Optimization and Generation of OpenCL Code for Embedded Computing

Luís Alexandre Cubal dos Reis

July 28, 2014
Optimization and Generation of OpenCL Code for Embedded Computing

Luís Alexandre Cubal dos Reis

Mestrado Integrado em Engenharia Informática e Computação

July 28, 2014
Abstract

Multicore heterogeneous architectures are spreading from desktop computers to mobile and embedded systems. In order to take advantage of those multicore architectures, sequential computations might need to be replaced by their parallel counterparts.

Programmers typically develop and validate sequential applications using high-level programming languages such as MATLAB and later manually migrate bottlenecks to lower-level parallel languages such as CUDA and OpenCL. This process is time consuming and error prone. Thus, automatic source-to-source approaches are highly desirable, even when there is a considerable slowdown compared to the manual porting process.

We present a tool able to compile code sections of MATLAB and output equivalent OpenCL. The generated code can be executed on GPU architectures and hardware accelerators. Our tool is based on an existing framework named MATISSE, which generates C code from MATLAB source code. The OpenCL generation relies on the manual insertion of directives to guide the compilation and is capable of generating C wrapper code to interface and synchronize with the OpenCL code. We evaluated our compiler with various benchmarks and domains and we have successfully automatically generated OpenCL code achieving significant speedups (up to 1000 times faster) for matrix multiplication and Monte Carlo financial simulations. Other benchmarks showed smaller improvements, but overall the geometric mean of the cases we tested shows a speedup of up to 13 times faster, depending on the device that is used.
Resumo

Arquiteturas heterogéneas com múltiplos núcleos estão a espalhar-se, deixando de ser domínio exclusivo de computadores e passando a estar disponíveis em telemóveis e sistemas embebidos. Para tomar partido destas arquiteturas, computações sequenciais podem ter de ser substituídas pelo equivalente paralelo.

Os programadores tipicamente desenvolvem e validam aplicações sequenciais usando linguagens de alto nível como o MATLAB e posteriormente convertem manualmente as secções que limitam o desempenho para linguagens de mais baixo nível, como a CUDA e o OpenCL. Este processo é demorado e com alto risco de introduzir erros, pelo que abordagens de código-fonte(source-to-source) automáticas são altamente desejáveis, mesmo quando há uma perda de desempenho substancial comparado com o processo de conversão manual.

Apresentamos uma ferramenta capaz de converter regiões de código MATLAB e exportar código equivalente OpenCL. O código gerado pode ser executado em arquiteturas de GPUs e em aceleradores em hardware. Esta ferramenta é baseada numa infraestrutura já existente chamada MATISSE, que gera código C a partir de código MATLAB. A geração de código OpenCL depende da inserção manual de diretivas para guiar a compilação e é capaz de gerar o código C de comunicação com o código OpenCL. Testamos o nosso compilador com várias benchmarks e domínios e geramos automaticamente e com sucesso código OpenCL com melhorias de desempenho (até 1000 vezes mais rápido) para multiplicações de matrizes e simulações financeiras de Monte Carlo. Outras benchmarks mostraram melhorias menos significativas mas, em média geométrica, os casos testados tornam-se até 13 vezes mais rápidos, dependendo do dispositivo utilizado para executar os cálculos.
Acknowledgments

There is a number of people who contributed in some part to this dissertation. In particular, I would like to thank my thesis supervisor, João Manuel Paiva Cardoso, for proposing the topic, reviewing my work and giving appropriate advice when needed.

I would like to thank João Bispo, who is the author of the original MATISSE C backend and whose assistance allowed me to understand the MATISSE internals much faster, Ricardo Nobre, who helped choose the components for the laboratory computer I used to measure results and directed me to some existent OpenCL programs to test, and the rest of the FEUP SPECS group.

Finally, I would like to thank my family, specifically my parents, godparents and brothers, for their support.
Contents

1 Introduction 1
  1.1 Context and Motivation ................................. 1
  1.2 Target Systems ........................................ 2
  1.3 Project Goals ......................................... 3
  1.4 Contributions ........................................ 4
  1.5 Outline .............................................. 4

2 Related Work 5
  2.1 The MATLAB Programming Language .................. 5
  2.2 The MATISSE Framework ................................ 7
  2.3 GPU Programming Languages .......................... 8
    2.3.1 CUDA ........................................ 8
    2.3.2 OpenCL ....................................... 8
  2.4 MATLAB to CUDA compilers ................................ 13
    2.4.1 MEGHA ....................................... 13
    2.4.2 Code Generation by Chun-Yu Shei et al. .......... 15
  2.5 MATLAB APIs for GPGPU Programming .................. 17
  2.6 Source-to-source MATLAB Compilation for GPGPU API Usage 17
  2.7 Directive-based Programming .......................... 18
    2.7.1 OpenMP ....................................... 18
    2.7.2 OpenACC ....................................... 21
  2.8 Overview ............................................. 22
  2.9 Summary .............................................. 26

3 Compiler Phases 27
  3.1 Directive Identification .............................. 27
  3.2 Expression Decomposer ................................ 30
  3.3 Function Inliner ...................................... 32
  3.4 Directive Cleaner .................................... 32
  3.5 Directive Region Outlining ......................... 34
  3.6 Code Generation ..................................... 34
  3.7 Summary .............................................. 36

4 Compiler Prototype 39
  4.1 Workspace Structure .................................. 39
  4.2 MATISSE Integration .................................. 41
  4.3 MATLAB AST ......................................... 42
  4.4 CIR .................................................. 42
List of Figures

2.1 A MATLAB function that computes a matrix with values on the diagonal. . . . . . . 5
2.2 A MATLAB expression that computes a matrix with 0 in all positions of the diag-
    onal, and 1 in all other positions. . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 The output of the statement in Figure 2.2. . . . . . . . . . . . . . . . . . . . . . 6
2.4 A MATLAB expression that computes a matrix with 0 in all positions of the diag-
    onal, and 1 in all other positions. . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.5 A MATLAB program showing how to use the end keyword. . . . . . . . . . . . . 7
2.6 MATLAB function declaration syntax . . . . . . . . . . . . . . . . . . . . . . . 7
2.7 CUDA program that adds two vectors. . . . . . . . . . . . . . . . . . . . . . . . 9
2.8 OpenCL program that adds two vectors . . . . . . . . . . . . . . . . . . . . . . 10
2.9 C code to call an OpenCL kernel. Error detection and recovery, as well as resource
    cleanup, has been omitted for brevity. . . . . . . . . . . . . . . . . . . . . . . . 11
2.10 (a) MATLAB program without type annotations. (b) MATLAB program with
    added type annotations. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.11 OpenMP program that executes loop in parallel. . . . . . . . . . . . . . . . . . . 19
2.12 OpenMP program with a section that is executed in parallel by each thread. ... 19
2.13 OpenMP program with barriers to enforce synchronization. . . . . . . . . . . . 20
2.14 OpenMP program that computes a value in parallel using a reduction. . . . . . . 20
2.15 OpenACC program that fills an array in parallel. . . . . . . . . . . . . . . . . . . 21
3.1 A MATLAB function that squares the value of each matrix position, with added
    OpenACC-based directives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.2 A MATLAB function with nested loops, with added OpenACC-based directives. 28
3.3 MATLAB comments. Only the last two are recognized as OpenACC-based direc-
    tives. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.4 Simple MATLAB program that can be transformed by the expression decomposer.
    31
3.5 MATLAB program of Figure 3.4 after being decomposed. . . . . . . . . . . . . 31
3.6 MATLAB function that is not yet ready to be inlined because it contains return
    statements. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.7 MATLAB function with the same behavior as the one in Figure 3.6. This version
    does not contain any return statements and therefore can be inlined. . . . . . . 33
3.8 Simple MATLAB function with directives that can be transformed by the expres-
    sion decomposer. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
3.9 Function from Figure 3.8 after the expression decomposer has been executed. . . . 33
3.10 Function from Figure 3.8 after the directive cleaner has been executed. . . . . . 34
3.11 MATLAB function before the outliner has been executed. . . . . . . . . . . . . 35
3.12 MATLAB function of Figure 3.11 after the outliner has been executed. . . . . . 35
3.13 The OpenCL code generated from the MATLAB program from Figure 3.8. . . . . 36
LIST OF FIGURES

3.14 The wrapper code generated from the MATLAB program from Figure 3.8, when profiling mode is enabled. .......................................................... 37
4.1 Diagram of MATISSE projects and project dependencies. ................. 40
4.2 A MATLAB program meant to demonstrate what the AST looks like. ....... 42
4.3 AST Representation of the program in Figure 4.2. ............................... 43
4.4 A CIR instruction list before being converted to C code. .................... 45
4.5 The generated C/OpenCL code from a function with the CIR tree of Figure 4.4. 45
5.1 Idiomatic version of the cpxdotprod3 benchmark. .............................. 48
5.2 cpxdotprod3 benchmark with directives. ......................................... 49
5.3 Relative increase in terms of lines of code of modified MATLAB compared to idiomatic MATLAB. ................................................................. 50
5.4 Speedup of OpenCL code relative to C, including overhead of OpenCL calls and data transfers. ................................................................. 51
5.5 Speedup of OpenCL code relative to C. Times measured by the built-in OpenCL functions and relative only to the kernel execution time. ................. 51
5.6 Kernel execution times for the cpxdotprod3 benchmark for the integrated GPU using the built-in OpenCL profiling feature. ........................... 52
5.7 Speedup of various benchmarks with different local sizes, compared to an automatic local size definition. ................................................. 52
B.1 Source code for the matrix multiplication benchmark. ....................... 67
B.2 Source code for the modified matrix multiplication benchmark. ........... 68
List of Tables

2.1 Comparison of MATLAB APIs for GPU programming . . . . . . . . . . . . . . 24
2.2 Comparison of compilers for GPU execution . . . . . . . . . . . . . . . 24
2.3 Comparison of MATLAB compilers for GPU execution . . . . . . . . . . . . . . 25
LIST OF TABLES

xii
# Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>ARM</td>
<td>Acorn RISC Machine or Advanced RISC Machines</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>AOT</td>
<td>Ahead-of-time</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CIR</td>
<td>C Intermediate Representation</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma-separated Values</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable gate array</td>
</tr>
<tr>
<td>GLSL</td>
<td>OpenGL Shading Language</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU’s Not Unix!</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLLC</td>
<td>High-level Languages Compiler</td>
</tr>
<tr>
<td>HLSL</td>
<td>High-level Shading Language</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-in-time</td>
</tr>
<tr>
<td>MATISSE</td>
<td>A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SysTemS</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Matrix Laboratory</td>
</tr>
<tr>
<td>MEGHA</td>
<td>MATLAB Execution on GPU based Heterogenous Architecture</td>
</tr>
<tr>
<td>NAS</td>
<td>Numerical Aerodynamic Simulation</td>
</tr>
<tr>
<td>OpenCL</td>
<td>Open Computing Language</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multi-Processing</td>
</tr>
<tr>
<td>PGI</td>
<td>Portland Group</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SOC</td>
<td>System-on-a-Chip</td>
</tr>
<tr>
<td>SPeCS</td>
<td>Special-Purpose Computing Systems, languages and tools</td>
</tr>
<tr>
<td>SSA</td>
<td>Static Single Assignment</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

MATLAB is a high-level programming language that is used by over a million engineers and scientists [Mat13b]. It is frequently used for complex and slow numeric computations, and in many situations the programs, or parts of them, are manually mapped to lower-level languages [SYRC11]. This can happen when programmers are mapped to embedded systems or when performance is not acceptable. The process is error prone and time consuming, so automatic approaches are highly desired, even if they are associated with a performance penalty. Embedded systems generally are not amenable to use the runtime MATLAB environment and have constraints which make the translation to highly optimized C code very important. However, high-performance embedded systems include co-processors, such as GPUs and FPGAs, which may be used to accelerate some bottlenecks. OpenCL [Gro13] is a programming language that is emerging as a way to program these processors for achieving parallel computing.

As part of this dissertation we developed a compiler, based on the MATISSE framework, that is able to generate C and OpenCL code from a MATLAB source file annotated with special-purpose directives. This dissertation describes the compiler, the techniques implemented to generate OpenCL code from MATLAB, and evaluates it with it with a set of representative benchmarks.

1.1 Context and Motivation

Currently, most computer applications are developed for CPUs. Historically, the evolution of processor technology has allowed the development of ever more complex programs without having to force users to wait longer for the results (barring some restrictions concerning the temporal complexity of algorithms). However, programs that need to take full advantage of the hardware resources must now be executed in parallel, using the processors’ multiple available cores [Sut04].

Sometimes, even using all the cores of a CPU is not enough. Particularly computation-heavy programs, notably some scientific or financial simulations, take too long to execute on the CPU even when all cores are used. For these programs, co-processors have to be used, such as GPUs. In some cases, GPUs can execute the same programs in less than a tenth of the time they would otherwise take [NV113]. The GPUs that are able execute algorithms for purposes that are not
Introduction

necessarily graphical are known as GPGPUs. Embedded systems typically do not execute this kind of computations, but it is still important to reduce the time it takes to execute tasks and lower the associated energy consumption.

The most commonly used programming languages for GPGPUs are CUDA [NVI14c] and OpenCL. CUDA is a programming language developed by NVIDIA for the purpose of programming NVIDIA GPUs. OpenCL is a standard that is not limited to any single company. OpenCL programs can be executed on any processor that implements the standard, including CPUs, GPUs, FPGAs [Alt13a] and other accelerators [Gro13]. However, not all OpenCL implementations support the same features.

Historically, embedded systems did not include OpenCL capable GPUs, but it is now supported by a variety of mobile GPUs, such as Mali from ARM [ARM14a] and Adreno from Qualcomm [Qua14].

Many programs are developed and tested in the MATLAB programming language. When performance is critical, these programs can be manually converted to more efficient languages, such as C and OpenCL. In practice, this means that such programs are written twice - one prototype in MATLAB and the final version in the target language. This manual conversion requires programmers to know both languages and it still takes time. In addition to the productivity reduction, there is always the possibility of introducing new errors that did not exist in the MATLAB version.

Automated tools capable of converting MATLAB to C (e.g. MATLAB Coder [Mat14b]) and OpenCL are important because they allow programmers to more easily obtain programs with acceptable performance. The approach described in this dissertation requires some manual effort, but this is still more practical than rewriting entire functions in a new language. Fortunately, the model-driven nature of the MATLAB language allows a flexible translation to a desired target language, potentially even allowing multiple versions of the same program to be generated in order to better take advantage of hardware-specific optimizations.

1.2 Target Systems

Historically, most programmers developed applications that only took advantage of the CPU. As single-threaded CPU hardware performance improvements began to stall, hardware manufacturers began to transition to multi-core architectures and software developers needed to follow if they wished to further improve the performance and efficiency of their programs [Sut04].

CPU architectures, such as those for the x86 and ARM instruction sets, feature a number of cores - units that execute operations (instructions) sequentially. Different cores may be executing different parts of a program, or even different programs, at the same time. In order for a single-program to take advantage of multi-cores, multiple threads can be used. Each thread still executes code sequentially, but each thread can be executed in a separate core, thus running in parallel. Each thread can execute independently, but this is not strictly required: threads can share data and can block waiting for other threads to perform certain actions. Additionally, many processor architectures feature SIMD instructions, such as x86’s MMX, SSE and AVX [Int14, subsection 2.2.7]
- SIMD Instructions] and ARM’s NEON [ARM14b]. These instructions operate on larger sets of data (vectors) than conventional instructions (which operate on scalars). In order to properly exploit the performance of these systems, both types of parallelism (multiple cores and SIMD) must be properly taken advantage of. Although programs can be portable (meaning they can be executed on a wide variety of architectures), they must be compiled to the target instruction set in order to be efficiently executed on the processor. Compilation is typically divided in one of two types: AOT (ahead-of-time) compilation or JIT (just-in-time) compilation. When a program is AOT compiled, an executable is generated. Even if the program itself is portable and architecture-agnostic, the executable can only be natively executed by the instruction sets it was compiled for. JIT compilation delays the generation of native code until the code is executed, so a JIT compiled program can be executed on any machine that includes a compatible implementation. Some approaches (such as the one used by the Java Virtual Machine) use a hybrid approach, where the program code is AOT compiled to a platform-independent intermediate representation (the Java bytecode), which is later JIT compiled on the target machine.

CPUs are not sufficiently fast or efficient to execute certain programs. When this happens, one common solution is to solve the problem on a GPU. Several GPU architectures and instruction sets exist, and manufacturers occasionally switch to different architectures. For this reason, executables that depend on specific GPUs are not common. Instead, computers typically feature GPU drivers, software that communicates with GPUs and abstracts the implementation details from the programs, that instead use GPU-agnostic APIs. This means that a pure AOT approach is not feasible for GPUs. Instead, some sort of JIT compiler is used - either compiling the source code directly or an intermediate bytecode.

In addition to CPUs and GPUs, other types of processors can be used. For instance, FPGAs are semiconductor devices that can be programmed after being manufactured. These devices can be modified if necessary to execute different programs when the FPGA is already in the field [Alt14]. Unfortunately, none of the systems we’ve tested included FPGAs.

On desktop systems, CPUs and GPUs are often two separate chips, although integrated GPUs exist and are common. On embedded systems such as mobile phones, CPUs, GPUs and other components are typically integrated on a single SOC (System-on-a-chip).

In this dissertation, we target two systems: a conventional desktop computer with a x86 (64 bits) CPU and a discrete GPU, and an embedded system called ODroid XU+E with a SOC that features an ARM CPU and a PowerVR GPU.

1.3 Project Goals

The goal of this project is to develop a tool capable of generating C and OpenCL code from MATLAB source code, in order to improve the performance of compute-heavy programs running on embedded systems by exploiting a variety of processor types, notably GPGPUs.

Specifically, our goals were to:

- Propose techniques to translate MATLAB code to OpenCL code
Introduction

- Develop a tool capable of generating valid C and OpenCL code
- This tool should support a significant subset of MATLAB code
- The programs generated by this tool should perform better than the equivalent C code whenever they have potential to gain overall speedups on the target GPU
- The effort required to make the programs execute in parallel with this tool should be less than the effort it would take to manually rewrite it in OpenCL

In short, our goal is to map MATLAB programs to the target architecture consisting of a CPU and a GPU and to possibly accelerate their execution by automatically converting them to OpenCL.

It is not our goal to develop a MATLAB parser. Instead, we will reuse an existent one from a framework named MATISSE and develop a new backend for OpenCL generation.

1.4 Contributions

When developing this dissertation, we have developed and produced the following:

- We extended and adapted the directive driven programming model of OpenACC [Ope13] to MATLAB programs
- We have developed a compiler prototype capable of generating valid OpenCL code from MATLAB
- We published a paper and presented a workshop that described the MATISSE framework, including the OpenCL backend [BRC14].
- We have confirmed that semi-automatic OpenCL generation and exploiting GPU parallelism can improve program performance.

1.5 Outline

Aside from this introduction, this dissertation is structured as follows: Chapter 2 describes relevant background and the most relevant related work in the context of this dissertation. Chapter 3 describes the compiler phases developed and included in our compiler prototype. Chapter 4 describes how the prototype works, how we tested it and the most important implementation details. Additionally, it describes the Intermediate Representations used by the compiler. Chapter 5 presents experimental results considering a number of representative MATLAB benchmarks. Finally, Chapter 6 presents concluding remarks and describes possible future work.
Chapter 2

Related Work

This chapter describes the programming languages addressed in this dissertation, as well as other languages, compilers and APIs that attempt to solve problems that are similar to the ones we focused on.

2.1 The MATLAB Programming Language

MATLAB [Mat13b], short for Matrix Laboratory [Mat14c], is a high-level programming language developed by MathWorks [Mat14a] that is used for numeric computation and develop simulations, models and applications. Despite MATLAB itself being proprietary, GNU Octave [Oct14a] is a free software application that is mostly, but not entirely, compatible with MATLAB [Oct14b].

In MATLAB, all variables are multi-dimensional arrays. Even scalars are just arrays of size 1. MATLAB is dynamically typed, so the type and number of dimensions of a variable can change during execution of the program. Additionally, most operations in MATLAB, including operators and functions, can operate on matrices of any size and various types, making MATLAB particularly well suited for array programming. Figure 2.1 shows a MATLAB function that computes a matrix of size $N \times N$ where all elements in the diagonal are scalars of value $X$ and all elements in other positions are scalars of value 0.

```
1 function A = diagX(N, X)
2     A = eye(N) * X;
3 end
```

Figure 2.1: A MATLAB function that computes a matrix with values on the diagonal.

Figure 2.2 shows that even comparison operators, such as == and <, can return matrices. In this case, the returned matrix is displayed in Figure 2.3.
Related Work

```matlab
eye(3) == zeros(3)
```

Figure 2.2: A MATLAB expression that computes a matrix with 0 in all positions of the diagonal, and 1 in all other positions.

```matlab
ans -
1 0 1
2 1 0 1
3 1 1 0
4 1 1 0
5
```

Figure 2.3: The output of the statement in Figure 2.2.

In spite of this, MATLAB still supports `if`, `while` and `for` loops. MATLAB `for` loops are different from the ones seen in languages such as C. Figure 2.4 shows how the MATLAB `eye` function could be implemented for square matrices, albeit in an inefficient manner. In line 3 of this program, `1:N` creates a matrix with a single row with `N` scalars, ranging from 1 to `N` (in sequence). The loop then iterates over each column of the matrix, which is assigned to `i`. Although this syntax is the closest to the most common `for` loops in C, MATLAB does not require the expression of the loop to be a range. Any matrix, including matrices with multiple rows, can be used. Line 4 shows how to set a single position of the matrix. Note that it uses parenthesis to access the position, like function calls do.

```matlab
function y = manual_eye(N)
    y = zeros(N);
    for i = 1:N
        y(i, i) = 1;
    end
end
```

Figure 2.4: A MATLAB expression that computes a matrix with 0 in all positions of the diagonal, and 1 in all other positions.

When accessing matrix positions, it is possible to use indexes and ranges relative to the size of the matrix. Figure 2.5 shows the use of the `end` keyword, which refers to the index of the last valid value of a dimension. Line 2 shows the use of `end`. In this context, `end` is 10 because that is the size of the dimension it is used in. In line 3, `y(1, :)` is equivalent to `y(1, 1:end)`. MATLAB single comments start with `%` and block comments are delimited by `%{` and `%}`, though some restrictions apply. Block comments may be nested.

Figure 2.6 shows the syntax of MATLAB functions. Note that each function may receive zero or more inputs, and may have zero or more outputs. Each MATLAB file may have one or
1  \texttt{y = zeros(10);} \\
2  \texttt{A = y(1, 2:end-1);} \\
3  \texttt{B = y(1, :);} \\

Figure 2.5: A MATLAB program showing how to use the end keyword.

more functions. However, only the first function in each file may be called from other files. The remaining functions in any file can be only be called by functions in the file they were declared.

\begin{verbatim}
function [a, b] = test(c, d) 
  a = c; 
  b = d; 
end
\end{verbatim}

Figure 2.6: MATLAB function declaration syntax

MATLAB also supports \textit{scripts}, files without any user-defined functions. Instead, MATLAB statements are inserted directly on the script.

MATLAB’s greatest strength comes from the extensive and highly optimized set of built-in functions for specific operations (such as efficient matrix multiplication) and its wide range of "Toolboxes", which are libraries that extend MATLAB with domain-specific functions and classes. MATLAB programmers can use these functions to focus on their particular problems instead of having to implement the building blocks of their area first.

At the same time, MATLAB has two significant issues: performance and portability. MATLAB code often suffers from insufficient efficiency. This can be mitigated e.g. by vectorizing code, that is, applying functions and operators to entire arrays rather than iterating every element in a loop [Mat14f] or by converting the code to C using an automatic tool such as MATLAB Coder [Mat14b]. The problem of portability is the fact that MATLAB programs require an appropriate environment to be installed in order to run, and many systems are unsupported (e.g. only x86 processors are supported [Mat14d]). Once again, porting the code to C using MATLAB Coder or another equivalent tool can mitigate this issue.

\section{2.2 The MATISSE Framework}

The MATISSE framework [BPN$^{+}$13] is a MATLAB compiler infrastructure that is able to generate C code. MATISSE is extensible, so new backends can be built to support additional target languages.

MATISSE’s code generator is highly configurable, in order to support not only conventional C compilers but also hardware synthesis tools that have specific limitations such as lack of dynamic memory allocation.
Related Work

In addition to a type inference phase, MATISSE users can manually specify the types of variables. When the MATLAB and the user-defined types not to match, the user-specified type overrides the default one. The ability to override variable types can be useful when targeting architectures with specific limitations. Types are specified using a language called LARA [CCC+12], that also features the ability to match and transform user-defined AST patterns.

2.3 GPU Programming Languages

Traditionally, GPU programming was done by mapping computations to graphics operations or using shader languages [ND10, p. 58] such as HLSL [Mic14], GLSL [KBR14] or Cg [NVI12]. However, these languages were not specifically designed for GPGPU programming. So new languages were developed to simplify this use-case. This section describes the two most important GPGPU programming languages: CUDA [NVI14c] and OpenCL [Gro13].

2.3.1 CUDA

CUDA [NVI14c] is a platform developed by NVIDIA for GPGPU programming. Introduced in 2006 [NVI14a], CUDA extends C, C++ and Fortran with GPGPU-specific extensions. In this section, we describe the C/C++ variant of CUDA.

CUDA programs can be written in a programming language such as C++. Normally, this code is executed in the CPU (the host). However, certain functions can be executed on the GPU (the device). These functions are embedded in the program source code and can be in the same files as the host code. However, by default, functions can not be executed on the GPU. Those that can must be explicitly marked as such.

Figure 2.7 shows a simple CUDA C++ program that computes the sum of two vectors. Line 4 declares a function that can be executed on the GPU. The syntax is very similar to the normal C++ function declaration but modified with the keyword __global__. Line 5 and 6 compute the index of the execution and verify if it is within the vector range. If it is not, then no code is executed, because the current position is out of bounds. If it is, then the operation is computed. Lines 25 to 31 allocate GPU memory and copy the vectors from CPU memory. Line 34 executes the function on the GPU and line 37 moves the result from the GPU to the CPU. The function in Line 4 is executed multiple (N) times, as if there was an implicit for loop calling it repeatedly. However, multiple “iterations” are supported to be executed simultaneously.

2.3.2 OpenCL

OpenCL [Gro13], Open Computing Language, is a royalty-free standard developed and ratified by Khronos and originally proposed by Apple [Gro08]. OpenCL features an API and a C-like language for general-purpose parallel programming across multiple types of processors, including CPUs, GPUs and, more recently, FPGAs [Alt13b]. Figure 2.8 features a simple program written in OpenCL that adds two vectors. The first line declares the function, named float_add, with
Related Work

```c
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void vector_add(int N, float* out, float* a1, float* a2)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < N) {
        out[index] = a1[index] + a2[index];
    }
}

int main(void)
{
    // Initialize vectors
    float a1[128], a2[128];
    float aOut[128];
    int N = 128;
    for (int i = 0; i < N; ++i) {
        a1[i] = i;
        a2[i] = i;
    }
    float* a1_device;
    float* a2_device;
    float* aOut_device;
    cudaMalloc((void**)&a1_device, sizeof(a1));
    cudaMalloc((void**)&a2_device, sizeof(a2));
    cudaMalloc((void**)&aOut_device, sizeof(aOut));

    // Copy data to the GPU device
    cudaMemcpy(a1_device, a1, sizeof(a1), cudaMemcpyHostToDevice);
    cudaMemcpy(a2_device, a2, sizeof(a2), cudaMemcpyHostToDevice);

    // Execute code on the GPU
    vector_add<<<N>>>(N, aOut_device, a1_device, a2_device);

    // Copy data back to the host
    cudaMemcpy(aOut, aOut_device, sizeof(aOut), cudaMemcpyDeviceToHost);
    // aOut now contains the result

    // Free data
    cudaFree(a1_device);
    cudaFree(a2_device);
    cudaFree(aOut_device);
}
```

Figure 2.7: CUDA program that adds two vectors.

three arguments - all global memory pointers. When a kernel is called, the programmer typically indicates the number of times it should be called. This is somewhat similar to calling a function in the body of a `for` loop, however the multiple executions can be performed simultaneously. As most programmers using loops need to know the current iteration, OpenCL also lets programmers distinguish between different executions of the program using the `get_global_id` function, as used in line 2. Finally, the result is computed and stored in line 4. We explain what a kernel, global
Related Work

memory and IDs are in more detail in this section.

```c
kernel void float_add(global float* buffer1, global float* buffer2, global float* result) {
    size_t index = get_global_id(0);
    result[index] = buffer1[index] + buffer2[index];
}
```

Figure 2.8: OpenCL program that adds two vectors

Figure 2.9 shows the C code of a program that uses the OpenCL API. We will refer to this example throughout this section.

Using OpenCL, an application running on the host (the CPU) execute programs on one or more devices (e.g. multi-core CPUs and GPUs). Devices are grouped in platforms, which typically correspond to OpenCL implementations. Multiple platforms can share some or all devices. In the example C code, line 22 is used to get an OpenCL platform, and line 23 is used to get a device from a platform.

Programs include sets of kernels, which are functions that can be executed on OpenCL devices. Additionally, they may also contain auxiliary functions that are used by kernels. Programs can be built from source code files, as shown in line 31. Line 33 obtains a specific kernel from a program.

OpenCL applications submit commands to a device’s command queue. The device may either execute the commands in-order (the commands are executed in the order they were submitted to the command-queue) or out-of-order (each command must wait for a set of events called a wait-list that is explicitly specified by the programmer, but other than that any order is acceptable). The application must specify which mode to use (in-order or out-of-order) whenever it creates a new command queue. Different command queues may have different modes. In this example, line 25 creates a command-queue in in-order mode. Several other API calls use this command queue.

OpenCL performance gains are due to exploited parallelism. OpenCL supports two programming models for parallelism: Task parallelism and Data parallelism.

Data Parallel Programming Model In this model, performance gains are obtained by performing multiple executions of a kernel at the same time. In this case, performance gains come from applying the same operation to large chunks of data simultaneously. OpenCL supports this model using the clEnqueueNDRange function, as seen in Line 38.

Task Parallel Programming Model In this model, the unit of parallelism is the task - a single kernel being executed sequentially - and performance gains come from executing multiple tasks at the same time (e.g. using out-of-order execution mode). OpenCL supports this model using the clEnqueueTask function, which is equivalent to calling clEnqueueNDRange for a single unit
Related Work

```c
#include <CL/cl.h>
#include <string.h>

#define ELEMENTS 1024

int main() {
    cl_platform_id platform;
    cl_device_id device;
    cl_context context;
    cl_command_queue queue;
    float buffer[ELEMENTS];
    cl_mem buffer1_gpu;
    cl_mem buffer2_gpu;
    cl_mem bufferres_gpu;
    const char* program_src = ...
    int src_length = strlen(program_src);
    cl_program program;
    cl_kernel kernel;
    int global_size = ELEMENTS;
    // buffer assignment omitted
    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIds(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
    queue = clCreateCommandQueue(context, device, 0, NULL);
    buffer1_gpu = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                  sizeof(buffer), buffer, NULL);
    ...
    bufferres_gpu = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(buffer), NULL,
                                  NULL);
    program = clCreateProgramWithSource(context, 1, &program_src, &src_length, NULL);
    clBuildProgram(program, 1, &device, "", NULL, NULL);
    kernel = clCreateKernel(program, "float_add", NULL);
    clSetKernelArg(kernel, 0, sizeof(cl_mem), buffer1_gpu);
    ...
    clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, NULL, 0, NULL, NULL);
    clEnqueueReadBuffer(queue, bufferres_gpu, CL_TRUE, 0, sizeof(buffer), buffer, 0,
                        NULL, NULL);
    // Result is in buffer
    return 0;
}
```

Figure 2.9: C code to call an OpenCL kernel. Error detection and recovery, as well as resource cleanup, has been omitted for brevity.
Related Work

of data. Since each clEnqueueTask call executes only a single task, multiple calls in out-of
order mode must be performed to properly exploit task parallelism.

It is possible to combine task and data parallelism, to a certain extent, and execute different
tasks (coarse-grained task parallelism), each applied to a chunk of data (fine-grained data par
allelism), all simultaneously. This model is also supported by calling the clEnqueueNDRange function multiple times, in out-of-order mode. This dissertation is focused on exploiting data parallelism.

Each device has one or more compute units. Each simultaneous kernel execution is a work-
item. Each work-group is executed on a single compute units, and each work-group is a set of
one or more work-items. In other words, it is possible to force multiple work-items to execute
on the same device by specifying that they belong to the same work-group. However, if only a
single work-group is used, then performance may suffer, as only one device compute unit will be
used. The sizes of work-items and work-groups can be specified across multiple dimensions as the range. OpenCL defines two types of ranges: the global_size (the total number of work-items) and the local_size (the number of work-items per work-group). The local size may be NULL, in which case the OpenCL driver automatically decides what value to use. For instance, to execute 16 work-items, a programmer may specify a global size of (16), (2, 8), (4, 4, 1) or others. Each kernel execution has access to the execution ID in a work-group (local ID) or across all work-
groups (global ID) on any given dimension. Additionally, it is possible to obtain the ID of the work-group itself (group ID). In this example, the global size is defined in a variable that is used in line 38. The local size is not explicitly defined, so the driver automatically chooses its value.

Synchronization mechanisms during kernel execution in OpenCL are very limited. It is pos-
sible to force kernels to wait for other kernels in the same work-group to reach a certain point of execution before proceeding using barriers. However, there is no way to synchronize kernels in different work-groups.

OpenCL defines 4 memory regions: private, local, constant and global. Private memory is
accessible only by a given work-item and must not be accessed by other work-items. Each work-
item has its own private memory. Local memory is accessible only by work-items on a given
work-group and must not be accessed by work-items on other work-groups. Constant memory is
initialized by the host and remains constant during the kernel execution. Global memory is acces-
sible to all work-items on all work-groups. It can also be read or modified by the host, through
commands on a command-queue. Line 27 shows how global memory allocation is performed, and
line 40 copies a buffer back to the host.

Work-items reading local or global memory are not guaranteed to get the most recent value
if it has been modified by other work-items, although memory writes are immediately visible on
the work-item it occurred in. It is possible to ensure work-items read the most recent version
of memory using barriers. A local barrier ensures that work-items local memory reads provide
the correct value if it was not modified since the barrier. Global barriers do the same for global
memory. However, this only applies to work-items in the same work-group. There is no way
Related Work

to force work-items to read the most recent value if it was written by a work-item in a different work-group.

OpenCL includes a built-in profiler that can be optionally enabled by the programmer when any command queue is created. Any OpenCL command that is associated with an event can be profiled so that its impact on the application performance can be measured. Four different times can be measured using the built-in OpenCL mechanism:

- **QUEUED** - When the host created the command to be profiled.
- **SUBMIT** - When the host submitted the command to the device.
- **START** - When the device started execution of the command.
- **END** - When the device finished execution of the command.

2.4 MATLAB to CUDA compilers

Some authors have successfully developed compilers of MATLAB to CUDA code. This section describes the approaches we are aware of.

2.4.1 MEGHA

MEGHA [PAG11][PG12] is a compiler developed by members of the Indian Institute of Science (Bangalore, India) and introduced in 2011. This tool processes MATLAB scripts using GNU Octave and generates CUDA and C++ code.

This compiler uses heuristics to decide which portions of the code should be executed on the CPU and which should be offloaded to the GPU. In order to measure the speedup of the resulting executables, the authors used benchmarks from a previous project called Mat2C [JB07] and a new one developed in-house. For the tested benchmarks, the authors obtained a geometric mean speedup of 12.2 times over MATLAB. These results were obtained on a computer with an Intel Xeon 2.83 GHz processor and a GeForce 8800 GTS 512, with single-precision floating point.

MEGHA uses the following compiler structure:

**Code Simplification**  Converts complex expressions into simpler ones, introducing temporary variables if needed. This phase, however, does not handle expressions that refer to the end of an array, such as \(a(:)\) and \(a(2:end)\).

**Semantics Preserving Transformations**  The purpose of this phase is to simplify subsequent phases. This phase converts matrix accesses that use `end` to the equivalent code without accesses. For instance, \(a(:)\) becomes \(a(1:length(a))\). Additionally, it simplifies loops so that the iterated variable is never modified.
Related Work

Static Single Assignment Construction In this phase, the compiler converts the code to a SSA [Mue97] representation. The insertion of $\phi$ nodes is optimized to ensure that only live variables are kept. The authors consider this optimization important because variables may be arrays.

Type and Shape Inference With the code in SSA format, the compiler then attempts to discover the types of each variable, including the intrinsic type (boolean, integer or real), the shape (scalar or matrix) and the size (in each dimension). The compiler uses constants and the semantics of built-in functions and operators to discover this information, although user annotations in the input MATLAB are also possible.

Kernel Identification The compiler identifies sets of IR statements that can be executed on the GPU in an efficient manner. In order to do this, it executes the following steps:

1. Identification of data parallel IR statements.
2. Identification of sets of statements that can be grouped in the same kernels.
3. Scalarization and Index Array elimination to remove unnecessary array accesses (accesses that become unnecessary due to the way they were grouped).
4. Replacement of data parallel statements by the equivalent parallel loops.

Parallel Loop Reordering The order the kernels are executed can have a significant impact on performance, e.g. impact memory access locality and memory coalescing. This phase attempts to compute both the best approach for the CPU and the best approach for the GPU, since the decision of where to execute the code has not been done yet. The decision of which loop order to use is based on heuristics.

Mapping and Scheduling At this point, the kernels have been identified but not the processor they are executed in. The compiler represents each kernel as a node in a graph, and the data dependencies between kernels are the edges of that graph. The compiler then uses a variant of list scheduling to set what processor each kernel executes in and attempts to minimize the total time required to compute the values and transfer data between the processors. According to the authors, this problem is NP-hard, so heuristics were used.

Global Data Transfer Insertion At this point, the processor the kernels are executed in is already selected. This phase introduces the required data transfers.

Code Generation Finally, the compiler generates C++ code for the parts of the program mapped to the CPU and CUDA code for the GPU. The code picks one of the two versions of loops generated by the Parallel Loop Reordering phase depending on the processor the code is going to execute in.
MEGHA has various limitations. Notably, it only accepts a small subset of MATLAB. The following is a list of the most important limitations:

- User-defined functions are not supported (only MATLAB scripts are), unless the compiler frontend is extended.
- Anonymous functions and function handles are not supported.
- Only a few data types are supported. Notably, strings and arrays with more than 3 dimensions are not supported.
- The type inference mechanism is limited. MATLAB is dynamically typed, so there is no way to accurately determine the types of variables at compile time in all cases.

Despite these limitations, the supported subset does not need any changes or annotations of any sort for the code to be compiled by MEGHA. MEGHA supports user annotations to indicate the type of variables, which can be used when the type inference mechanism is unable to determine them automatically.

Since 2012, the MEGHA compiler optimizes memory accesses, further improving program performance by 3.1 times (geometric mean) when compared to the 2011 version [PG12].

2.4.2 Code Generation by Chun-Yu Shei et al.

Another MATLAB/Octave to CUDA compiler was presented by Chun-Yu Shei et al. [SYRC11] as part of the HLLC/ParaM project [Cha13].

This compiler generates both C++ and CUDA code. However, unlike MEGHA, portions of the resulting code remain in MATLAB.

The authors used benchmarks from various domains, one of which being the NAS benchmarks [BBB+91]. The results were obtained using MATLAB 2010b, GCC 4.5.1 and NVIDIA CUDA C 3.1, v0.2.1221, on a Gentoo Linux 2.6.32 64-bits system, running on an Intel Xeon X5365 processor (8 cores, 3 GHz, 8MB L2 cache) with 8 GB RAM DDR2 and a NVIDIA Tesla C1060 GPU with 4 GB of RAM. Each test was executed and measured 5 times. The obtained speedups range from 1.5 to 17 times.

This compiler uses the following phases:

**Front end** First, the MATLAB code is parsed using the Octave parsing library. Once the Octave tree is obtained, it is converted to the format used by the compiler, using an internal tool called RubyWrite.

**Preliminary Optimizations** The compiler flattens the AST, introducing new temporary variables in the process as necessary. This is similar to the Code Simplification phase of the MEGHA compiler. Then, the type inference analysis is executed.
Related Work

Type inference introduces new variables indicating the type of each variable. For instance, for every variable \( x \) a new variable called \( \text{iType}_x \) is created. Many of these variables are eliminated by a posterior partial evaluation and dead code elimination phases. Figure 2.10 (a) shows a MATLAB script without these additional annotations. Figure 2.10 (b) shows the same program after annotations have been introduced but before the dead code elimination phase has been executed. In this case, a new function named \( \text{IXF\_sum} \) has been introduced to compute the type of the addition result. During the static evaluation analysis, the IR is in the SSA form.

```
1 a = 1;
2 b = 2;
3 c = a + b;
```

(a)

```
1 iType_a = 'i';
2 a = 1;
3 iType_b = 'i';
4 b = 2;
5 iType_c = IXF_sum(iType_a, iType_b);
6 c = a + b;
```

(b)

Figure 2.10: (a) MATLAB program without type annotations. (b) MATLAB program with added type annotations.

**Advanced optimizations**  This compiler performs *code sequence optimization* and *type-based specialization*.

**Backend**  The compiler selects which functions from libraries to use and then generates the CUDA code.

The authors took care to ensure that the generated code can be vectorized by SSE-aware C++ compilers.

The main limitation of this approach compared to MEGHA is that the generated outputs still require a MATLAB environment to run, making it unsuitable for embedded systems.
Related Work

2.5 MATLAB APIs for GPGPU Programming

There are several APIs, made by different authors, that allow MATLAB code to execute operations on the GPU.

MathWorks itself provides the Parallel Computing Toolbox [Mat13c], which provides a set of functions that are executed on the GPU, as well as a way to execute user-defined CUDA code from MATLAB. The functions that execute on the GPU coexist, rather than replace, the ones that execute on the CPU. For instance, there is a function named `gpuArray.ones` which is roughly equivalent to the native `ones` [Mat13a]. The execution of CUDA code, too, is performed using MATLAB functions, such as `parallel.gpu.CUDAKernel` [Mat13d].

GPUmat [gpy14] is an open-source library [Gro12, p. 13] for MATLAB using an approach similar to the one of Parallel Computing Toolbox. GPUmat too, is based on CUDA and defines a set of functions, variables and classes that can be used by MATLAB code in order to execute operations on the GPU [Gro12, p. 17].

AccelerEyes, now named ArrayFire [Arr14], used to have a product named Jacket, which added functions and commands to MATLAB, such as `gzeros` and `gfor` [Wik13]. This tool supported both CUDA [Mel12b] and OpenCL [Sco11]. However, in 2012, Jacket was discontinued and AccelerEyes encouraged users to instead use the Parallel Computing Toolbox and MATLAB Distributed Computing Server, as the result of a partnership with MathWorks [Mel12a].

2.6 Source-to-source MATLAB Compilation for GPGPU API Usage

Chun-Yu Shei et al. developed an additional approach [SRC11] that consists of compiling MATLAB code to MATLAB code that uses GPGPU APIs. The authors used GPUmat [gpy14], although they note that their work could easily be adjusted to use any similar library. This work is also part of the HLLC/ParaM project [Cha13]. Programs do not need any changes to be compiled with this tool, although optional type annotations are supported.

In order to measure the performance of the generated code, the authors executed over 50 benchmarks, from several sources. Of those, about 12 had speedups with the developed compiler. The tests were ran on a computer with an Intel Xeon X5365, with 8GB of DDR2 RAM, NVIDIA Tesla C1060 (4GB RAM), MATLAB 2010b and GPUmat 0.25. The speedups depend on the benchmark and the input data size but in the best case scenario, they obtained a speedup of 7 times faster than the original version.

The compiler uses the following phases:

Type inference Chun-Yu Shei et al.’s approach uses static analysis to infer the types of variables. This phase is similar to the one described in Section 2.4.2. Should the static analysis be insufficient for full type inference, the compiler introduces code to perform the missing checks at runtime.
Related Work

**Identification of sections of the program that may be parallel** The compiler uses a white-list approach to this. Every time a MATLAB function is used, the compiler checks if it is on a list of functions that can be offloaded to the GPU. If part of the types are not fully known at compile-time, then portions of this phase are also performed at runtime.

**Cost estimation** For every function that may be executed on the GPU, the compiler estimates how long that would take. For every function, there is an associated cost heuristic function that depends on the input size. These functions were computed empirically.

**Computation partitioning** The operations to compute are distributed among the two types of processors. Computing the optimal partitioning is a NP-complete problem, so the authors used an approximation.

**Computation reordering** Once it is known which processors will be executing each operation, the compiler reorders operations so that, whenever possible, both processors will be performing tasks simultaneously.

**Code generation** The MATLAB code with the additional operations and checks is added. Finally, the compiler performs a partial evaluation and dead code elimination phase.

### 2.7 Directive-based Programming

For C/C++ parallel programming, there are a number of approaches that rely on directives, that is, code that is manually added by the programmer to convey information beyond that which is supported by the official syntax. The two most relevant approaches, OpenMP [Boa13] and OpenACC [Ope13], are described in this section.

#### 2.7.1 OpenMP

OpenMP [Boa13] is a standard for C, C++ and Fortran that extends the languages with functions, environment variables and directives to write parallel programs with shared memory [Boa13, p. 1]. In C and C++, these directives use the built-in `#pragma` command, while Fortran programs specify directives using comments with a specific format.

Figure 2.11 shows an OpenMP program that executes a loop in parallel. The iterations of the loop are distributed across multiple threads, which execute simultaneously. In line 7, the `omp_get_thread_num` function obtains the index of the current thread. Note that the order the lines are printed is undefined and, in fact, lines may appear mixed.

In OpenMP programs, there is a **master thread**, which is the thread that normally executes code sequentially, and **worker threads**, which execute code only when instructed to do so. When a `parallel` directive is added, the parallel code region is executed by the master and the worker threads, simultaneously.
Related Work

Among the features of OpenMP, we select the following as the most relevant to our work:

- **parallel for**: Execute a for loop in parallel (the iterations are distributed by multiple threads).
- **parallel**: Execute a code block in parallel, so that each thread executes the block. Figure 2.12 shows a program that uses this directive. Each thread will execute the printf function once.

- **num_threads**: Indicates how many threads should be used to run a parallel block.
- **master**: Indicates that a section of code should be executed only by the master thread.
- **barrier**: Indicates that the execution of the code should only continue once all threads reach that point of the code. The for directive has an implicit barrier at the end by default unless the nowait parameter is specified. Figure 2.13 shows a program with barriers. When the master section is executed, the buffer has already been filled and it is zeroed only after line 18 was executed.
- **private**: Indicates that each thread should have its own private copy of a variable (as opposed to being shared by all threads).
Related Work

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int buffer[10];

    #pragma omp parallel num_threads(10)
    {
        int thread_num = omp_get_thread_num();
        buffer[thread_num] = thread_num;

        #pragma omp barrier
        #pragma omp master
        {
            int total = 0;
            for (int i = 0; i < 10; ++i) {
                total += buffer[i];
            }
            printf("0+1+...+9 = %d\n", total);
        }
        #pragma omp barrier
        buffer[thread_num] = 0;
    }
    return 0;
}
```

Figure 2.13: OpenMP program with barriers to enforce synchronization.

- **reduction**: Indicates that a variable is partially computed by multiple threads and the end-result should be the combination of those partial results. Figure 2.14 shows a program with a reduction.

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int result = 0;
    // result = 0+1+2+...+999
    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < 1000; ++i) {
        result += i;
    }
    printf("Result is %d\n", result);
    return 0;
}
```

Figure 2.14: OpenMP program that computes a value in parallel using a reduction.

- **num_threads** Indicates how many threads should be used.
Related Work

- **if**: Indicates that a loop should only be executed in parallel if a condition is true (not zero, in C/C++). If not, the loop is executed sequentially.

There have been projects that compile C with OpenMP directives and generate code to be executed on GPUs. For instance, the Cetus project compiles C/OpenMP code to CUDA [LME09].

### 2.7.2 OpenACC

OpenACC [Ope13] is a standard for parallel programming, similar to OpenMP, but focused on heterogeneous systems. OpenACC is based on directives for C, C++ and Fortran.

Since many of the devices OpenACC intends to support do not have shared memory, the OpenACC specification includes parameters to define which memory regions should be copied from the host to the device before execution can begin (copyin), from the device to the host after execution ends (copyout) or both (copy). Figure 2.15 shows an OpenACC program that fills a buffer with square numbers and copies it to the host.

```c
int main() {
  double results[1000];

  #pragma acc kernels copyout(results)
  for (int i = 0; i < 1000; ++i) {
    results[i] = i * i;
  }

  // Use results array here
  return 0;
}
```

Figure 2.15: OpenACC program that fills an array in parallel.

The supported outputs vary depending on the implementation. Regardless, both the CAPS OpenACC Compiler [CAP13] and accULL [RLRFdS12] can generate both CUDA and OpenCL code. However, CAPS enterprise has been experiencing financial difficulties for over a year and announced that it will close down and stop all activities on June 27, 2014, as part of its bankruptcy process [CAP14]. In addition to these, the PGI Accelerator [PGI14] can generate CUDA or OpenCL code from C or Fortran programs annotated with directives [Wol10]. Originally, this compiler used custom directives, but it now supports OpenACC as well.

Among the directives and parameters OpenACC supports, the following are the most relevant to our work:

- **parallel directive**: Indicates that a block should be executed in parallel by one or more threads in parallel.

- **kernels directive**: Region of code to be compiled into one or more kernels.
Related Work

- **if** parameter: Indicates that the code should only be run on the device if the condition is true, and run in the host otherwise.

- **num_gangs** parameter: The number of *gangs* to use (equivalent to OpenCL’s work-groups).

- **num_workers** parameter: The number of *workers* to use (equivalent to OpenCL’s work-items).

- **copyin** parameter: The indicated variables’ values must be copied to the device’s memory (if the memory is not shared) before the kernel can be executed. The data does not need to be copied back to the host after the kernel is executed.

- **copyout** parameter: The indicated variables’ values must be copied to the host’s memory (if the memory is not shared) after the kernel has finished executing. The data does not need to be copied to the device before the kernel can be executed.

- **copy** parameter: The variables’ values must be copied to the device’s memory, and then back to the host’s, if it is not shared.

- **reduction** parameter: The listed scalar variables and partially computed by each worker, and must be combined using the specified operator.

2.8 Overview

There are various approaches to execute code in parallel. Overall, we can separate the approaches we have presented in this section in three groups:

- Approaches that require a rewrite of the program in a different programming language, such as OpenCL.

- Approaches that require some changes in order to indicate which portions of the program should be offloaded, such as APIs and directive approaches.

- Approaches that attempt to offload certain sections of the code automatically, using heuristics to decide where to execute the code.

Some of these approaches (including the GPGPU languages, the MATLAB GPU APIs, the MATLAB to GPU compilers and OpenACC) were designed specifically with GPGPU programming in mind, while others OpenMP was primarily designed for CPUs and later adapted to GPUs.

Table 2.1 compares multiple MATLAB APIs for parallel computing. There is an official, proprietary API by MathWorks named the Parallel Computing Toolbox, which supports CUDA. Additionally, the GPUmat is an open-source implementation that also uses CUDA for the backend, although it hasn’t been updated since 2012. The Jacket API supported both CUDA and OpenCL, but was discontinued. We are not aware of any other APIs that use OpenCL.
Related Work

Table 2.2 compares compilers that generate code to perform computations in parallel. We have focused on compilers for MATLAB and C with directives. All MATLAB compilers described in this table are based on Octave. MEGHA supports only a subset and generates C++ and CUDA code, the HLLC/ParaM project supports, at least in theory, all MATLAB code but accelerates only a subset, either by generating C++ and CUDA code or adding GPumat calls for the compiled subset. Overall, all MATLAB compilers described here either compile to CUDA or GPumat (which indirectly uses CUDA). We are not aware of any way to generate OpenCL code from MATLAB files. For C compilation with directives, there are approaches (e.g. Cetus) that use OpenMP and others (e.g. accULL) that uses OpenACC. The accULL project supports both CUDA and OpenCL, whereas Cetus supports only CUDA.

Finally, Table 2.3 focuses on the differences between MATLAB compilers for parallel execution.
Table 2.1: Comparison of MATLAB APIs for GPU programming

<table>
<thead>
<tr>
<th>Name</th>
<th>Owner</th>
<th>Base API used</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Computing Toolbox [Mat13c]</td>
<td>MathWorks</td>
<td>CUDA</td>
<td>Supported, official</td>
</tr>
<tr>
<td>Jacket [Mel12a]</td>
<td>AccelerEyes</td>
<td>CUDA or OpenCL</td>
<td>Discontinued</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison of compilers for GPU execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Institution</th>
<th>Source</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLLC / ParaM [SRC11]</td>
<td>2011</td>
<td>Indiana University</td>
<td>MATLAB/Octave</td>
<td>MATLAB with GPUmat API calls</td>
</tr>
<tr>
<td>HLLC / ParaM [SYRC11]</td>
<td>2011</td>
<td>Indiana University</td>
<td>MATLAB/Octave</td>
<td>MATLAB with C++ and CUDA C</td>
</tr>
<tr>
<td>Cetus [LME09]</td>
<td>2009</td>
<td>Purdue University</td>
<td>C with OpenMP directives</td>
<td>CUDA</td>
</tr>
<tr>
<td>accULL [RLRFdS12]</td>
<td>2012</td>
<td>Universidad de La Laguna</td>
<td>C + OpenACC</td>
<td>C + Fragollo API</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(The Frangollo API supports both OpenCL and CUDA)</td>
</tr>
</tbody>
</table>
Table 2.3: Comparison of MATLAB compilers for GPU execution

<table>
<thead>
<tr>
<th>Name</th>
<th>Use annotations</th>
<th>Benchmarks</th>
<th>Base tools</th>
<th>Performed Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEGHA [PAG11]</td>
<td>Optional (for type inference)</td>
<td>Mat2C and one additional in-house benchmark</td>
<td>GNU Octave</td>
<td>Conversion to SSA, Data flow analysis, Type inference, List Scheduling variant</td>
</tr>
<tr>
<td>HLLC / ParaM (for GPUmat) [SRC11]</td>
<td>Optional (for type inference)</td>
<td>Over 50, from various sources</td>
<td>GNU Octave</td>
<td>Conversion to and from SSA, Type inference, Partial evaluation, Dead code elimination, Computation Partitioning</td>
</tr>
<tr>
<td>HLLC / ParaM (for C++ / CUDA C) [SYRC11]</td>
<td>Optional (for type inference)</td>
<td>6 benchmarks, one of which from NAS</td>
<td>GNU Octave</td>
<td>Conversion to and from SSA, Type inference, Partial evaluation, Dead code elimination, Scalarization</td>
</tr>
</tbody>
</table>
2.9 Summary

The MATLAB programming language is widely used for scientific and financial applications. However, it suffers from portability and efficiency problems. To reduce the efficiency concerns, there are APIs that allow offloading some MATLAB computations to GPUs.

Modern GPGPU programming is done in languages such as OpenCL or CUDA. CUDA is largely unsupported outside of NVIDIA GPUs, but is widely used for a variety of computations. OpenCL is supported by various GPU manufacturers and even by companies that develop other types of processors, but it is more recent and many tools only support CUDA.

Finally, there are compilers that allow executing code on GPUs without having to use GPU-specific APIs or manually writing OpenCL/CUDA code. We described approaches to compile MATLAB code to run on GPUs and compilers that used C with directives (OpenMP or OpenACC) to generate programs that exploit GPU parallelism.
Chapter 3

Compiler Phases

The MATLAB to OpenCL compiler is divided in multiple phases. Each phase performs a transformation to the intermediate representation to bring the program closer to the intended output. These phases are described in this chapter.

3.1 Directive Identification

The MATISSE framework [BPN+13] is capable of parsing MATLAB code and generates an AST. All MATISSE CL directives exist in the AST as simple comments. These directives are based on OpenACC’s [Ope13], but adapted to MATLAB. In particular, MATLAB does not have an equivalent to C/C++’s #pragma directive. We use comments starting with %acc instead. This allows MATLAB code with directives to run on tools that do not recognize our directives. These tools merely ignore the %acc directives.

We have used directives in order to give MATLAB users fine grained control over which portions of the program are offloaded to the OpenCL device and which should execute sequentially on a CPU.

Figure 3.1 shows a simple MATLAB program that uses MATISSE CL directives. This program computes the element-wise square of a given matrix (equivalent to MATLAB’s A .* A). The directives in this program indicate that the for block must be compiled to OpenCL, that the A matrix must be copied to the OpenCL device (readonly because the input matrix itself is not modified) and that Y should be copied back to the host once the loop execution is finished.

Figure 3.2 shows a MATLAB program with nested directives. In this case, a single OpenCL kernel is generated with a range of two dimensions. The observable behavior of this example is the same as Figure 3.1. If the %acc loop directive had not been included, then the outer loop would be executed in parallel, but each iteration would execute the inner loop sequentially.

In this phase, the compiler parses all comments to detect which ones are directives. After this phase is run, the AST comment nodes are replaced by directive nodes.
function Y = square_matrix(A)
    Y = zeros(size(A, 1), size(A, 2), 'single');
    %acc parallel loop copyin(readonly A) copyout(Y)
    for i = 1:numel(A)
        Y(i) = A(i) * A(i);
    end
end

Figure 3.1: A MATLAB function that squares the value of each matrix position, with added OpenACC-based directives.

function Y = square_matrix(A)
    Y = zeros(size(A, 1), size(A, 2), 'single');
    %acc parallel loop copyin(readonly A) copyout(Y)
    for i = 1:size(A, 1)
        %acc loop
        for j = 1:size(A, 2)
            Y(i, j) = A(i, j) * A(i, j);
        end
    end
end

%acc end

Figure 3.2: A MATLAB function with nested loops, with added OpenACC-based directives.
Some directive (such as parallel loop) start blocks that are terminated with an end directive. The input AST is flat, because the MATISSE parser does not recognize these blocks. However, the output AST correctly indicates that these regions are blocks with the appropriate children.

The compiler supports five types of directives:

- **parallel loop**: Indicates that a for loop should be compiled to OpenCL and have its iterations executed in parallel.
- **parallel**: Indicates that an inner for loop should be executed in parallel. This directive must be inside a parallel loop block.
- **ignore**: Indicates that a section should be ignored by MATISSE CL. These sections are still executed when run by MATLAB or MATISSE C.
- **end**: Indicates the end of a parallel, parallel loop or ignore block.
- **barrier**: Indicates a memory barrier. The compiler recognizes two different types of barriers: barrier local and barrier global. These are equivalent to OpenCL’s barrier function [Gro09]. For local and global barriers, CLK_LOCAL_MEM_FENCE and CLK_GLOBAL_MEM_FENCE are used as the function argument, respectively.

Parallel loop and parallel blocks should contain a single for or parfor block as the body. Parallel loop directives can take the following parameters:

- **copyin**(vars): A list of variables that should be copied for use by the compiled kernel. All variables used in the parallel loop block should be declared in this parameter, with the exception of the loop iteration count (e.g. \( x \) in \( \text{for } i = 1:x \)) if it is not used anywhere else in the parallel block. Matrix variables that are never modified within a parallel loop region may be prefixed with readonly. If only a portion of the variable is used in a parallel loop region, then the range copy syntax (e.g. \( A(1:100) \)) can be used.
- **copyout**(vars): A list of variables that should be copied from the compute device to the host device once the kernel finishes execution.
- **reduce**(vars:operation): A list of scalar variables that are computed partially by multiple threads. Two operations (+ for sum and * for product) are supported. This parameter is not optimized by MATISSE CL and will likely result in suboptimal performance.
- **local**(vars): A list of matrix variables that are different for each local group. The values of these matrices are not copied to the device and changes to them are discarded once the kernel execution ends. The shape of these matrices must be fully known at compile time.
- **local_id**(vars): For each iteration, the value of these variables will be set to the respective local ID. The value of these variables remains the same outside of the parallel loop block.
### Compiler Phases

#### Figure 3.3: MATLAB comments. Only the last two are recognized as OpenACC-based directives.

- **group_id(vars):** Similar to `local_id`, but the variables store the respective group IDs.
- **local_size(constants):** The local size that should be used.

If the loop coalescing optimization is disabled, then directive parameters such as `local_id`, `group_id` and `local_size` take as many arguments as there are kernel range dimensions. In other words, they take one argument for the outer parallel loop block and one more per nested parallel block. If loop coalescing is enabled, then these directives take a single parameter.

Both single line and block comments can be directives. In Figure 3.3, the first three comments are not recognized as directives and therefore remain comment AST nodes. The fourth comment is recognized as a directive. However, "foo" is not a valid directive so the compiler emits an error message. The last comment is recognized as a directive and its syntax is valid so the compiler replaces the comment node with a parallel loop directive node. Note that every parallel loop block must have a matching end directive, so the compiler would still emit an error message for the last comment.

Appendix A describes the syntax (context-free grammar) of the directive comments supported by the compiler.

We do not support all of OpenACC’s directives. However, we found that the subset we cover is enough for most applications and that the most of the missing ones would require significant work to implement.

### 3.2 Expression Decomposer

The expression decomposer is a compiler phase designed to modify certain expressions without changing the behavior of the program, in order to simplify the subsequent phases. In order to do so, it introduces new temporary variables. For instance, the code in Figure 3.4 is transformed to the code in Figure 3.5. As a result of these changes, the following post-conditions are true:

- Every access call is directly assigned to a variable (never used as part of another expression). This is important because it simplifies function inlining.
- In order to ensure that the above is true, `while(x)` loops become `while(1)` with an additional if statement that breaks the loop when the condition is false.
Compiler Phases

- All for loops are in the form `for varname1=1:1:x` (even if originally the `for` loop iterated over an array, not a range). This means that it is trivial to compute the number of iterations of any loop if `x` is known. This is useful for the OpenCL backend, which must be able to compute the number of iterations of every parallel loop before it can execute it.

```matlab
function y = test(x)
    y = 0;
    for w = 1:2:x
        while w < 10
            y = y + aux(w);
            w = w + 1;
        end
    end
end

function y = aux(x)
    y = x;
end
```

Figure 3.4: Simple MATLAB program that can be transformed by the expression decomposer.

```matlab
function y = test(x)
    y = 0;
    tmp_MaxValue1 = (x - 1) / 2 + 1;
    tmp_Iterations1 = floor(tmp_MaxValue1);
    for tmp_LoopIndex1 = 1:1:tmp_Iterations1
        while 1
            if ~(w < 10)
                break;
            end
            w = (tmp_LoopIndex1 - 1) * 2 + 1;
            tmp_AccessCall1 = aux(w);
            y = y + tmp_AccessCall1;
            w = w + 1;
        end
    end
end

function y = aux(x)
    y = x;
end
```

Figure 3.5: MATLAB program of Figure 3.4 after being decomposed.
3.3 Function Inliner

The function inliner is a compiler phase that replaces all function calls by the function body without changing the observable behavior of the application. After the inliner has been executed, the code contains no function calls so the code generator can assume that any time $A(x)$ appears it is a matrix access expression.

One of the problems of the inliner is that function calls and matrix accesses have the same syntax, but only function calls can be inlined. To handle this, the inliner has a table that keeps track of the function names, containing both user functions and predefined MATLAB functions, such as `numel`.

The other major problem that the inliner has to solve is the return statement. This statement interrupts the function that is currently being executed. However, when the code is injected on the parent function, these return statements must be removed. If this did not happen, then these statements would interrupt the caller function as well and change the observable behavior. In order to prevent this, a new variable is introduced to keep track of whether execution of the current function should stop and new if and break statements are introduced so that code after the return statement is not executed and any loops are interrupted. Figure 3.6 shows a program with early returns. This function is modified to become identical to the one in Figure 3.7.

```matlab
function y = early_return(x)
    if x > 0
        y = 1;
        return;
    end
    y = 2;
end
```

Figure 3.6: MATLAB function that is not yet ready to be inlined because it contains return statements.

Finally, since the caller and the callee functions may use variables with the same names, callee variables must be renamed when they are inserted into the parent function.

3.4 Directive Cleaner

As a result of the expression decomposer, some directives may no longer be correct. Consider the program in Figure 3.8. After the expression decomposer, the result is the code in Figure 3.9. Note that the directive block now contains more than just a `for` loop and is, as such, incorrect.

The directive cleaner is the compiler phase that corrects the programs so that they have the expected format again. It works by extracting the statements before the `for` loop to outside the directive region block. The result of this phase is shown in Figure 3.10.
function y = early_return(x)
    k_return = 0;
    if x > 0
        y = 1;
        k_return = 1;
    end
    if ~k_return
        y = 2;
    end
end

Figure 3.7: MATLAB function with the same behavior as the one in Figure 3.6. This version does not contain any return statements and therefore can be inlined.

function A = test(x)
    A = ones(x, x, ’single’);
    %acc parallel loop copyin(A) copyout(A)
    for w = 1:x * 2
        A(w) = w;
    end
end

Figure 3.8: Simple MATLAB function with directives that can be transformed by the expression decomposer.

function A = test(x)
    A = ones(x, x, ’single’);
    %acc parallel loop copyin(A) copyout(A)
    tmp_Iterations1 = floor(x * 2);
    for tmp_LoopIndex1 = 1:tmp_Iterations1
        w = tmp_LoopIndex1;
        A(w) = w;
    end
end

Figure 3.9: Function from Figure 3.8 after the expression decomposer has been executed.

After the directive cleaner is run, if the number of iterations is not a constant, the variable containing the number of iterations (e.g. tmp_Iterations1) is added to the copyin parameter.
3.5 Directive Region Outlining

At this point, the source code is nearly ready for code generation by the MATISSE CL backend. Not all code should be parallelized (only the sections with directives). The code sections to be executed in sequence should be handled by the MATLAB C backend, which communicates with the OpenCL sections through wrapper code. This means that different parts of the code are compiled by different backends.

To simplify code generation, we decided that the choice of which backend to use to compile a function should be performed on a per-function basis. However, since a function may have both sequential and parallel regions, we need to split such functions. This is what the directive region outliner does.

The outliner detects all directive region blocks and creates a new function for each one. It keeps track of which functions were generated in a list. The functions on this list are to be compiled by the CL backend. The main function (the caller) is compiled by the C backend.

The copyin, copyout and reduce parameters are used to define which variables should be used as function arguments and which variables to return from the outlined functions.

Figure 3.11 shows a MATLAB program after the expression decomposer, the inliner and the directive cleaner have been run, but before the outliner has. Figure 3.12 shows the transformed program after the outliner. Note that the main function no longer contains any directives, and that the extracted function contains no code outside the directive block.

3.6 Code Generation

At this point, we have a single function to be compiled by the C backend and a list of functions to be compiled by the OpenCL backend. A list of function names is obtained from the list of outlined functions. The main function, along with this list, is passed to the C backend which compiles the code as it would even if the CL backend did not exist. However, when the function attempts to call a function on the outlined list, the CL backend intercepts the call and generates the code for that function.
When the OpenCL backend generates code for a function, it outputs two versions of the function: the wrapper, a pure C function with OpenCL API calls to run the parallelized code and copy the data in and out of the OpenCL device, and the OpenCL code itself. The C backend receives the wrapper code and is unaware of the OpenCL program itself. When the compilation is over, the OpenCL functions are aggregated on a single file, which is stored alongside the generated executable.

The CL backend compiles the code on a per-statement basis using a simple type inference engine that assumes that each variable remains of the same type it was first assigned with.

Consider the MATLAB program in Figure 3.8. Figure 3.13 shows the automatically generated OpenCL code. This program contains some boilerplate structures and matrices (in this case, `global_float_mat2_t` and `matrix_set_magf42_1`) that is used to simplify the rest of the code. The kernel is generated for the extracted function. The arguments of the kernel are obtained from the arguments of the outlined function. Each matrix is passed to the kernel as a pointer to the data and one scalar per matrix dimension indicating the size. The line `A(w) = w;` is translated to `matrix_set_mgf42_1(A, w, ((float) w));`.

Figure 3.14 shows the wrapper code that is generated for this example. The input and output
Compiler Phases

```c
typedef struct {
    global float * data;
    int dim1;
    int dim2;
} global_float_mat2_t;

float matrix_set_mgf42_1(global_float_mat2_t mat, int pos, float value)
{
    return mat.data[pos - 1] = value;
}

/*
 * tmp_Iterations1 threads
*/
kernl void test_extracted1_mgf42s4(global float * Adata, int Adim1, int Adim2)
{
    size_t thread_id1;
    int w;
    size_t global_size1;
    int tmp_Iterations1;
    global_float_mat2_t A;

    thread_id1 = get_global_id(0);
    w = thread_id1 + 1;
    global_size1 = get_global_size(0);
    tmp_Iterations1 = global_size1;
    A.data = Adata;
    A.dim1 = Adim1;
    A.dim2 = Adim2;
    matrix_set_mgf42_1(A, w, ((float) w));
}
```

Figure 3.13: The OpenCL code generated from the MATLAB program from Figure 3.8.

types of this function are native MATISSE C types. However, the implementation code is completely different. The generated code consists of OpenCL API calls to copy the data, prepare the kernel and execute it. When the function is over, the allocated OpenCL resources are released.

3.7 Summary

In this chapter, we presented the phases that our compiler prototype uses in order to generate C and OpenCL code from an input MATLAB file. We start by parsing the input file to an AST, we decompose certain statements and expressions so that the subsequent phases can be simplified. Once that is over, we perform inlining so that the code generator does not need to deal with function calls and directives in called functions. After function inlining, we apply the directive cleaner, that fixes any directives that might have become incorrect due to changes performed by previous phases. Finally, we outline the parallel regions and generate the C and OpenCL code.
void test_extracted1_ptfi(tensor_f ** A, int tmp_Iterations1, tensor_f ** A_out)
{
    size_t thread_count1;
    cl_mem Adata;
    cl_kernel kernel;
    cl_int retval;
    cl_int Adim1;
    cl_int Adim2;
    size_t thread_count_range[1];
    cl_event kevt;
    cl_ulong profile_start;
    cl_ulong profile_end;

    thread_count1 = tmp_Iterations1;
    Adata = clCreateBuffer(context->context, CL_MEM_READ_WRITE |
                             CL_MEM_COPY_HOST_PTR, sizeof(float) * (*A)->length, (*A)->data, &retval);
    clhelper_check_return("clCreateBuffer", retval);

    kernel = clCreateKernel(context->program, "test_extracted1_mgf42s4", &retval);
    clhelper_check_return("clCreateKernel", retval);
    retval = clSetKernelArg(kernel, 0, sizeof(cl_mem), &Adata);
    clhelper_check_return("clSetKernelArg", retval);
    Adim1 = (*A)->shape[0];
    retval = clSetKernelArg(kernel, 1, sizeof(cl_int), &Adim1);
    clhelper_check_return("clSetKernelArg", retval);
    Adim2 = (*A)->shape[1];
    retval = clSetKernelArg(kernel, 2, sizeof(cl_int), &Adim2);
    clhelper_check_return("clSetKernelArg", retval);

    thread_count_range[0] = thread_count1;
    retval = clEnqueueNDRangeKernel(context->command_queue, kernel, 1, NULL, thread_count_range, NULL, NULL, NULL, &kevt);
    clhelper_check_return("clEnqueueNDRangeKernel", retval);

    /* Get profiling data */
    clWaitForEvents(1, &kevt);
    clGetEventProfilingInfo(kevt, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &profile_start, NULL);
    clGetEventProfilingInfo(kevt, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &profile_end, NULL);
    printf("KERNEL: %e\n", (profile_end - profile_start) / 1.0e9);

    copy_alloc_f(*A, A_out);
    retval = clEnqueueReadBuffer(context->command_queue, Adata, CL_TRUE, 0, sizeof(float) * (*A_out)->length, (*A_out)->data, 1, &kevt, NULL);
    clhelper_check_return("clEnqueueReadBuffer", retval);

    retval = clReleaseKernel(kernel);
    clhelper_check_return("clReleaseKernel", retval);
    retval = clReleaseMemObject(Adata);
    clhelper_check_return("clReleaseMemObject", retval);
}

Figure 3.14: The wrapper code generated from the MATLAB program from Figure 3.8, when profiling mode is enabled.
Compiler Phases
Chapter 4

Compiler Prototype

Our compiler prototype is able to generate C and OpenCL code from modified MATLAB source files. Our compiler is written in Java 8 and is based on the MATISSE framework. We use two intermediate representations, both implemented as trees. One of these representations is the MATLAB AST generated by MATISSE, and the other is the C tree representation that the C backend uses to generate code. This chapter describes the architecture and inner-workings of our compiler.

4.1 Workspace Structure

MATISSE consists of multiple projects, each handling a different subsystem. We took care to ensure each subsystem in isolated in a separate project. Naturally, some projects depend on others. In this section, we describe the most important projects (from the point of view of the OpenCL backend).

Figure 4.1 shows the various projects and what they depend on. The dependencies of MatlabToCLTester were not shown in the chart because it depends on all the remaining projects listed in this section.

**MatlabIR** Contains the MATLAB AST node definitions. The MATLAB AST is explained in more detail in Section 4.3.

**MatlabProcessor** The MATLAB parser. Builds a MatlabIR AST from a MATLAB source file. Since it depends on the AST classes, this project depends on the MatlabIR.

**CIR** The C intermediate representation. See Section 4.4 for further details.

**MatlabToCLib** Includes the C implementation of the built-in MATLAB functions.
**CMainFunction**  Includes the implementation of the C\_main() function, as well as utility functions it uses, such as benchmarking code.

**MatlabToC**  Includes the compiler GUI code to allow generating the C code files. This project can be executed by the user.

**MatlabToCTester**  Another compiler GUI for the C backend. This project extends MatlabToC with the ability to generate Makefiles and compile the generated C code. It is also possible to execute the generated C files and run the same program in MATLAB to check the correctness of the results.

**MatlabIRTransformations**  Includes a set of source-to-source MATLAB AST transformations. Several of the transformations mentioned in Chapter 3 are implemented in this project. It was developed to be used by the OpenCL backend, and is not currently used by any other MATISSE backend. However, the transformations in this project are completely independent of OpenCL itself, so it was moved to an independent project. For instance, any backend could use inlining.

**MatlabToCL**  This project depends on MatlabIRTransformations and MatlabToCLib and implements all OpenCL-specific code transformations, notably code generation of both OpenCL and C
wrapper code. This project does not generate the C `main()` function or the Makefiles and is not meant to be directly executed by the user.

**MatlabToCLTester**  The compiler GUI for the OpenCL backend. It is able to generate the OpenCL and C code, including the `main` function, and the appropriate Makefiles. It also includes the ability of compile the generated code. This is the main project for the OpenCL backend.

In addition to these, there are JUnit-based projects to handle automated testing. See Section 4.5 for further details.

In order to obtain the elapsed time for each benchmark, we used a Python script to automatically run the test executables multiple times and generate a CSV file with the results. This script is not considered part of the MATISSE project.

### 4.2 MATISSE Integration

The integration of the OpenCL backend projects with MATISSE is mostly handled with project dependencies. For instance, MatlabIRTransformations is able to use MATLAB AST nodes by including MatlabIR in the build path.

Projects such as MatlabToC and MatlabToCLTester define the structure of their configuration files, which store the inputs they require in order to perform any operations. These files include data such as source code path, data inputs and optimization levels. The structure of these files is automatically used to generate the GUI.

The recommended approach to use MatlabToCLauncher is to use these configuration files. For MatlabToCLTester, we also define a configuration file and the automatically generated GUI. In order to interface with MatlabToCLauncher, however, we bypass the file storage and create the equivalent class instances directly.

Not all configuration features are accessible for GUI users. Notably, the C backend includes the feature to intercept certain MATLAB function calls and use custom instance providers. This feature is not exposed by the GUI, but it is used extensively by the OpenCL backend (as described in Section 3.6).

Because configuration options that consist of class instances are serialized and later deserialized, it is important to be careful when relying on the state of these instances. The C backend will use a copy of the instances and not the original instances themselves. For instance, we use a class named `CLProgram` that aggregates all generated kernels so that we can later generate the equivalent OpenCL file. Every time a wrapper function instance is created, the equivalent OpenCL function instance is created and added to the `CLProgram`. However, because of the serialization step, the `CLProgram` that the functions are being added to is a copy. The solution we used was to ensure that the OpenCL file generation also used the copy, rather than the original.
4.3 MATLAB AST

The initial transformations, that is, all the transformations described in Chapter 3 before Code Generation, use the MATLAB-based AST. However, starting with the Directive Identification phase, some of the nodes in the AST are directives, meaning they are no longer comments.

Figure 4.2 shows a simple MATLAB function. Figure 4.3 is the equivalent AST representation, obtained using the toString() method of the root node.

```
function A = f(x)
% Hello, World
A = zeros(x);
%acc parallel loop copyout(A)
for i = 1:x*x
    A(i) = i;
end
%acc end
end
```

Figure 4.2: A MATLAB program meant to demonstrate what the AST looks like.

The root node of any MATLAB file parsed by the MatlabProcessor is File. A file may be a Script, if it contains no functions, or one or more Function nodes. A Function may contain Statement and Block nodes. All statements have a StatementType, which identifies the type of statement without the need to further inspect the child nodes. All Function contain one Statement node of type FunctionDeclaration, which indicates the inputs, name and outputs of the function. Block nodes are used to represent control flow statements. The first child is a Statement node that indicates the type of block, namely whether it is an If or While block. The remaining children are the body of the statement.

Every node may have a content, which is a value whose type depends on the token type. For instance, the content of a Statement node is the line it appears in, whether it has a semicolon at the end and the type of the statement and the content of a Comment node is the text content of the comment.

4.4 CIR

The CIR, C Intermediate Representation, is a tree that is created in order to assist the generation and validation of the C code output. The C IR does not intend to fully cover the C language. It is tailored for the specific needs of MATISSE. The main concepts of the C IR are:

**Nodes** Represents something in the C tree, such as a Statement. The purpose of every node is to be eventually converted to C code, using a method called getCode(). The simplest type of node is a literal, which represents a section of code to be injected into the file. Other types of
Figure 4.3: AST Representation of the program in Figure 4.2.

nodes include keywords, blocks, numeric literals and function calls. The decision of when to use a literal depends on what is the most convenient for the code generator. These are represented by the class CToken, although this is a misnomer because tokens are typically atomic units of the code, and this class is not necessarily atomic. For this reason, we are planning to rename this class to CNode.

Instruction Lists  Represents a set of C instructions (statement nodes). Instruction lists can be used to define the body of a function or a block.

Types  Types represent the equivalent concept of C. C expressions and variables have types, and they do so as well in the CIR. The CIR checks the types of nodes according to a set of rules to
Compiler Prototype

ensure correctness and either emits an error message or injects the necessary casts when the types do not match. The OpenCL backend extends the CIR with its own OpenCL-specific types, such as `cl_mem`.

**Function Instances**  A function instance is the concept of a C function, which may be built-in, such as `malloc`, or an user-defined function. Generally, function instances have a declaration (added to the C header) and an implementation (added to the code file itself). The exception being inline functions, which have neither. Inline functions are instances that instead inject their code directly on the caller location. Function instances also have a set of dependencies - the functions they depend on.

**Instance Providers**  MATLAB is dynamically typed, but C is statically typed. Some MATLAB functions, such as `zeros`, return different types depending on the inputs they were given. Matisse handles this by making every version of the function (that is, every combination of input and output types) be a separate function instance. Each MATLAB function has an associated instance providers which returns the appropriate function instance depending on the given input. The OpenCL backend extends the CIR with new instance providers to be used by the generated OpenCL code.

The CIR does not have the concept of operator. Instead, operators are handled with instance providers that return inline function instances.

Figure 4.4 shows what a CIR tree looks like. Figure 4.5 shows the equivalent C/OpenCL code. In this example, a function named `example_function_name` is called with two inputs and one output. The variable nodes indicate the variable name and its type. When the variables are constants, then the value of the constant is part of the type. The generated functions are always specialized for their input types. Since a MATLAB function may be specialized multiple times (for different input types), a suffix is required to distinguish the multiple versions that may exist - that is what the `ui32ui32` and `ui32tf` suffixes are for. Note that these suffixes are generated by the MATLAB to CIR compiler phase. In CIR itself, there is no distinction between these suffixes and the proper function name.

### 4.5 Validation

We used two main methods to determine whether the compiler was outputting the correct code: a JUnit test suite and a benchmark suite.

We used a JUnit test suite with a set of test programs and the equivalent expected output. We executed this test suite whenever we changed the compiler. As we found new programs that did not work with our tool, we added them to our test suite to ensure that once we fixed them, we would never break them again with future changes to the code (regression testing). Although each compiler phase has its own tests, most tests depend on a variety of modules to work.
Figure 4.4: A CIR instruction list before being converted to C code.

```c
void test_function_ui32tf(uint32_t variable_name, tensor_f* matrix) {
    uint32_t constant_name;
    constant_name = 100;
    variable_name = 200;
    example_function_ui32ui32tf(constant_name, variable_name, &matrix);
    // End of function
    return;
}
```

Figure 4.5: The generated C/OpenCL code from a function with the CIR tree of Figure 4.4.

For the benchmarks, we ran the programs described in Chapter 5 and compared the results to MATISSE C and MATLAB. The results are occasionally different due to rounding errors in floating point calculations, but we found that any significant differences were caused by bugs in our tool and ceased to exist when fixed.

4.6 Limitations

The current version of tool currently has several limitations. On certain occasions, the programmer needs to manually rewrite some sections of code in order to take advantage of the available directives. Certain features take only a reduced number of operations in MATLAB but need to be transformed into a longer loop in order to apply directives. This limitation is studied in greater detail in Section 5.2.
Additionally, the way we currently schedule OpenCL operations is suboptimal. We currently create the OpenCL command queue in the in-order execution mode (see Subsection 2.3.2). We also generate some unnecessary data transfers. After a kernel is executed, we immediately copy all outputs back to the host, even if the data is not immediately needed. Ideally, we should continue the execution until there is a data dependency so that the host and the device can execute simultaneously.

Another problem is the fact that we currently do not support vectorial/matrix operations within a parallel section. Finally, not all MATLAB functions are supported. Overall, this means that the code with directives tends to not be what a MATLAB programmer would normally write. In other words, it’s not idiomatic.

4.7 Summary

This chapter described the various components and data structures that our compiler is based on. We were able to develop automatically generate OpenCL by reusing a significant portion of the MATISSE framework extended with an OpenCL backend. The rest of MATISSE still has no dependency on the OpenCL backend. All the Intermediate Representations we use are from the MATISSE project, with only small adjustments.

Overall, the biggest problem of our compiler is that the programmers need to manually change their MATLAB programs to a non-idiomatic style if they want their programs to be compiled by the OpenCL backend.
Chapter 5

Experimental Results

This chapter describes the tests that were performed to evaluate the quality of the generated executables and the results we obtained. We also study the impact of the required code modifications in the input MATLAB files.

5.1 Benchmarks and Environment

All tests were executed on a desktop computer running Windows 8.1 Enterprise Edition 64-bits with an AMD A-10 7850K CPU reported by Windows as running at 4.10 GHz \(^1\), 8 GB RAM (6.94 GB usable). This computer has two GPUs (one integrated with the CPU and the other discrete). The integrated GPU is the Radeon R7 Graphics included in the aforementioned CPU and the discrete GPU is an AMD Radeon R9 280X \(^2\). This computer is running the official AMD drivers, reported as Version 14.10.1006-140417a-171099C (Catalyst version 14.4). Clinfo, a command-line utility, reports Platform Version OpenCL 1.2 AMD-APP (1445.5). All code was compiled on Windows with GCC 4.8.2 (64-bits MinGW) with the \(-O3\) optimization level.

We used the following benchmarks:

- **cpxdotprod3** - Multiplies two complex 3D matrices. Because MATISSE C currently does not support complex numbers, the real and imaginary components were supplied as separate matrices (for a total of 4 single-precision floating point matrices) [CDCP13].

- **dilate** - Function from a Stereo navigation application [CDCP13].

- **matmul** - Simple \(\Theta(n^3)\) single-precision floating point matrix multiplication.

- **matmul_nv** - OpenCL Matrix Multiplication code sample by NVIDIA [NVI14b], manually converted to MATLAB with directives. This benchmark was neither run on MATLAB nor

\(^1\)The official specification indicates that the CPU clock speed is 4.0 GHz / 3.7 Ghz [AMD14]

\(^2\)Reported by the clinfo command-line utility as an AMD HD 7900 Series
Experimental Results

the MATISSE C backend, since it relies on OpenCL-specific features (such as barriers) to work properly. Appendix B shows a comparison of this file with the matmul benchmark.

- **monte_carlo_option_pricing** - A Monte Carlo simulation based on an example by MathWorks [Mat14e] but modified to use a linear congruential pseudo-random number generator [Wei14b] and a Box-Muller transform [Wei14a], instead of the native MATLAB randn (which MATISSE C and OpenCL backends do not currently support).

- **rgb2yuv** - A simple conversion of a RGB uint8 matrix to the YUV color space equivalent.

- **subband_2d** - Function translated from part of a C MPEG2 encoder [CDCP13] and generalized for different input sizes.

The cpxdotprod3, dilate and subband_2d are modified versions of the benchmarks with the same names from the REFLECT project [CDCP13] previously also used in [BPN+13].

The matmul_nv benchmark can be seen on B and is meant to show how to tune a MATLAB program to perform as close to manual OpenCL as possible.

Although our tool supports double precision floating point numbers, all these benchmarks were run with single-precision.

We were also able to run some preliminary tests on an embedded ODroid SOC, but we have not been able to observe any speedups at all on that system.

5.2 Impact of Directives and Compatibility Adjustments on Code Size

MATLAB code occasionally needs to be modified in order to compile correctly in MATISSE. Additional code modifications are required in order to generate OpenCL code, including adding directives and replacing matrix operations by the equivalent for loops. To understand the impact of these changes, consider Figure 5.1, which shows the idiomatic version of the cpxdotprod3 benchmark, and contrast it with Figure 5.2, which shows the non-idiomatic equivalent. The cpxdotprod3 is a worst-case scenario for required code changes.

```matlab
function [Creal, Cimag] = cpxdotprod3(Areal, Aimag, Breal, Bimag)
% complex dot product

Creal = Areal .* Breal - Aimag .* Bimag;
Cimag = Areal .* Bimag + Aimag .* Breal;

end
```

Figure 5.1: Idiomatic version of the cpxdotprod3 benchmark.
Experimental Results

```matlab
function [Creal, Cimag] = cpxdotprod3(Areal, Aimag, Breal, Bimag)
% complex dot product

numElements = numel(Areal);

Creal = zeros(1, numElements, 'single');
Cimag = zeros(1, numElements, 'single');

%acc parallel loop copyin(readonly Areal, readonly Aimag, readonly Breal, readonly Bimag, numElements) copyout(Creal, Cimag)
for j=1:numElements
    index=j;
    Ar = Areal(index);
    Ai = Aimag(index);
    Br = Breal(index);
    Bi = Bimag(index);
    Creal(index) = Ar*Br-Ai*Bi;
    Cimag(index) = Ar*Bi+Ai*Br;
end
%acc end
end
```

Figure 5.2: cpxdotprod3 benchmark with directives.

Figure 5.3 shows the MATISSE CL backend input MATLAB code size compared to the equivalent idiomatic size, in factor of lines of code. Empty lines and non-directive comments are ignored and not counted. The `matmul` and `matmul_nv` benchmarks were not measured because there is no "idiomatic" code for them. An idiomatic approach would be to simply multiply two matrices using the built-in product operator. However, this would mean that MATLAB could be using a different matrix multiplication algorithm. As Figure 5.3 shows, there is always a significant number of added lines, though the exact impact depends on the benchmark.

For the `monte_carlo_option_pricing` benchmark, the idiomatic code excludes the Random Number Generation functions (except the initial seed definition), since MATLAB includes its own built-in pseudo-random number generators. If the same code for the random number generation is considered for both versions, then the MATISSE CL backend version contains only 7.9% more code than the idiomatic version.

5.3 Comparison of the MATISSE C and OpenCL Backends

We have tested the performance of the C and OpenCL backends for various benchmarks. Although the C backend correctly compiles the `matmul_nv` example, it executes a significantly different algorithm because implementation details such as local memory are ignored. As such, the results
would not be fair so we have decided not to show the results for this benchmark here. The simple `matmul` example does not have this problem and its results are reported here.

Although the MATISSE CL backend allows the programmer to manually specify explicit local size for each loop, none of the benchmarks shown in this section do so and instead use the default local size of `NULL`. This means that the decision of what local size to use is made by the OpenCL driver automatically.

We have measured two variants of the CL code. Each test was executed 30 times and the results were averaged. The standard deviation of the execution times was also computed. The speedups reported in Figure 5.4 include the overhead of data transfer and other driver calls such as setting kernel arguments and obtaining the OpenCL kernel instance and the execution time is measured using the `QueryPerformanceCounter` Windows API function. The chart label also indicates the sizes of the input data matrices. The `monte_carlo_option_pricing` benchmark does not receive matrices as input, so instead the label reports the input scalar $N$. The speedups reported in Figure 5.5 are relative to the time spent only on kernel execution, using the built-in OpenCL profiling capabilities.

The standard deviation was generally less than 10% of the average elapsed time, with 3 exceptions: the integrated GPU in the `cpxdotprod3` benchmark (kernel execution time only) results have a standard deviation of around 18.2% of the average time, as seen in Figure 5.6. Even in this example, the results were generally consistent, with a few outliers. The CPU OpenCL device running the `cpxdotprod3` benchmark (including data transfers) results have a standard deviation of 10.7%. Finally, the discrete GPU device running the `matmul` benchmark (including data transfers) results have a standard deviation of 10.9% of the average time. The outliers in all these three cases are a few executions that take longer to execute. The first execution of the OpenCL CPU device `cpxdotprod3` benchmark is the only outlier of that benchmark. If removed, the standard deviation would be less than 3% of the average time.
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Integrated GPU</th>
<th>Discrete GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cpxdotprod3_single-2048x2048x20</strong></td>
<td>6.01</td>
<td>3.09</td>
<td>1.80</td>
</tr>
<tr>
<td><strong>dilate_single_2048x2048</strong></td>
<td>14.88</td>
<td>2.00</td>
<td>19.14</td>
</tr>
<tr>
<td><strong>matmul_single_1024x1024</strong></td>
<td>19.14</td>
<td>3.81</td>
<td>149.08</td>
</tr>
<tr>
<td><strong>rgb2yuv_uint8_2000x2000</strong></td>
<td>1009.60</td>
<td>1.80</td>
<td>3.81</td>
</tr>
<tr>
<td><strong>monte_carlo_option_pricing_N1e6</strong></td>
<td>18.21</td>
<td>3.58</td>
<td>14.88</td>
</tr>
<tr>
<td><strong>subband_2d_partial_single_128x64ki</strong></td>
<td>739.32</td>
<td>14.89</td>
<td>739.32</td>
</tr>
</tbody>
</table>

Geometric Mean

<table>
<thead>
<tr>
<th></th>
<th>0.30</th>
<th>0.19</th>
<th>0.10</th>
</tr>
</thead>
</table>

**Figure 5.4**: Speedup of OpenCL code relative to C, including overhead of OpenCL calls and data transfers.

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Integrated GPU</th>
<th>Discrete GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>cpxdotprod3_single-2048x2048x20</strong></td>
<td>1.28</td>
<td>7.72</td>
<td>14.89</td>
</tr>
<tr>
<td><strong>dilate_single_2048x2048</strong></td>
<td>3.10</td>
<td>2.88</td>
<td>2.51</td>
</tr>
<tr>
<td><strong>matmul_single_1024x1024</strong></td>
<td>14.89</td>
<td>3.86</td>
<td>5.09</td>
</tr>
<tr>
<td><strong>rgb2yuv_uint8_2000x2000</strong></td>
<td>161.56</td>
<td>88.49</td>
<td>307.49</td>
</tr>
<tr>
<td><strong>monte_carlo_option_pricing_N1e6</strong></td>
<td>844.67</td>
<td>14.89</td>
<td>844.67</td>
</tr>
<tr>
<td><strong>subband_2d_partial_single_128x64ki</strong></td>
<td>1175.78</td>
<td>14.89</td>
<td>1175.78</td>
</tr>
</tbody>
</table>

Geometric Mean

<table>
<thead>
<tr>
<th></th>
<th>1.28</th>
<th>7.72</th>
<th>14.89</th>
</tr>
</thead>
</table>

**Figure 5.5**: Speedup of OpenCL code relative to C. Times measured by the built-in OpenCL functions and relative only to the kernel execution time.

### 5.4 Impact of Manual Local Size Definition

OpenCL allows programmers to manually specify the local size or let the OpenCL platform automatically define it. Some algorithms depend on specific local sizes but, even for those not depending on any specific local size, there are significant performance differences between different local sizes for the same inputs. The CL backend of MATISSE also allows local size definition with the `local_size` directive parameter.

Figure 5.7 shows the impact of manual local size definitions on the performance of the test machine’s discrete GPU. The `matmul` benchmark was loop coalesced so that only a single dimension has to be specified for the local size. The automatic local size found the optimal value for the `cpxdotprod3` benchmark, and the `monte_carlo_option_pricing` benchmark achieved an improvement of 17%.
Experimental Results

Figure 5.6: Kernel execution times for the cpdodtprod3 benchmark for the integrated GPU using the built-in OpenCL profiling feature.

Figure 5.7: Speedup of various benchmarks with different local sizes, compared to an automatic local size definition.


5.5 Summary

This chapter presented the results achieved by our compiler considering a number of representative benchmarks.

In spite of having to modify the input MATLAB files, we found that the results were very encouraging, particularly for the Monte Carlo benchmark. When we exclude data transfers, only a single benchmark/machine combination suffered slowdowns and the best improvement we got was a speedup of over 2000 times over the C backend. When we include data transfers, the results are somewhat less impressive, but we were still able to get good results, the best case being a bit over 1000 times faster. When we compare our times with MATLAB’s runtime, rather than MATISSE C’s, the results become even more promising.
Experimental Results
Chapter 6

Conclusion

This chapter presents the conclusions of this thesis, namely what was developed, what the limitations of our prototype are and what work is left to do.

6.1 Concluding Remarks

During the development of this dissertation, we created a compiler capable of generating C and OpenCL code from MATLAB input files with directives based on OpenACC. To the best of our knowledge, this is the first compiler that exists to do so.

Using this compiler, we were able to significantly speed up most of the MATLAB benchmarks we tested, though not all. Although the kernel execution times themselves showed speedups in all but one case, the overhead from data transfers significantly worsens most results, causing severe slowdowns in two cases. The best results were matmul and monte_carlo_option_pricing, where the amount of data transferred is small compared to the computations that need to be performed.

The main problem with our approach are the manual changes to the original MATLAB source code that are required for MATISSE to generate OpenCL code. The directives tend to add few lines to the total number of lines of code, but certain idiomatic MATLAB programs need to be significantly modified because matrix operations need to be converted to for loops.

6.2 Future Work

Future improvements to the compiler should include directive inference. The current implementation requires the user to specify information that is redundant and should be easily statically inferred by the compiler. A future version of the tool should avoid the copyin and copyout parameters entirely, except for the case when only portions of a matrix need to be copied. One possible approach would be to an aspect-oriented language, such as LARA, to automatically insert the missing directives.
Conclusion

Additionally, data transfers could be better optimized, by removing unnecessary transfers and maximizing the portion of computations that are performed simultaneously by the CPU and the GPU.

Finally, we should adapt the compiler to properly handle and parallelize vector operations.
References


REFERENCES


REFERENCES


REFERENCES


REFERENCES


Appendix A

Directive Syntax

This appendix describes the syntax of the directives in the MATISSE OpenCL backend. A directive is a MATLAB comment that starts with "acc". This comment may be a MATLAB single-line comment, or a block comment. Line breaks are not allowed in the content of a single-line directive.

Keyword and identifier tokens are separated by whitespace characters or line breaks. Symbols, such as ’(’ and ’:’, automatically terminate keywords and identifiers.

A.1 Directive Syntax

Each directive has the following syntax:

- Comment symbol (%, for single line comments, or %{}, for block comments).
- Zero or more whitespace or line break characters. For directives in block comments, the line break is mandatory.
- acc token
- Directive content
- Zero or more whitespace or line break characters. For directives in line comments, the line break ends the directive. For directives in block comments, the line break is mandatory.
- The %} symbol, if the directive is in a block comment.

A comment that does not follow this syntax is not considered a directive and is ignored by the compiler just like any other comment would be. A comment that follows this syntax is considered a directive. If the content does not obey the correct syntax, an error is emitted and the compiler refuses to proceed.

A.2 Supported Directives

The following directives are supported:
Directive Syntax

Parallel loop - Declares a parallel section

- parallel keyword
- loop keyword
- For each parameter:
  - Directive parameter

Loop - Indicates that a nested loop is parallel

- loop keyword

Ignore - Indicates that section should not be compiled and instead be considered a comment. MATLAB still executes this section normally.

- ignore keyword

Barrier - Indicates that the OpenCL code should include a barrier at the specified location.

- barrier keyword
- local keyword or global keyword

End - Indicates the end of a directive region.

- end keyword

A.3 Parallel Loop Directive Parameters

The parallel loop directive has a number of parameters to configure the code generation. These parameters are listed below:

Copy in - Indicates the list of variables that should be copied to the device before the execution of the kernel can begin.

- copyin keyword
- The ( symbol
- Optionally, the readonly keyword
- The name of the first variable to copy
- For every additional variable to copy to the device:
Directive Syntax

- The , symbol
- Optionally, the readonly keyword
- The name of the variable to copy

  The ) symbol

Copy out - Indicates the list of variables that should be copied from the device after the execution of the kernel is finished.

  • copyout keyword
  • The ( symbol
  • The name of the first variable to copy
  • For every additional variable to copy from the device:
    - The , symbol
    - The name of the variable to copy
  • The ) symbol

Local Size - Indicates the local size of the kernel to be executed.

  • local_size keyword
  • The ( symbol
  • The size of the first dimension
  • For each additional dimension:
    - The , symbol
    - The size of the current dimension

Local - Indicates the list of local (not shared across work groups) variables.

  • local keyword
  • The ( symbol
  • The name of the first local variable
  • For each additional variable:
    - The , symbol
    - The name of the variable
  • The ) symbol
Directive Syntax

**Local ID** - Indicates the list of variables to store the local IDs.

- local_id keyword
- The ( symbol
- The name of the variable to store the local ID for the first range dimension.
- For each additional variable:
  - The , symbol
  - The name of the variable to store the local ID for the current range dimension.

**Group ID** - Indicates the list of variables to store the group IDs.

- group_id keyword
- The ( symbol
- The name of the variable to store the group ID for the first range dimension.
- For each additional variable:
  - The , symbol
  - The name of the variable to store the group ID for the current range dimension.

**Reduce** - Performs one or more reduction operations.

- reduce keyword
- ( symbol
- The first reduction to perform.
- For each additional reduction:
  - The , symbol
  - The reduction to perform.
- ) symbol

Each reduction has the following syntax:

- The name of the variable to reduce
- The : symbol
- The reduction operation symbol (either + or *)
Appendix B

Comparison of Matrix Multiplication code versions

This appendix shows various versions of the matrix multiplication code. In idiomatic MATLAB, multiplications are written as \( Y = A \times B \). Figure B.1 shows how matrix multiplication is done in the matmul benchmark. Finally, Figure B.2 and shows an optimized matrix multiplication program written in MATLAB with directives, based on an NVIDIA example. MATLAB and the NVIDIA examples use different conventions to order indices (row-major vs column-major). Reading a row-major matrix as column-major (or vice-versa) is equivalent to getting the transpose of the matrix. \( \text{transpose}(A \times B) = \text{transpose}(B) \times \text{transpose}(A) \), so we just swap the \( A \) and \( B \) matrices and use the height when the original algorithm used width. The shrRoundUp function computes the result of the division rounded up to the nearest integer.

```matlab
1 Y = zeros(size(A, 1), size(B, 2), 'single');
2 %acc parallel loop copyin(readonly A, readonly B) copyout(Y)
3 for y = 1 : size(A, 1)
4 %acc loop
5 for x = 1 : size(B, 2)
6 acc = single(0);
7 for pos = 1 : size(A, 2)
8 acc = acc + A(y, pos) * B(pos, x);
9 end
10 end
11 Y(y, x) = acc;
12 %acc end
13 end
14 %acc end
```

Figure B.1: Source code for the matrix multiplication benchmark.
Comparison of Matrix Multiplication code versions

```matlab
function Y = matmul_nv(B, A, BLOCK_SIZE)
Y = zeros(size(B, 1), size(A, 2), 'single');

%acc ignore
BLOCK_SIZE = 1;
%acc end

As = zeros(BLOCK_SIZE, BLOCK_SIZE, 'single');
Bs = zeros(BLOCK_SIZE, BLOCK_SIZE, 'single');
%acc ignore

tx = 1;
ty = 1;
%acc end

global_size0 = shrRoundUp(BLOCK_SIZE, size(C, 1));
global_size1 = shrRoundUp(BLOCK_SIZE, size(C, 2));

% acc parallel loop
copyin(A, B, BLOCK_SIZE, global_size0, global_size1)

local_size(BLOCK_SIZE, BLOCK_SIZE)
local_id(tx, ty) group_id(bx, by)

for g0 = 1 : global_size0
    %acc loop
    for g1 = 1 : global_size1
        %acc ignore
        bx = g0; by = g1;
        %acc end
        uiWA = uint32(size(A, 1)); uiWB = uint32(size(B, 1));
        uiHA = uint32(size(A, 2));
        aBegin = uiWA * BLOCK_SIZE * (by - 1) + 1;
        aEnd = aBegin + uiWA - 1;
        aStep = BLOCK_SIZE; bStep = BLOCK_SIZE * uiWB;
        bBegin = BLOCK_SIZE * (bx - 1) + 1;
        Ysub = single(0.0);
        b = bBegin;
        for a = aBegin : aStep : aEnd
            local_index = ty + (tx - 1) * BLOCK_SIZE;
            As(local_index) = A(a + uiWA * (ty - 1) + tx - 1);
            Bs(local_index) = B(b + uiWB * (ty - 1) + tx - 1);
            %acc barrier local
            for k = 1 : BLOCK_SIZE
                Ysub = Ysub + As(ty + (k - 1) * BLOCK_SIZE) * Bs(k + (tx - 1) * BLOCK_SIZE);
            end
            %acc barrier local
            b = b + bStep;
            end
        if g1 <= uiHA
            Y((g1 - 1) * global_size0 + g0) = Ysub;
        end
    end
%acc end
%acc end
end
%acc end

Figure B.2: Source code for the modified matrix multiplication benchmark.
```