transfer between the host and device. When the device finishes executing, we enter the shutdown phase which handles deallocation of the device memory and device; essentially we free all resources that we reserved in the start-up phase.

We now turn to the execution model used in OpenCL. A *parallel loop* is a loop where each iteration of the loop can be executed independently such that the outcome is the same as that of the sequential execution. We only treat `for` loops.

A *nested loop* is a loop that is placed inside the body of another loop. A nested loop may also contain nested loops which are called doubly-nested loops. In general, a loop can be nested any number of times. A *loop nest* is a loop which contains a set of nested loops. In Fig. 1 we give two examples of how nested loops may occur. We use the loop index to distinguish the loops from each other. In the top example, the

```
for (size_t i = 0; i < NTEST; i++) {
    for (size_t j = 0; j < NTRAIN; j++) {
        float d = 0.0;
        for (size_t k = 0; k < dim; k++) {
            float tmp = test_patterns[i][k]
                      - train_patterns[j][k];
            d += tmp * tmp;
        }
        dist_matrix[j][i] = d;
    }
}

for (size_t i = 0; i < NTEST; i++) {
    for (size_t j = 0; j < NTRAIN; j++) {
        ...
    }
    for (size_t l = 0; l < NTRAIN; j++) {
        for (size_t k = 0; k < dim; k++) {
            ...
        }
    }
}
```

Fig. 1: Two examples of loop nests.

`j`-loop is nested inside the `i`-loop, and the `k`-loop is doubly-nested inside the `j`- and `i`-loops. In the bottom example, the `j`-loop is nested inside the `i`-loop as is the `l`-loop. The `k`-loop is nested inside the `l`- and `i`-loop.

A *perfect loop nest* is a sequence of nested loops where each of the outer loops may only contain a loop in its body, except for the innermost loop which may also contain statements other than loops. A perfect loop nest of *size* m has $m-1$ outer loops plus the innermost loop. The top example in Fig. 1 shows a perfect loop nest of size 2 consisting of the `i`- and `j`-loops, and the bottom example shows one of size 1 consisting of the `i`-loop. In the latter example the outermost and innermost loop of the perfect loop nest is the same.

In OpenCL, a transformation, which assigns threads the task of executing a distinct loop iteration of a parallel loop, takes place. In the following we present OpenCL terminology to be needed later.

A *work item* is the OpenCL way of describing that which is executed by a thread. Work items are organized in a *grid*. The grid has one, two, or three dimensions, where the product of the length of each of the dimensions is the total amount of parallel work. Work items are distinguished by a *global thread identifier*, a tuple with the same number of elements as the grid has dimensions. The global thread identifier corresponds to the Cartesian coordinate of the work item in the grid. Special functions are available in OpenCL for getting the thread identifier.

The parallelization of one, two, or three perfectly nested loops translates directly into using a one-, two- or three-dimensional grid in the execution model. For example, if we want to parallelize two loops, each of size 100, this translates into using a 100-by-100 grid with 10000 work items.

Inside the grid, the work items are further grouped into *local grids* or *local work groups*, which are executed in parallel on the processing units of the GPU. In Fig. 2 we show an

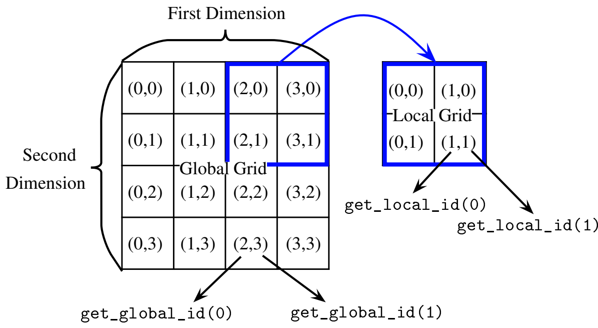


Fig. 2: An example of a grid, the indexes of the work items in the grid and the functions that return the index.

example of a 4-by-4 grid. Each position in the grid is identified by a 2-tuple. The first element of the tuple is used as the thread identifier in the first dimension of the grid and the second element for the second dimension. We have defined a local work group of size 2-by-2.

Elements of the global thread identifier can be accessed using the `get_global_id(dim)` function. The argument `dim`, which can be `0`, `1`, or `2`, specifies which element of the tuple is returned. The function `get_local_id(dim)` does the same but for local thread identifiers. We use the terms *global thread-identifier function* and *local thread-identifier function* when referring to these functions.

The form of parallelism used in the OpenCL programming model is called *data parallelism*, because each thread performs the same operation, but on different elements of data. This has similarities to vector instructions, but vector instructions can only be performed for a limited set of arithmetical instructions, whereas a thread in the OpenCL programming model may execute any C statement defined in the device language specification.

We make use of two important GPU hardware components. The first one is the local memory which is intended to be used for shared data. An array can be placed in local memory by adding the `__local` qualifier. The second is the register file which holds private registers used by the threads. A variable without a memory qualifier is placed in registers by default. Both are located close to the processing units of the GPU and they are much faster to access than global memory. Optimizations that use these components may depend on information available only at run-time. A clever aspect of the OpenCL programming model is the ability to compile kernel code at run-time. This means that we can optimize the kernel at run-time for any set of program parameters before executing it.

IV. Overview of the Tool

The input to the tool is a C/C++ perfect loop nest containing the computation that one wishes to execute on the GPU. To make the parsing task easier, the input must be placed in a separate *input file*. In front of the loop nest, the programmer provides the types of the variables not declared in the loop nest. The tool supports function calls inside the loop nest. The definition of the functions must be placed in a separate file

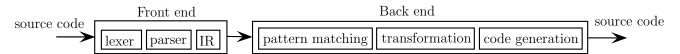


Fig. 3: An overview of the compiler phases implemented in the tool.

Original C code

```
unsigned NTEST; unsigned NTRAIN; unsigned dim;
float *test_patterns; float *train_patterns;
float *dist_matrix;
for (size_t i = 0; i < NTEST; i++) {
    for (size_t j = 0; j < NTRAIN; j++) {
        float d = 0.0;
        for (size_t k = 0; k < dim; k++) {
            float tmp = test_patterns[i][k]
                      - train_patterns[j][k];
            d += tmp * tmp;
        }
        dist_matrix[j][i] = d;
    }
}
```

OpenCL kernel code

```
__kernel void SquaredEuclid(
       __global float *dist_matrix,
       __global float *train_patterns,
       __global float *test_patterns,
       unsigned dim) {
float d = 0.0;
for (unsigned k = 0; k < dim; k++) {
    float tmp = test_patterns[get_global_id(1)][k]
              - train_patterns[get_global_id(0)][k];
    d += tmp * tmp;
}
dist_matrix[get_global_id(0)][get_global_id(1)] = d;
}
```

Fig. 4: An example of how the original loop nest is transformed into kernel code.

which is included at the top of the input file. The tool cannot transform arbitrary loop nests into kernel code, since the loop nest may contain code not conforming to OpenCL C99. The programmer is required to rewrite the code until it conforms. This is probably the most time-consuming task when using the tool.

We implement a lexer and a parser which generate a basic abstract syntax tree (AST) that we use as the internal representation (IR), see Fig. 3 for an overview. From the IR, we set up additional data structures needed in the transformation and pattern-matching phases, which we explain in Sections V and VI.

The tool generates a basic version of the host and kernel code. The host code does all the things explained in Section III. To get the kernel code, we parallelize the outermost loop in a perfect loop nest of size one, and the two outermost loops if the size is two or greater, unless the programmer specifies that only the outermost should be parallelized. The programmer is responsible for checking that data dependencies which prohibit parallelization do not exist.

In Fig. 4 we show such a transformation using a loop nest performing a distance calculation. The original code is a perfect loop nest of size 2 and both of the outermost loops are parallelizable. In the kernel code, we have removed

these two loops and replaced the indices of the two loops with the global thread-identifier function. We replace `i` with `get_global_id(1)` and `j` with `get_global_id(0)`. Note that we display the code with two-dimensional array references for readability and brevity.

A pass over the kernel code checks if any pattern-matching rules apply and, if so, the relevant transformations are performed. A code-generation module converts the AST to source code and saves it in a file that the programmer includes in his original code. The programmer can then execute the generated GPU code with a function call. The tool is written in python.

V. TRANSFORMATIONS

In this section we present source-code transformations which optimize the code for the GPU hardware. A *profitable* transformation speeds up the code by any factor greater than one. In general, the transformations are aimed at any GPU with a corresponding OpenCL implementation, but the transformations are mostly intended for NVIDIA GPUs, specifically for the Kepler GK110 architecture [8]. Due to similarities in the architectures, the transformations should in theory be profitable on a recent AMD GPU architecture, or at least not unprofitable.

In Subsection V-A we explain how to define kernel function arguments as constants, while we in Subsection V-B describe how we coalesce memory access. Then, in Subsections V-C and V-D we present the transformations for placing reusable data in registers and shared data in local memory. The goal of the transformations in the last two subsections is to move data as close as possible to the processing units, where close means with minimum latency.

A. Defining arguments

To *define* a variable means that we propagate the value of the variable into the kernel code at compile time. This is similar to using the `#define` statement in C instead of using a constant global variable. The non-pointer arguments of the kernel function can be defined when we compile the kernel code. This provides additional information to the OpenCL compiler which allows it to perform more optimizations, such as loop unrolling [9].

We call the transformation DEFINEARG. To perform it, we iterate over the kernel function arguments and, for each argument which is not a pointer, we delete it from the list of arguments. Then, we generate a string containing the compiler option that define the removed argument. The strings are concatenated and passed to the OpenCL compiler.

This transformation is similar to constant propagation [10] which can be found in many contemporary compilers. The compilers often perform the transformation offline. Since we are able to compile the kernel code at run-time, we can also define variables that are initialized from command-line arguments or files.

The transformation is not specific to any hardware as such, but we include it as it often gives a noticable speedup. The running time of this transformation is linear in the number of kernel function arguments.

B. Memory coalescing

Kernel codes exhibit several common memory access patterns. In one pattern, each thread accesses a distinct data element from global memory in the same instruction. If these elements are not located consecutively in the global memory, a performance penalty occurs. The reason is that, on the GPU, a memory access is an instruction which accesses several consecutive memory locations. In the case of a memory read, a small chunk of data is read, much like when a CPU reads an element of data, it brings the cache line containing the element to the CPU cache. If we do not use all the elements of a chunk, then we are wasting memory bandwidth. The worst case is when none of the data elements that the threads access are in the same chunk of data. Then each thread will read one chunk and use only one element from this chunk. This is called *uncoalesced memory access*. Not only are we wasting bandwidth, but the memory transfers are performed sequentially resulting in a latency penalty as well.

In order to obtain coalesced memory access we perform the TRANSPOSITION transformation which interchange the two subscripts of a two-dimensional array. The change in the memory access pattern means that we must transpose the data in the array, much like when one transposes a matrix. Related is the transformation of an array of structs (AoS) to a struct of arrays (SoA). When the data is layed out as an AoS, the stride between the same member of two consecutive structs is the size of the struct. To obtain coalesced memory access, we can convert to the SoA layout where the stride is one.

The transformation takes a list of arrays as its argument. For each array in the list we swap the dimensions of the array. We create, in the host code, a new array of the same size as the original. Let the *write-only set* consist of arrays which are only written to. If the array is not in the write-only set, we add code which copies the data from the original array to a new array in the transposed layout. Finally, we replace the old array with the new array in the kernel function argument.

If the array is in the write set then, after the data is transferred back from the GPU, we transpose it in order to give the data the layout that the rest of the user's code expects. The running time of the transformation is linear in the number of arrays given as arguments.

This transformation is not a GPU-specific transformation, because it is often profitable on the CPU architecture, where it rearranges the data to enable vectorization and improve data locality.

C. Placing reusable data in registers

Sometimes, a thread is reading the same data from the global memory in each iteration of a `for` loop. The transformation to be described is about reading the data once, saving it in a variable, and then reading this variable where we read from the global memory before. This transformation has some similarities to loop hoisting, also called loop-invariant code motion [9], which moves loop-invariant code, for example a computation, outside the loop.

The transformation can be performed on the GPU which, unlike the CPU, has a large amount of registers per thread. In

Original kernel code

```
1  __kernel void NBody(
2          __global float *Mas,
3          __global float *Pos,
4          __global float *Forces) {
5    for (unsigned j = 0; j < N; j++) {
6          float a_x = Pos[0][get_global_id(0)];
7          float a_y = Pos[1][get_global_id(0)];
8          float a_m = Mas[get_global_id(0)];
9          ...
10   }
11   ...
12 }
```

Transformed kernel code

```
1  __kernel void NBody(
2          __global float *Mas,
3          __global float *Pos,
4          __global float *Forces) {
5    float Mas0_reg = Mas[get_global_id(0)];
6    float Pos0_reg = Pos[0][get_global_id(0)];
7    float Pos1_reg = Pos[1][get_global_id(0)];
8    for (unsigned j = 0; j < N; j++) {
9          float a_x = Pos0_reg;
10         float a_y = Pos1_reg;
11         float a_m = Mas0_reg;
12         ...
13   }
14   ...
15 }
```

Fig. 5: An example of how the HOISTTOREG transformation works.

Algorithm 1 HOISTTOREGLOOP

Input: A list, R, of 2-tuples containing an array reference and a loop.

1: **for** $(\mathit{ref}, \mathit{loop})$ in R **do**
2: Allocate a temporary array with length equal to the number of iterations of *loop*.
3: Copy *loop* in front of the loop nest it is in.
4: Create assignments from *ref* to the temporary array inside the new loop.
5: Replace *ref* with references to the temporary array inside *loop*.
6: **end for**

---

the GK110 architecture, each thread can use a maximum of 255 registers.

The transformation applies when a thread is reading the same data from the global memory inside one loop, or inside two loops. If the global memory reads are inside one loop, we perform the HOISTTOREG transformation. We proceed by creating a variable outside the loop, assign to the variable the value which is read from the global memory, and then replace the global memory read inside the loop with a read to that variable. In Fig. 5 we see part of kernel code from an N-body simulation. In lines 6–8 of the original kernel code we see that we are reading the same data, the position and mass of a body, from the global memory in each iteration of the `j`-loop. In the transformed kernel code we start by creating an assignment where the left-hand side is a temporary variable and the right-hand side is a copy of the array reference to the global memory location that we read. These assignments are placed at the

Original kernel code

```
1  __kernel void Laplace(
2          __global double *X,
3          __global double *Y,
4          __global double *Z,
5          ...) {
6    for (unsigned j = 0; j < storagesize; j++) {
7        for (unsigned d = 0; d < dim; d++) {
8              double X_d = X[d][get_global_id(0)];
9              double Y_d = Y[d][get_global_id(0)];
10             double Z_d = Z[d][get_global_id(0)];
11             ...
12       }
13   ...
14   }
15 }
```

Transformed kernel code

```
1  __kernel void Laplace(
2          __global double *X,
3          __global double *Y,
4          __global double *Z,
5          ...) {
6    double X_reg[dim];
7    double Y_reg[dim];
8    double Z_reg[dim];
9    for (unsigned d = 0; d < dim; d++) {
10         X_reg[d] = X[d][get_global_id(0)];
11         Y_reg[d] = Y[d][get_global_id(0)];
12         Z_reg[d] = Z[d][get_global_id(0)];
13   }
14   for (unsigned j = 0; j < storagesize; j++) {
15       for (unsigned d = 0; d < dim; d++) {
16             double X_d = X_reg[d];
17             double Y_d = Y_reg[d];
18             double Z_d = Z_reg[d];
19             ...
20       }
21   ...
22   }
23 }
```

Fig. 6: An example of how the HOISTTOREGLOOP transformation works. See text for description.

beginning of the kernel code before any `for` loops. Finally, we replace the right-hand side of the assignments inside the loop with the temporary variables.

If the global memory reads are inside two loops, we perform a slightly different transformation, which we call HOISTTOREGLOOP, see Alg. 1 for a detailed description. First, we allocate a temporary array with a length equal to the number of iterations of the second loop. Then we read data from global memory into this array. The old array reference to global memory inside the two loops are replaced by the corresponding reference to the temporary array. In Fig. 6 we show an example of the HOISTTOREGLOOP transformation. First, in lines 6–8 of the transformed kernel code, we generate allocations of three temporary arrays `X_reg`, `Y_reg`, and `Z_reg`. Then we copy the `d`-loop from line 7 of the original kernel code and place it in front of the other two loops. Inside the new `d`-loop we read the values from the global memory and save them in the temporary arrays. Then, in lines 16–18 of the transformed kernel code we replace the right-hand sides of the assignments with references to the temporary arrays.

This transformation is unprofitable when it causes the

kernel code to use more registers than what is available. This can happen for example if the second loop has a large number of iterations. In Section VI we present the conditions that we check to determine whether we can perform the transformation at all, and if so, if it is profitable. The running time of this transformation is linear in the number of array references given as arguments.

Many contemporary compilers perform loop hoisting, including the LLVM compiler infrastructure [11] that the NVIDIA OpenCL compiler is based on. Much, but not all loop-invariant code perform a computation, which may be hoisted without using extra registers. We only perform it on reads from global memory, in which case it uses several extra registers and the compiler may not be able to decide if it is profitable.

D. Placing shared data in local memory

The GPU has a local memory segment which is shared between all threads in a local work group. The TILEINLOCAL transformation makes use of the local memory to reduce data transfer from global memory when shared data exist. The local memory is located close to the processing units of the GPU and it is therefore significantly faster to access than global memory.

The transformation follows the overall scheme: Each thread reads one value from the global memory into local memory, then reads from the local memory the values that the thread needs. Hence, we are replacing multiple reads from the global memory with one read from the global memory and multiple reads from the local memory. This transformation is similar to loop tiling [10] which optimizes cache behaviour in CPUs.

Alg. 2 gives an abstract description of how we rewrite the code. It starts by allocating an array in local memory. We tile the loop by a factor equal to the length of the first dimension of the local work group. At the start of the outer loop, we read data from the global memory into the local array, and add a barrier to synchronize the threads. In the inner loop, we replace reads to the global memory with reads to the local memory. After the inner loop we add another barrier. These barriers are important, otherwise a thread may overwrite some data before it was used by the thread that needed it.

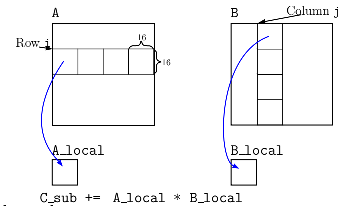
In Fig. 7 we show an example of this transformation in action. At the top we show a graphical depiction of the execution after the transformation is performed. We perform the transformation on the `A` and `B` matrices, and the local work-group size is 16-by-16. The rows of `A` is divided into groups of size 16. Each of the groups are divided into 16-by-16 sized tiles. The same is done for `B`, but for the columns. Each of the tiles are loaded to the local memory, a matrix multiplication of the tiles is performed, and the result is accumulated in `C_sub`. When all tiles have been multiplied, we save the `C_sub` tile in the `C` matrix.

Code-wise, we proceed as follow: In lines 3–4 of the transformed kernel code we allocate the local arrays. In lines 6–17 we tile the `k`-loop by a factor of 16. Next, we add the assignments in lines 7–10 which read values from the global memory into the local memory. Then we add the local thread identifier to the subscript which contains the loop index `k`. In lines 13–14, we replace the global array references with the local array references, and global thread identifiers with local thread identifiers. In the subscripts, we replace the outer loop index `k` with the inner loop index `kk`.

Several conditions need to be met before we can perform this transformation. For example, this transformation only

Algorithm 2 TILEINLOCAL

Input: A list, R, of 2-tuples containing an array reference and a loop.
Input: The length, len, of the first dimension of the local work-group size.
1: **for** $(ref, loop)$ in R **do**
2: Allocate an array of size $len \cdot len$ in local memory.
3: Change increment of $loop$ to len.
4: Create an inner loop, whose number of iterations is len and increment is one, move the body of $loop$ into this loop, and place the inner loop inside $loop$.
5: Place in the local array the data needed inside the inner loop.
6: Add barriers just before and just after the inner loop.
7: Replace ref with references to the local array.
8: Rewrite other expressions which uses the loop index of $loop$.
9: **end for**



Original kernel code

```
1 __kernel void MatMul(
2   __global float *A, __global float *B, __global float *C) {
3   float C_sub = 0;
4   for (unsigned k = 0; k < wA; k++) {
5       C_sub += A[get_global_id(1)][k] * B[k][get_global_id(1)];
6   }
7   C[get_global_id(1)][get_global_id(0)] = C_sub;
8 }
```

Transformed kernel code

```
1 __kernel void MatMul(
2   __global float *A, __global float *B, __global float *C) {
3   __local float A_local[16][16];
4   __local float B_local[16][16];
5   float C_sub = 0;
6   for (unsigned k = 0; k < wA; k+=16) {
7       A_local[get_local_id(1)][get_local_id(0)] =
8                       A[get_global_id(1)][k + get_local_id(0)];
9       B_local[get_local_id(1)][get_local_id(0)] =
10                      B[k + get_local_id(1)][get_global_id(0)];
11      barrier(CLK_LOCAL_MEM_FENCE);
12      for (unsigned kk = 0; kk < 16; kk++) {
13          C_sub += A_local[get_local_id(1)][kk]
14                              * B_local[kk][get_local_id(0)];
15      }
16      barrier(CLK_LOCAL_MEM_FENCE);
17  }
18  C[get_global_id(1)][get_global_id(0)] = C_sub;
19 }
```

Fig. 7: An example of how the TILEINLOCAL transformation works. The illustration shows how we divide the matrices into 16-by-16 sized tiles.

For 1D:

```
A[get_global_id(0)][d]
```

For 2D:

```
A[get_global_id(0)][get_global_id(1)]
```

Fig. 8: Examples of patterns where we would perform TRANSPOSITION. d is a constant or a loop index.

works for two-dimensional local work groups where each dimension has the same length. In Section VI we describe what these conditions are. The running time is linear in the number of array references given as arguments.

VI. PATTERN-MATCHING RULES

In this section we present pattern-matching rules that enables the tool to perform the presented transformations automatically. We iterate over the array references, and for each found pattern, we check a set of conditions. If met, we create the arguments and perform the respective transformation. The conditions are not exhaustive, but we think that they are sufficiently thorough to make the tool usable for many programs. The running time for DEFINEARG is linear in the number of kernel function arguments, and for the other transformations, it is linear in the number of array references in the code.

A. DEFINEARG

We need no pattern-matching rule or conditions to perform this transformation because it can always be performed. For some programs though, a kernel function argument can change between two kernel invocations, which means that we cannot define this argument at compile time. We provide a function which allows the user to exclude an argument when this transformation is performed.

B. TRANSPOSITION

We divide the pattern-matching rule into two cases: one where one loop is parallelized and the other where two loops are parallelized, that is, 1D- and 2D-parallelization. For 1D, the pattern is a two-dimentional array reference having the global thread identifier in the outermost subscript, and something that is not in the innermost subscript, see Fig. 8 for an example. For 2D, the pattern is a two-dimentional array reference having the first element of global thread identifier in the outermost subscript and the second element in the innermost subscript. We do not check any conditions, because it is always safe to transpose the data.

C. HOISTTOREG *and* HOISTTOREGLOOP

For HOISTTOREG, the pattern is an array reference that is inside one or more loops, but contains no loop index, see Fig. 9 for examples. For HOISTTOREGLOOP, the pattern is an array reference that is inside two loops, and the loop index of the outermost loop is not in the subscripts of the reference.

```
for (unsigned k = 0; k < N; k++) {
    ... = A[10];
    ... = B[get_global_id(0)][l];
    for (unsigned g = 0; g < dim; g++) {
        ... = C[get_global_id(1)];
        ... = D[l][10];
    }
}

for (unsigned k = 0; k < N; k++) {
    for (unsigned g = 0; g < dim; g++) {
        ... = A[10][g];
        ... = B[g][get_global_id(0)];
        ... = C[get_global_id(1)][g];
        ... = D[g][l];
    }
}
```

Fig. 9: Examples of patterns. For the top ones, we perform HOISTTOREG, for the bottom ones, HOISTTOREGLOOP.

```
for (unsigned k = 0; k < N; k++) {
    ... = A[get_global_id(1)][k];
    ... = B[k][get_global_id(0)];
}
```

Fig. 10: Examples of patterns where we would perform TILEINLOCAL.

We cannot always perform this transformation, because although it is usually profitable to use some of the GPU registers for this transformation, it is unwise to use all of them. It would be difficult to estimate exactly how many registers can be used. Instead, we make it a condition that the number of used registers may not exceed a heuristical limit of 20. The number of registers needed depends on a set of variables and constants, which we find at compile time. Since the values of the variables may not be known at compile time, we insert `if`-statements into the code which check the condition at run-time. Hence, if a pattern is matched, we generate two versions of the kernel code: one with the appropriate transformation and one without. We decide at run-time which version to use.

D. TILEINLOCAL

The pattern is an array reference with two subscripts where one of them contain a loop index and the other a global thread identifier, see Fig. 10 for examples. The conditions, that need to be met, are:

- The grid must have two dimensions.
- The loop index must have a stride of one.
- The length of the range of the loop index must be divisable by the length of the first dimension of the local work-group size.

The last condition is checked at run-time just as we did in the previous subsection.

VII. Performance experiments

In this section we evaluate our tool by testing the performance of the generated code for five real-world sample programs. The evaluation has three parts: We compare the performance against (a) frameworks with comparative capabilities, (b) the theoretical performance of the test hardware, and (c) the performance on a price-equivalent CPU.

First, we give a description of the sample programs and how they were benchmarked. Then, we present the hardware platforms and the frameworks that we compare against. Finally, we discuss the benchmark results.

A. Sample programs

The sample programs used in the benchmarks are:

MatMul:
This program multiplies two N-by-N matrices. In theory, this program is compute-bound because it performs $\mathcal{O}(N^3)$ floating-point operations, and reads $\mathcal{O}(N^2)$ floating-point values. Our tool performs two transformations profitable to matrix-matrix multiplication and we therefore do not expect it to able to compete with hand-optimized versions from software libraries. Nevertheless, we include it as it is a typical subroutine in high-performance software. We benchmark the program with $N = 12\,544$, which was the maximum we could use before we reached the memory limit of the GPU.

Squared Euclid:
This program calculates the squared Euclidean distance between all pairs of N objects with 16 dimensions each. The program has many uses, for example as a subroutine in the classical machine-learning algorithm k-nearest neighbour. We benchmarked the program using the input size $N = 16\,384$, which was the maximum problem size we could fit in the memory of the GPU. The program is memory-bound.

NBody:
This program is the force calculation in a simulation of N bodies, where we used the standard all-to-all $\mathcal{O}(N^2)$ algorithm. This is a classic method for simulating interactions between objects. We used the input size $N = 1\,081\,600$. This program is compute-bound.

Laplace:
This program is a subroutine in a solver of the Black-Scholes partial differential equation with k underlyings [12]. It has a similar structure to a matrix-vector multiplication, but instead of multiplying, we apply a much more compute-heavy operator. Hence, the program is compute-bound. Its running time is $\mathcal{O}(N^2)$ and we use the input size $N = 215\,296$ and $k = 5$.

Gaussian kernels:
This program is used in image registration. It calculates the first, second, and third order derivatives of Gaussian kernels [13] in a number of points, N. It has $\mathcal{O}(N^2)$ running time and appears to be compute-bound. We benchmarked this program for input size $N = 4\,608$ which was the maximum we could use.

In general, the algorithms underlying the programs are not important, since our tool performs hardware-specific transformations. We executed each program ten times and computed the average running time. We only tested each program with one input size which we think would give the program enough parallelism to saturate the processing units of the GPU. However, for some of the programs we were limited by the memory size of the GPU and we could not be sure of full saturation.

The code was compiled with the NVIDIA OpenCL compiler available in CUDA 5.5 and the Intel OpenCL compiler version 3.0 using the `-cl-relaxed-math` flag. The host code was compiled with gcc 4.7 with the `-O3` flag. We used the PGI compiler version 14.1 to compile the OpenACC code.

B. Systems under investigation

The GPU-side benchmarks were performed on a NVIDIA K20 GPU which has a peak performance of 3.52 TFlop/s and 1.17 TFlop/s in single and double precision, respectively [14]. For the CPU benchmark, we use a machine with two Intel Xeon E5-2670 processors clocked at 2.6 GHz [15]. Together, they have a peak performance of 664 GFlop/s and 332 GFlop/s in single and double precision.

When selecting competitors for our tool, we require that they should have approximately the same characteristics. That means that the frameworks should (a) take as input some piece of source code, that is a loop nest, from a common programming language, and (b) generate GPU-executable code for that piece of code. Point (a) excludes parallel languages that generate GPU-executable code. While these languages are useful to people who are writing new parallel programs, we do not think that they will be of much use when parallelizing parts of existing programs, because it is necessary to rewrite the entire program in the parallel language.

We were not able to find many other frameworks, in fact, we only found one, namely the OpenACC API [4]. We use the implementation by PGI [16]. One can parallelize a loop nest with OpenACC by adding so-called *pragmas*, which are directives that provide additional information to the compiler, before the loop nest. The pragmas contain information about parallelization of the loop nest and data transfer between the host and the GPU. The compiler generates GPU-executable code for the loop nest and incorporates the data-transfer patterns given.

After reading the documentation it is not clear whether the PGI compiler performs hardware-specific optimizations. However, it does provide compiler flags, which we made use of, that enable compilation of the code for a specific GPU architecture. The PGI compiler generates CUDA code, but CUDA does not allow compilation of the kernel code at run-time. This may prevent some optimizations, because values of command-line arguments cannot be propagated into the kernel code.

We also compared the performance of the GPU execution with that of the CPU execution of the programs. We ran the programs sequentially on the CPU, but they were 2-3 orders of magnitude slower than the GPU execution of the code and we do not think that this comparison would be particularly informative.

We looked for ways to optimize the code for the CPU architecture. For example, OpenMP provides the `parallel` pragma to distribute the work across the cores and the `simd`

	MatMul	Squared Euclid	NBody	Laplace	Gaussian kernels
GPU OPT to GPU BAS	3.1	55.7	3.4	3.6	1.7
GPU BAS to PGI	0.9	4.6	2.2	–	–
GPU OPT to PGI	2.8	257.4	7.5	–	–

TABLE I: Speedup in the execution time of the code generated by the different frameworks.

pragma to use the SIMD capabilities of the CPU. However, several of the programs would require the TRANSPOSITION transformation, before the SIMD capabilities could be used efficiently. Instead, we extend our tool to generate OpenCL code which utilizes the cores and the SIMD capabilities of the CPU.

C. Benchmark results

The code was set up to initialize all data structures pertaining to GPU execution before the loop nests were executed on the GPU. Data copy from the host to the GPU is included in the running time, while data copy from the GPU to the host is not. For many of the programs, we do not want to copy the data back, but instead continue with another computation on the GPU which uses that data. Take for example the N-body simulation which updates the positions of the bodies in each time step. Instead of copying the data back and forth in each time step, we want to keep the data on the GPU and copy it back when all time steps are completed.

The code generated by the PGI compiler for **Laplace** and **Gaussian kernels** calculated wrong values, and we therefore do not compare against these two programs. The problem seems to be with privatizing static arrays that are allocated somewhere inside the loop nest, e.g. through statements such as `float temp[3];`. Despite our best efforts to inform the compiler through pragmas that these arrays should be privatized, we did not get it to work. Several things could go wrong with our tool as well which may cause it to produce incorrect results. In general we think that there is a certain fragility in these kinds of frameworks, perhaps because there is just so many things to consider and get right for the framework creators.

We compiled the code both with and without the transformations from our tool, denoted GPU OPT and GPU BAS, and with the PGI compiler. In Table I we give the relative speedup of the running times of the code generated by the three different compilations of the five sample programs. In the first row we see that our transformations yield high speedups for four programs. We perform the HOISTTOREGLOOP transformation on **Squared Euclid**, which saves much reusable data in registers, and since it is a memory-bound program, this gives a large speedup. We did not expect a high speed up for **Gaussian kernels**, since only the DEFINEARG transformation could be performed.

When we compare GPU BAS to PGI, we see that GPU BAS is slightly slower for **MatMul**, but faster for the other programs. This suggests that the PGI compiler is not performing hardware-specific transformations. We already now know that GPU OPT is going to be faster than PGI, and row three shows exactly how much faster. The speedups are mixed, but sizable.

	MatMul	Squared Euclid	NBody	Laplace	Gaussian kernels
Performance [GFlop/s]	205	611	872	245	104
% of peak performance	6	18	25	21	3

TABLE II: The measured performance of the code generated by our tool for the five programs.

Next, we compare the achieved performance to the theoretical peak performance. We counted the number of floating-point operations for each program and divided that number by the running time. In Table II we give the performance in giga floating-point operations per second (GFlop/s) and what percentage of the peak performance was reached. We observe that for **Squared Euclid**, **NBody**, and **Laplace** we achieve 18–25% of peak performance, which we think is reasonably good. There are many reasons why we cannot reach the peak performance for all programs. First, programs may contain integer and comparison operations, which we do not include in the calculation. Second, the theoretical peak performance calculation assumes that all instructions are fused multiply-adds (FMAs), which perform an addition and a multiplication in one instruction. However, many programs do not have an instruction mix of sheer FMAs, but instead of additions, multiplications, and FMAs. Even when a program has only FMAs, such as **MatMul**, NVIDIA claims that the program can reach around 80% of the peak performance [8].

We observe that the performance of **MatMul** and **Gaussian kernels** is far from the peak. For **MatMul**, we think that the program is memory-bound, despite our transformations which reduce the amount of data transfers from global memory. We are less sure on why **Gaussian kernels** performs poorly. It does have some uncoalesced memory accesses that our optimizations could not resolve, and it has many operations such as square roots, divisions and powers, each of which is counted as one floating-point operation, but they take more cycles to execute than an addition or a multiplication.

We would like to point out that we, in another context, have created a hand-optimized version of **Laplace** which had a performance of 450 GFlop/s, which shows that we are within a factor of two of that hand-optimized code.

We benchmarked two CPU versions of the programs: one with transformations, and one without, denoted CPU OPT and CPU BAS. In Table III we compare CPU OPT to CPU BAS as well as GPU OPT to CPU OPT.

For **Squared Euclid**, **NBody**, and **Laplace**, we observe that our transformations have no impact. In these three programs, either HOISTTOREG or HOISTTOREGLOOP was performed, and as we expected, the transformations have no effect, most likely because performing them results in register spilling. In **MatMul** we perform the TILEINLOCAL

	MatMul	Squared Euclid	NBody	Laplace	Gaussian kernels
CPU OPT to CPU BAS	6.8	1.1	1.1	1.1	15.6
GPU OPT to CPU OPT	3.3	36.1	10.9	6.5	1.8

TABLE III: Speedup in the execution time of the code generated by our tool for the CPU and GPU.

transformation, which reduces the memory transfers from the global memory even on CPUs. Surprisingly, **Gaussian kernels** gains a significant speedup even though only the DEFINEARG transformation was performed. When comparing GPU OPT to CPU OPT, we see high speedups for **Squared Euclid**, **NBody**, and **Laplace** which were the programs that were closest to the peak performance of the GPU. None of the hardware-specific transformations were applicable to **Gaussian kernels**, so we did not expect a large speedup.

VIII. CONCLUSION

Our future work consists of extending the tool with (a) a mechanism for decomposing the work load to be executed in parallel on multiple GPUs, (b) a more user-friendly interface, and (c) a model which abstracts what our transformations do. If we look aside from the DEFINEARG and TRANSPOSITION transformations, then, more abstractly, our transformations are about the reuse of data. With a model of data usage patterns, we could find data that can be reused and reorder the instructions to optimize the ratio between the amount of reused data and the amount of parallel work.

We presented a methodology that enables a programmer to take advantage of the computing resources of a GPU. We created a simple mechanism to parse a loop nest for which the programmer wants to generate GPU-executable code. Then we implemented several transformations to speed up the running time of the generated code and pattern-matching rules to perform the transformations automatically. We discovered that transformations can be grouped into three sets: transformations that can be performed (a) without checking any conditions, (b) by checking conditions at compile time, and (c) by checking conditions at compile time and at run-time. Also, many of the transformations could, with little effort, be performed automatically instead of semi-automatically, which makes our tool usable for both novices and experts.

When we benchmarked five sample programs, we observed that significant improvements in time-to-solution were obtained when comparing to code from an OpenACC compiler and optimized CPU code. For some programs the code ran one order of magnitude faster in both cases. Furthermore, we found that, for three programs, the generated code could attain close to 25% of the peak performance of the GPU. For the others, we concluded that further transformations would be needed to obtain higher performance.

ACKNOWLEDGMENT

I am grateful to Jyrki Katajainen for suggesting improvements on the paper, and Stefan Sommer and the Munich Centre of Advanced Computing[1] for providing benchmark platforms.

REFERENCES

[1] (Nov. 2012) OpenCL specification v1.2r19.

[2] NVIDIA. (2013) Compute unified device architecture C programming guide.

[3] (Mar. 2014) OpenMP API specification.

[4] OpenACC. (2011) The OpenACC application programming interface.

[5] K. Kennedy, K. S. McKinley, and C. W. Tseng, "Interactive parallel programming using the ParaScope editor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 3, pp. 329–341, 1991.

[6] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *PPoPP 2009*. New York: ACM, 2009, pp. 101–110.

[7] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL — Revised OpenCL 1.2 Edition*. Waltham: Morgan Kaufmann, 2013.

[8] NVIDIA. (2012) NVIDIA's next generation CUDA compute architecture: Kepler GK110 (Whitepaper).

[9] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco: Morgan Kaufmann, 1997.

[10] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco: Morgan Kaufmann, 2002.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO 2004*. Washington: IEEE Computer Society, 2004, pp. 75–88.

[12] A. Heinecke, S. Schraufstetter, and H.-J. Bungartz, "A highly parallel Black-Scholes solver based on adaptive sparse grids," *IJCM 2012*, vol. 89(9), pp. 1212–1238, 2012.

[13] S. Sommer, M. Nielsen, S. Darkner, and X. Pennec, "Higher order kernels and locally affine LDDMM registration," *SIIMS 2013*, vol. 6(1), pp. 341–367, 2013.

[14] (Feb. 2014) NVIDIA Tesla GPUs.

[15] (Mar. 2014) Intel Xeon processor E5-2670 specification.

[16] (Feb. 2014) PGI accelerator compiler with OpenACC directives.

[1] http://www.mac.tum.de/wiki/index.php/Home