U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Luís Alexandre Cubal dos Reis

Multitarget Compilation Techniques for Generating Efficient OpenCL Code from Matrix-oriented Computations

Supervisor: João Manuel Paiva Cardoso

Doctoral Program in Computer Science
of the Universities of Minho, Aveiro and Porto

universidade de aveiro

Universidade do Minho

U.PORTO

February, 2020

Abstract

Modern computer systems are often heterogeneous and parallel, featuring a mix of different compute units, including CPUs and GPUs. To achieve high performance, these compute units must be properly used, so programmers are often required to develop different program versions tuned for different target processors, written in low-level languages such as C or OpenCL. The process of tuning an existing C/OpenCL program to a new target often involves considerable restructuring of the code. However, manually performing this task is time-consuming and requires significant programming and computer architecture expertise. A possible approach is to have a single, high-level description of a program, and generate custom implementations according to the target processor. Not only can some high-level languages provide to the compiler more opportunities to generate optimized code (as implementation details are omitted from the original code), but also many programs in fields such as engineering and science are already initially written in these languages for modeling and prototyping, the MATLAB programming language being a well-known example of this. This thesis proposes techniques to generate efficient target-aware C and OpenCL code from high-level MATLAB functions annotated with simple and concise directives. These techniques include a large set of transformations to process MATLAB high-level idioms into efficient low-level code, automatically parallelize sequential code, optimize memory accesses and transfers, and adapt the generated code to a given target device. The proposed approach is validated with a compiler prototype based on the MATISSE compiler framework and considering a set of representative MATLAB benchmarks. One of the tested compiler techniques – efficient use of Shared Virtual Memory – allows the compiler to achieve geometric mean speedups on the generated code between 9% and 126%, depending on the target device, over no usage, and between 9% and 80% over a naive use of the technique. Another compiler technique – the cooperative schedule – allows for more efficient code generation on certain benchmarks (up to 38.8× faster). Furthermore, there are representative cases in which the proposed compiler prototype is able to generate code that outperforms manually written code.

Resumo

Os sistemas de computadores modernos são geralmente heterogéneos e paralelos, tendo uma combinação de várias unidades de computação diferentes, incluindo CPUs e GPUs. Para alcançar um desempenho elevado, estas unidades devem ser usadas adequadamente, pelo que frequentemente é necessário, para um dado programa, desenvolver várias versões adaptadas para cada processador-alvo, usando linguagens de programação de baixo nível como o C e o OpenCL. O processo de adaptar um programa C/OpenCL existente para um novo alvo geralmente envolve uma reestruturação significativa do código. No entanto, fazer esta reestruturação manualmente é demorado e requer conhecimentos avançados de programação e de arquitetura de computadores. Uma possível abordagem é ter uma única descrição de alto nível de um programa e gerar implementações adaptadas para cada processador-alvo. Não só podem as linguagens de alto nível fornecer aos compiladores mais opções para gerar código otimizado (visto que detalhes de implementação são omitidos do código original), como também é frequente programas em áreas de engenharia e ciência serem inicialmente modelados e prototipados nestas linguagens. Uma das principais linguagens usadas neste contexto é o MATLAB. Esta tese propõe técnicas para gerar código C e OpenCL eficiente e adaptado, com base em funções escritas numa linguagem de programação alto-nível baseada em matrizes chamada MATLAB, anotadas com diretivas simples e concisas. Estas técnicas incluem um conjunto substancial de transformações que processam idiomas de alto-nível do MATLAB para código de baixo-nível, automaticamente paralelizam código sequencial, otimizam acessos e transferências de memória e adaptam o código gerado a cada dispositivo-alvo. A abordagem proposta é validada com um protótipo de compilador baseado na infraestrutura de compilação MATISSE, tendo em conta um conjunto representativo de testes. Uma das técnicas de compilação testadas – uso eficiente de Memória Virtual Partilhada (Shared Virtual Memory) – permitiu ao compilador atingir melhorias de desempenho no código gerado em média geométrica entre 9% e 126%, dependendo do dispositivo-alvo, em relação a não utilizar memória partilhada, e entre 9% e 80%, em relação a um uso ingénuo da técnica. Outra das nossas otimizações – o mapeamento colaborativo de tarefas – permite uma geração de código mais eficiente em certos programas de referência (até 38.8× mais rápido). Nos nossos testes, determinamos que o código gerado pelo nosso protótipo de compilador tem melhor desempenho que o código OpenCL equivalente escrito manualmente em testes representativos.

Acknowledgments

This thesis has been funded by the *Fundação para a Ciência e a Tecnologia* (FCT), under the PhD grant PD/BD/105804/2014.

Furthermore, this work would not have been possible without the rest of the SPeCS group of the Faculty of Engineering of the University of Porto. I give special emphasis to João Manuel Paiva Cardoso, the advisor of this thesis, João Bispo, the original author of MATISSE, and Ricardo Nobre, who I have worked with to determine how to optimize GPU kernels.

I would also like to thank my family and friends for their support.

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

ALU	*Arithmetic Logic Unit*
AMD	*Advanced Micro Devices*
AOT	*Ahead-Of-Time*
API	*Application Programming Interface*
ASIP	*Application-Specific Instruction Processor*
AST	*Abstract Syntax Tree*
AVX	*Advanced Vector Extensions*
BLAS	*Basic Linear Algebra Subprograms*
CIR	*C Intermediate Representation*
CPU	*Central Processing Unit*
CSSA	*Conventional Static Single Assignment*
CUDA	*Compute Unified Device Architecture*
DSL	*Domain-Specific Language*
FFI	*Foreign Function Interface*
FPGA	*Field-Programmable Gate Array*
GCN	*Graphics Core Next*
GLSL	*OpenGL Shading Language*
GNU	*GNU's Not Unix!*
GPU	*Graphics Processing Unit*
HIP	*Heterogeneous-compute Interface for Portability*
HLS	*High-Level Synthesis*
HLSL	*High-Level Shading Language*
ILP	*Instruction-Level Parallelism*
IR	*Intermediate Representation*
JIT	*Just-in-time*
LoC	*Lines of code*
MATISSE	*A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms*
MATLAB	*Matrix Laboratory*
MEGHA	*MATLAB Execution on GPU based Heterogenous Architecture*
MPI	*Message Passing Interface*
NaN	*Not a Number*
NUMA	*Non-Uniform Memory Access*

OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
SIMD	*Single Instruction, Multiple Data*
SIMT	*Single Instruction, Multiple Threads*
SMT	*Satisfiability Modulo Theories*
SoC	*Systems on a Chip*
SPeCS	*Special-Purpose Computing Systems, languages and tools*
SPIR	*Standard Portable Intermediate Representation*
SSA	*Static Single Assignment*
SVM	*Shared Virtual Memory*

1

Introduction

Contents

Modern computing systems, from smartphones to supercomputers, are increasingly heterogeneous, with a mix of Central Processing Units (CPUs), Graphics Processing Units (GPUs) and occasionally other hardware accelerators. GPUs are now available on common embedded Systems on a Chip (SoC) [ARM19], while FPGA vendors are aiming for the datacenter [Int19].

Taking advantage of the processing power of these systems remains challenging, as programmers must deal with the different characteristics of each processing unit, and often need to develop different program variations for each combination of processing units using low-level languages such as C and OpenCL [Khr15].

As an alternative, high-level languages can be used to describe program specifications, without platform-specific optimizations, and use fully automated or user-guided approaches to generate efficient customized implementations that target specific platforms, such as GPUs. High-level languages are often considered inefficient, but the lack of low-level implementation details also allows for a larger set of optimization strategies to be applied. One of these high-level languages is MATLAB [Mat13a], a matrix-oriented programming language that is widely used for scientific, engineering and financial models.

This thesis proposes techniques for the generation of efficient C and OpenCL code from MATLAB models annotated with simple and concise directives. Additionally, the proposed techniques are validated with a compiler prototype based on the MATISSE compiler framework [BPN+13].

1.1 Context and Motivation

Processor technology has improved substantially over the last years and, with it, CPU performance has improved as well. However, as far back as 2004, indications were that "free" single-threaded performance gains (i.e., performance improvements that require no program modifications) were over [Sut04]. In 2012, a page on Intel's developer zone confirmed this trend, mentioning that "No one expects a leap forward in processors' core execution engines, which are already at the edge of the manufacturing envelope." [Int12]

Instead, to take advantage of modern CPUs, programs must divide their workloads across multiple cores – units that execute simultaneously (i.e., in *parallel*). Individually, each core may not be getting much faster, but with multicore CPUs, applications can execute more work at the same time.

Still, pure CPUs computing is not always the best choice for optimal performance. One of the most common alternatives is to use a combination of CPUs and GPUs, as this approach can enable significant performance speedups (in some cases of more than 10× [NVI18a]). These hybrid systems containing multiple types of processors are known as *heterogeneous systems*.

GPUs are also highly parallel processors, that require program modifications to be properly used. In order to target GPU devices, programmers typically write code using programming languages or APIs specifically designed for that purpose.

Low-level languages such as C and OpenCL [Khr15] can enable programmers to write high-performance parallel applications. However, these languages tend to be difficult to use compared to high-level languages, such as MATLAB. Moreover, the implementation details specified in C/OpenCL can be overly restrictive to target the particular performance requirements of certain devices, so when porting programs to new devices, they may perform worse unless portions of their code are rewritten to apply manual

device-specific optimizations. In other words, low-level languages are not *performance portable*.

In contrast, high-level languages are very expressive in terms of program semantics, but express relatively few implementation concerns, giving optimizers ample space to determine how to properly optimize the code.

MATLAB [Mat13a] is a high-level matrix oriented programming that is very flexible, and widely used by domain specialists of multiple domains. Many algorithms are first prototyped and validated in high-level languages, such as MATLAB, and later manually rewritten in low-level languages for optimization. Doing so is costly and error-prone, so automatic tools to perform this conversion are highly desirable.

This thesis demonstrates techniques to automatically generate C/OpenCL code from MATLAB code, allowing MATLAB programmers to tap into the performance benefits of these languages, while avoiding the overhead of the MATLAB runtime and enabling the use of this language for heterogeneous computing. Furthermore, this thesis demonstrates how high-level languages such as MATLAB can be used to facilitate the process of tuning programs to specific devices and assist in achieving *performance portability*.

1.2 Thesis Goals

The goals of this thesis are to:

- Research and develop techniques to facilitate the development of parallel software suitable for heterogeneous systems with GPUs;
- Research and develop target-aware techniques to optimize code for GPUs, based on representative benchmarks;
- Demonstrate that the proposed techniques are feasible to implement by developing a compiler prototype;

1.3 Contributions

The work of this thesis lead to the following contributions:

- A directive system for MATLAB that is easy to understand, simple to use, and enables efficient but flexible OpenCL code generation (see Section 4.1.2);
- A set of optimizations to improve the performance of the generated C and OpenCL code, notably in the form of usage of Shared Virtual Memory (SVM) and work schedules (see Sections 5.8, 5.5 and 5.6);
- Techniques to determine when to apply those optimizations, based on static program properties and the target device (see Sections 5.8, 5.6 and 6.9);
- A compiler prototype that integrates the proposed techniques and can be used to develop and test new optimizations (see Section 4.2);
- Experimental results to evaluate the impact of the approaches proposed in this thesis, using a set of representative benchmarks (see Chapter 6).

Our approach has been mostly designed/evaluated for multi-core CPUs and GPUs.

1.4 Outline of the Thesis

The remainder of this thesis is structured as follows. Chapter 2 describes relevant background necessary to understand the work presented in this thesis. Chapter 3 describes work that is related to the subject of this thesis. Chapter 4 explains the compiler prototype, namely the programming model it targets, the general internal structure of the compiler, and how its correctness we validated. Chapter 5 describes the optimizations that were studied/developed during the course of this thesis. Chapter 6 presents the experimental methodology and results for validation and the evaluation of the proposed optimizations. Finally, Chapter 7 presents concluding remarks and describes possible future work.

2

Background

Contents

This chapter briefly describes the programming languages, third part tools, and computing architectures that are relevant to this thesis, the performance concerns that have to be addressed and optimizations that can be applied in order to achieve parallel program performance, and some state-of-the-art optimizations.

2.1 The MATLAB Programming Language

MATLAB [Mat13a], an acronym for Matrix Laboratory [Mat14a], is a proprietary high-level matrix-oriented programming language developed by MathWorks [Shu16a]. MATLAB is used in a variety of fields, including engineering and science, for numeric computations, simulations, models and applications. Due to the wide adoption of MATLAB, free software environments similar to MATLAB have been proposed. One example is GNU Octave [Oct14a], which provides a programming language very similar to MATLAB [Oct14b].

MATLAB was designed to operate on matrices, so all MATLAB variables are matrices. Even scalars are matrices of size 1×1. 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, given two scalar inputs N and X, computes a matrix of size $N \times N$ where all elements in the diagonal are double-precision 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 a constant value X on the diagonal. Note that most matrix allocation functions, such as `eye`, allocate square matrices when given a single scalar argument.

With the exception of `&&` (*logical and*) and `||` (*logical or*), all MATLAB operators support matrices. For instance, `A == B` performs an element-wise comparison of the elements of `A` and `B`, as seen in Figure 2.2.

```
1 eye(3) == zeros(3)
```

(a) A MATLAB expression that compares two matrices.

```
1 0     1     1
2 1     0     1
3 1     1     0
```

(b) The output of the comparison above.

Figure 2.2: Demonstration of a MATLAB operator (==) operating on whole matrices, as opposed to scalars.

MATLAB also supports conventional control-flow mechanisms, such as `if`, `while` and `for` loops. MATLAB `for` loops are different from the ones seen in languages such as C. Figure 2.3 shows how the MATLAB `eye` function could be implemented for square matrices. At 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 assign a value to a single position of the matrix. Note that it uses parenthesis to access the matrix position, like function calls use for the list of arguments.

```
1 function y = manual_eye(N)
2     y = zeros(N); % Allocate NxN matrix with all elements initialized to
      zero.
3     for i = 1:N
4         y(i, i) = 1;
5     end
6 end
```

Figure 2.3: A MATLAB expression that computes a matrix with 1 in all positions of the diagonal, and 0 in all other positions.

Identifiers can refer to both variables and functions, so distinguishing matrix accesses from function calls can be difficult. Fortunately, MATLAB imposes an important restriction on identifiers: in any given function, an identifier that may be used to refer to a variable can not refer to a function. Figure 2.4 shows this principle. Even when `use_var` is false, MATLAB determines that `fft` is a variable, not a function, and refuses to call the `fft` function on line 8. Due to this restriction, it is possible to distinguish between variables and functions using static analysis.

```
1 function y = dynfft(use_var, b)
2    if use_var,
3        fft = [1 2; 3 4];
4    end
5
6    % MATLAB error if use_var is false (Undefined function or variable "fft")
7    % even though 'fft' is an existent function.
8    y = fft(b);
9 end
```

Figure 2.4: A MATLAB function demonstrating that determining whether an identifier is a variable or a function can be performed statically.

Note that using a loop-based, scalar-oriented code as seen in Figure 2.3 tends to be significantly slower than the vectorized equivalent [Mat17b]. A vectorized version of the function in Figure 2.3 is shown in Figure 2.5. Line 3 shows a vectorized matrix access, where the value 1 is written to multiple positions of matrix y in a single line of code. The expression `1:N+1:N*N` outputs the indices to be written (all indices starting at 1,

```
1 function y = manual_eye(N)
2     y = zeros(N); % Allocate NxN matrix
3     y(1:N+1:N*N) = 1;
4 end
```

Figure 2.5: A vectorized version of the function in Figure 2.3.

incremented by `N+1` at a time, and ending at N^2). Although y is a two-dimensional matrix, it is possible to access its positions with a single index, in *column-major* order[1]. Figure 2.6 demonstrates how this works.

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Figure 2.6: Matrix indices in the column-major order starting at 1, as used by MATLAB.

When accessing matrix positions, it is possible to use indexes and ranges relative to the size of the matrix. Figure 2.7 shows the use of the `end` keyword in line 2, which refers to the index of the last valid value of a dimension. In this context, `end` is 9 because that is the size of the dimension it is in. In line 3, `y(1, :)` is equivalent to `y(1, 1:end)`.

```
1 y = zeros(10, 9);
2 A = y(1, 2:end-1);
3 B = y(1, :);
```

Figure 2.7: A MATLAB segment of code showing how to use the end keyword.

MATLAB single comments start with `%` and block comments are delimited by `%{` and `%}`, though some restrictions apply. Block comments may be nested.

Figure 2.8 shows an example of a MATLAB function. Note that each MATLAB function may receive zero or more inputs, and may produce zero or more outputs. Although each MATLAB file may have one or more functions, only the first function in each file may be called from other files. This function is expected to have the same name as the file where it is declared[2]. The remaining functions in any file can only be called by other functions in the same file.

Function arguments are passed by value (i.e., copied), meaning that changes to matrix arguments within a function are not visible to the caller.

[1] In this particular case, the function would be identical if MATLAB used row-major order, because the identity matrix is symmetric.

[2] If the names are not the same, MATLAB uses the name of the file as the name of the function, and ignores the name in the function declaration.

```
function [a, b] = test(c, d)
    a = c;
    b = d;
end
```

Figure 2.8: MATLAB function declaration example.

MATLAB also supports *scripts*, which are files without any user-defined functions (i.e., with MATLAB statements inserted directly in the script).

In MATLAB, variables and functions/scripts may have the same name. Moreover, there can be multiple functions with the same name (in different files). When MATLAB writes to a variable, it creates a new variable with that name if one does not already exist, regardless of any functions that may already have that name (see the discussion of Figure 2.4). For reads, the MATLAB call syntax and the MATLAB array access syntax are the same, with parenthesis being used to denote the arguments or indices. Even when no parenthesis are used, the referenced name can still be a function call with an empty argument list. When MATLAB encounters a name, it resolves in the following (simplified) order:

1. If there exists a variable in the current function with that name, use that variable;
2. Otherwise, if there exists a function in the same file with that name, call that function;
3. Otherwise, if there exists a MATLAB file with that name, call the main (first) function of that file;
4. Finally, if there exists a built-in MATLAB function with that name, call that function.

MATLAB's greatest strength comes from the extensive and highly optimized set of built-in functions for specific operations (such as efficient matrix operations) 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 a specific domain first.

At the same time, MATLAB has two significant issues: performance and portability. Poor performance of MATLAB programs can often be attributed to the use of inefficient MATLAB idioms, such as the use of loop-based, scalar-oriented code, as previously mentioned, or the incremental dynamic resizing of a data structure [Mat17a]. In these cases, modifying the code to rely on more efficient idioms can substantially improve performance. Additionally, program performance can be improved by converting the MATLAB code to C using an automatic tool such as MATLAB Coder [Mat18e]. In order to mitigate this performance issue, MATLAB has employed the use of JIT (*Just-in-time*) compilation since version R13, released in 2002, and have since then improved performance [Shu16b]. 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 [Mat14b]). Once again, porting the code to C using MATLAB Coder or another equivalent tool can mitigate this issue.

2.2 The Z3 SMT Solver

Z3 [DMB08] is a Satisfiability Modulo Theories (SMT) prover developed by Microsoft Research. Using Z3, programmers may define *assertions* (i.e., boolean expressions) and Z3 *checks* whether those assertions are *satisfiable*, that is, whether there is a *model* (i.e., an example of values) that meets all restrictions defined in the assertions.

Z3 supports a programming language called SMT-LIB2 [The18], and also features libraries for multiple programming languages, including Java. Figure 2.9 shows an example of a Z3 program written with SMT-LIB2 syntax. Lines 1 and 2 define two values x and y that are integers (in the mathematical sense, not limited to any specific ranges). Line 3 adds the assertion that $x > y+1$, meaning that any valid model must respect that restriction. Line 4 instructs Z3 to check whether the given assertions can be satisfied. The `check-sat` operation has three possible outcomes: *satisfiable* (there is at least one valid model), *unsatisfiable* (there is no possible valid model), or *unknown* (Z3 was unable to determine whether any valid models exist).

```
1 (declare-const x Int)
2 (declare-const y Int)
3 (assert (> x (+ y 1)))
4 (check-sat)
```

Figure 2.9: Z3 program that computes whether two integers x and y can be defined such that $x > y + 1$.

Z3 also supports other use-cases, such as optimizing certain formulas (e.g., minimization of an expression), but these features are not used by MATISSE, so we do not discuss them in this thesis.

2.3 Parallel Devices

Modern software requires parallel programming in order to achieve high performance, as improvements in sequential execution have slowed down due to thermal, power and current leakage issues [Sut04].

There are multiple types of parallel processors. This section describes the ones that are targeted in this thesis.

2.3.1 Multi-core CPUs

Modern CPUs support multiple forms of parallel processing, including multi-threaded execution and support for Single Instruction Multiple Data (SIMD) instructions [Int17a].

CPU parallel programming is typically based on the concept of *threads*. Multiple threads execute parts of a program simultaneously in CPU cores. Each thread has its own execution state (including a *program counter* and a *stack*) and can generally run independently of other threads – multiple ones can even co-exist as part of different processes. In some modern processors, each *physical core* can execute more than one thread to improve throughput [Int17c].

Multi-threaded programming enables higher performance, though at a cost. For instance, some CPUs are unable to reach their maximum supported clock frequencies

when running multiple cores simultaneously [Int17d].

Another approach for parallelism is the use of SIMD instructions, such as Intel AVX [Lom11] or ARM Neon [ARM17b]. SIMD instructions allow performing the same operation (such as addition or multiplication) to multiple pieces of data in a single instruction, while still preserving the sequential nature of program execution. Some SIMD instructions also cause the processor to run at lower clock frequencies. For instance, Intel notes that on the Intel Xeon E5-2600 v3, vector instructions at least 128 bit wide require additional voltage, which may reduce clock frequency, but even so recommends AVX/AVX2 as they consider that the benefits "far outweigh the issues due to drop in core frequency" [Kar14].

Note that multi-threading and SIMD instructions are not mutually exclusive. A combination of both can be used for optimal performance.

2.3.2 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) are processors designed for massively parallelizable tasks. GPUs are used in heterogeneous systems, where compute-intensive portions of the code are offloaded to these devices, while the remaining sections execute on the CPU [NVI17b].

Different GPUs have different architectures, but some of the more common GPUs are NVIDIA with its Single Instruction Multiple Thread (SIMT) model [NVI17d, Section 4.1]. Whereas CPUs generally feature cores in the single or low double digits, these GPUs can include thousands of smaller, individually slower, units capable of executing threads. NVIDIA's GPUs manage groups of 32 threads called *warps*. Threads in the same warp can branch independently, but each warp can only execute one instruction at a time (see Subsection 2.5.4). Warps, in turn, are grouped in *blocks* (equivalent to the concept of work-groups in OpenCL), which are distributed by multiple *Compute Units* (the closest equivalent to the CPU concept of core).

AMD Graphics Core Next (GCN) [Adv18] and ARM Bifrost GPU [Har18] architectures are similar. Threads are also grouped in wavefronts of 64 threads [Adv15, Section 1.5] or quads of 4 threads [ARM17a, Section 9.6], respectively, in which threads of the same group can branch independently, but only one path can be executed at a time.

To improve performance, many GPUs have different memory spaces, with different characteristics. For instance, NVIDIA describes the following memory regions usable on their GPUs, among others:

- Global memory is shared by the entire GPU device (but it is distinct across different GPUs) and can be accessed by any running thread, as well as the host CPU.
- Constant memory resides in the constant memory space and can not be modified. It can be accessed by any running thread, as well as the host CPU.
- Shared memory resides in a memory that is private to each block and, as such, can only be accessed by threads within that block.
- Register memory is private to each thread and accessing it consumes zero extra clock cycles per instruction [NVI17a, Section 9.2.6].
- Local memory resides in the same memory location as global memory, but it is accessible only by the thread that declared the variable/buffer. Local memory is

used for thread-private data that can not be placed in register memory (e.g., due to register spilling);

Also of note is managed memory, that can be referenced by both the device and the host CPU and is automatically migrated as needed. It is used in the context of unified memory [Har13].

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]. As these languages were not specifically designed for GPU programming, new languages were developed to simplify this use-case, notably CUDA [NVI14b] and OpenCL (see Section 2.4).

2.4 OpenCL

OpenCL [Khr15], Open Computing Language, is a royalty-free standard developed and maintained by Khronos and originally proposed by Apple [The08]. OpenCL features an API and a C-like language for general-purpose parallel programming across multiple types of processors, including CPUs, GPUs [NVI17c, Adv17] and FPGAs [Int17b, Xil17].

Figure 2.10 shows a simple program written in OpenCL that adds two vectors. The first line declares the function, named `float_add`, with three arguments - all global memory pointers. When a kernel is called, the programmer indicates the number of times it should be executed. The kernel can see the index of its execution using the `get_global_id` function, as seen in line 2. In this example, the result is computed and stored in line 4.

```
1 kernel void float_add(global float* buffer1, global float* buffer2, global
      float* result) {
2     size_t index = get_global_id(0);
3
4     result[index] = buffer1[index] + buffer2[index];
5 }
```

Figure 2.10: OpenCL kernel that adds two vectors.

Figure 2.11 shows the C code of a program that uses the OpenCL API. This example is used throughout this section.

Using OpenCL, an application running on the *host* (the CPU) executes *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, 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, or from binary formats, such as SPIR-V [The17], not shown in this example. Line 33 obtains a specific kernel from the 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

```c
#include <CL/cl.h>
#include <string.h>
#define ELEMENTS 1024

int main() {
    cl_platform_id platform;
    cl_device_id device;
    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);
    int global_size = ELEMENTS;

    clGetPlatformIDs(1, &platform, NULL);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
    cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL,
        NULL);
    cl_command_queue 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);
    // buffer assignment omitted

    cl_program program = clCreateProgramWithSource(context, 1, &program_src,
        &src_length, NULL);
    clBuildProgram(program, 1, &device, "", NULL, NULL);
    cl_kernel 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 the variable named 'buffer'
    return 0;
}
```

Figure 2.11: C code to call the OpenCL kernel in Figure 2.10. Error detection and resource cleanup have been omitted for brevity.

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). 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 the exploited parallelism. The primary form of parallelism for OpenCL is *data parallelism*. 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.

Each device has one or more *compute units*. Each simultaneous kernel execution is a *work-item*. Each *work-group*, consisting of one or more work-items, is executed on a single compute unit. The sizes of work-items and work-groups can be specified across multiple dimensions as the *range*. When launching OpenCL kernels, programmers specify 2 sizes: 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 determines which value to use. 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. OpenCL kernels can be organized in multiple dimensions. For instance, to execute 16 work-items, a programmer may specify a global size of (16), (2, 8), (4, 4, 1) or others.

Some OpenCL kernels may rely on specific local sizes to execute correctly. When this is the case, the kernel attribute `reqd__group_size` [Khr16, p. 48] can be used. If any value is passed to the kernel enqueue API call other than the one specified in the attribute, OpenCL refuses to execute the kernel and return an error code.

Starting in OpenCL 2.1 (or OpenCL 2.0 with the `cl_khr_subgroups` extension), work-groups are further divided into sub-groups, consisting of an implementation-defined number of work-items [Khr15, p. 22].

Synchronization mechanisms during kernel execution in OpenCL are very limited. It is possible 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, mechanisms to synchronize kernels in different work-groups are very limited (practically non-existent).

OpenCL defines 4 distinct memory regions for devices: private, local, constant and global [Khr15, p. 38]. Private memory is accessible only by a given work-item and must not be accessed by other work-items, as each work-item has its own private memory. Local memory is accessible only by work-items on a given work-group and is not visible by other work-groups, as each work-group has its own local memory. Constant memory is initialized by the host and remains constant during the kernel execution. Global memory is accessible 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. In addition to this, OpenCL has *image objects*, which can be used to access data in textures [Khr15, p. 128].

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 they occurred in. It is possible to ensure work-items read the most recent version of memory using memory fences, which can be specified as part

of the `barrier` function (renamed to `work_group_barrier` in OpenCL 2.0, though the old name is still supported) or using the separate `mem_fence` function (which seems to have been removed in OpenCL 2.0). A local memory fence ensures that work-items local memory reads provide the correct value if it was not modified since the memory fence. Global memory fences do the same for global memory.

Since version 2.0, OpenCL supports Shared Virtual Memory (SVM) [Sof14]. This feature provides a single address space for the host and the devices (as global memory). Some of the use-cases for SVM include reusing the same data structures (including pointers) for both host and device and avoiding the need for explicit data transfers. The OpenCL specification only requires support for *coarse-grained buffers* [Khr15, p. 174], in which buffers are allocated with `clSVMAlloc` and the host can only update memory that has been explicitly mapped to the host (e.g., with `clEnqueueSVMMap`). However, OpenCL implementations can optionally implement *fine-grained sharing* as well, in which memory can be read and modified by both the host and the device without the need for any mapping operations. Note that for the host and the device to concurrently update the same locations, atomic operations must be used, and support for atomics is not required by the standard. OpenCL further distinguishes between *fine-grained buffer SVM* and *fine-grained system SVM*. With fine-grained buffers, OpenCL kernels can only use SVM memory that has been allocated with `clSVMAlloc`, whereas with fine-grained system SVM any host memory can be used in the device, without requiring any specific allocation function.

OpenCL includes a built-in profiler [The09] 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.

One of the most widely used programming languages for GPGPU computing is NVIDIA's CUDA [NVI14b]. Since many GPGPU programmers are familiar with CUDA terminology, this thesis includes Table 2.1, which shows a list of CUDA and OpenCL terms, and how they relate to each other. Most CUDA concepts have an OpenCL equivalent and vice-versa. This happens because OpenCL and CUDA have very similar programming models. Note that *local memory* has a different meaning in CUDA and OpenCL. In CUDA, it refers to memory that is private to each thread/work-item, whereas OpenCL local memory refers to memory that is private to each block/work-group.

2.5 Target-aware Performance Characteristics

When targeting accelerator devices, there are a number of considerations that can significantly impact the performance of programs. These are generally addressed by programmers, but several of these concerns could be automatically optimized by compilers.

This section describes several of these issues, the devices they are relevant to, as well as their cause and impact.

[3]The size of a sub-group is implementation-defined, whereas warps always consist of 32 threads. Sub-groups are only part of OpenCL core since OpenCL 2.1. AMD refers to their equivalent in GCN GPUs as *wavefronts* [Adv12, p. 1-2].

[4]Used for arrays that are indexed with non-constant values and for register spilling [NVI17d, Section 5.3.2]. Resides in device memory.

[5]In CUDA, textures are always read-only, whereas in OpenCL read/write and write-only textures exist.

Table 2.1: CUDA concepts and equivalent OpenCL terminology.

CUDA	OpenCL
Thread	Work-item
Warp	Sub-group[3]
Block	Work-group
Grid	N/A
Grid size	Global size
Thread Index	Local ID
Block Index	Group ID
`blockIdx * blockDim + threadIdx`	Global ID
Global memory (`__device__`)	Global memory
Shared memory	Local memory
Constant memory	Constant memory
Local memory[4]	Private memory
Texture/Surface memory[5]	Texture memory
Unified Memory (CUDA 6)	Fine-grained buffer SVM

2.5.1 Memory Coalescing

Memory coalescing refers to the property of devices (e.g., GPUs) to combine multiple global memory accesses into fewer operations. NVIDIA describes memory coalescing as "perhaps the single most important performance consideration in programming for CUDA-capable GPU architectures" [NVI17a, Section 9.2.1].

On NVIDIA GPUs, memory coalescing occurs at the level of the warp. Starting with devices with compute capability 2.x[6], memory accesses to the same cache line by threads in the same warp result in a single combined memory access. In contrast, when threads in a warp access memory positions in different cache lines, each accessed line requires its own memory access. Older NVIDIA GPUs have stricter requirements for memory coalescing. As there are 32 threads per warp, non-coalesced accesses can cause up to 32× more access operations than the coalesced equivalent.

Figure 2.12 shows an OpenCL kernel with three different access patterns. On current NVIDIA GPUs, the L1 cache has a line size of 128 bytes. As each single-precision float has a size of 4 bytes, each cache line has $\frac{128}{4} = 32$ elements. Therefore, if the L1 is enabled, a single memory operation can store the entire data of `buffer1` for each warp. As for `buffer2`, due to the stride, different threads of the same warp are accessing elements in 2 cache lines, so twice as many memory operations are required to store the data of `buffer2`. The `buffer3` variable is accessed with a stride of 32, so each thread is accessing a different cache line. This is the worst-case scenario for memory coalescing of floats, as each warp accesses its own cache line, so the memory accesses are *serialized*.

Note that an access of stride 1 (similar to `buffer1`) can still require data from more than one L1 cache line. An example of this can happen for double-precision buffers, as each value takes 8 bytes and, therefore, each cache line contains only $\frac{128}{8} = 16$ elements. In this case, two L1 cache lines are used.

Similarly, on Intel GPUs, global memory accesses to the same L3 cache line are combined into a single memory operation [Int15].

[6] NVIDIA uses the term Compute Capability to indicate the available features of each GPU [NVI14a]. Compute Capability 2.0 was introduced with the Fermi GPU architecture, in 2010.

```
kernel void coalesced_access(global float* buffer1, global float* buffer2,
    global float* buffer3)
{
    int index = get_global_id(0);

    buffer1[index] = 0;
    buffer2[index * 2] = 0;
    buffer3[index * 32] = 0;
}
```

Figure 2.12: OpenCL program with various memory access patterns.

AMD GPUs can also coalesce memory accesses, albeit somewhat differently. For instance, on AMD's Southern Islands architecture, memory reads of 64-bit data types are not coalesced [Adv15, p. 2-33] and neither are memory writes [Adv15, p. 2-8].

2.5.2 Local Memory

The use of local memory (shared memory in CUDA terminology) can lead to performance improvements on some devices. GPUs typically have more than one type of memory, with different memory access times. On many of these devices, local memory is substantially faster than global memory. Note, however, that some GPUs do not have local memory (e.g., ARM's Mali [ARM17a, Section 7.3.2]).

On AMD GCN [Adv15, p. 2-9] and NVIDIA [NVI17a, Section 9.2.2.1] GPUs, local memory accesses have significantly more bandwidth than global memory. On Intel GPUs, local memory is allocated from the L3 cache [Int15].

However, the profitability of using local memory on GPUs depends on various factors. For instance, on NVIDIA [NVI17a, Section 9.2.2.1], AMD GCN [Adv15, p. 2-9] and Intel [Int15] GPUs, shared/local memory is organized on memory modules called *banks* and each access is processed through these banks. Broadly speaking, when multiple work-items in the same group of threads (a warp, in NVIDIA's case) attempt to access local memory in the same bank, the accesses are serialized.

Several authors have studied the impact of local memory on kernel performance. For instance, Brodtkorb et al. [BHS13] recommend always using the fastest available memory when attempting to optimize memory-bound programs. Intuitively they advise that data should be kept preferably in registers, followed by the local memory when that is not possible and finally the global memory. In contrast, Fang et al. [FSV14] argue that the profitability of local memory can sometimes be counter-intuitive, with three examples:

- Reusing data within a work group is not sufficient for local memory to be profitable, since GPUs have caches that do not have the data transfer overhead associated with local memory. Therefore, global memory may be faster than local memory.
- Additionally, even kernels that do not have data reuse might benefit from local memory. For instance, loading global memory data to local memory could change the global memory access order and enable memory coalescing, thus improving the performance.

- Finally, the fact that local memory is allocated in the main memory of CPU implementations suggests that it should not be used on those systems. However, in some cases, local memory can still be profitable due to better cache usage and specialized optimizations.

Finally, Shen et al. [SFSV13] studied the impact of local memory on CPU implementations of OpenCL, but concluded that it causes slowdowns. They explain that the use of local memory implies copying data from the global memory to the local memory and the addition of a memory barrier, which are both expensive operations.

2.5.3 Texture Memory

Texture memory [NVI17d, Section 3.2.11.1] is an alternative method to store/access data. Although on certain devices (e.g., NVIDIA GPUs [NVI17a, Section 5.3.2]) textures reside in the same off-chip memory as global memory, texture memory can still exhibit better performance due to differences in caching behavior. For instance, on NVIDIA GPUs, the texture cache is optimized for 2D locality. Additionally, operations to access textures can include clamping, filtering and normalization.

Du et al. [DWL+12] measured the impact of using texture memory. In the case of OpenCL, copying data to texture memory and using the textures from the kernel can lead to performance improvements. Although they did not test texture memory for the SGEMM algorithm, they did observe a large performance improvement on the DGEMM algorithm. One problem with using texture memory in OpenCL is that it is necessary to copy the data if it is not already in texture memory, which may imply noticeable overhead. In contrast, CUDA programs can map 1D textures to buffers, avoiding this overhead in some cases.

According to Brodtkorb et al. [BHS13], the use of texture memory can improve performance but only in rare cases, as the L1 cache is still faster than the texture memory subsystem.

2.5.4 Branch Divergence

Branch divergence [NVI18b, Section 9.5], or control-flow divergence, is a performance issue that arises on some devices (notably GPUs), when two work-items in the same warp/wavefront have divergent behavior in control flow instructions. This can happen on `if` statements (when some work-items enter the `if` while others enter the `else` case or skip to the end of the statement), `switch` statements (when different work-items enter different `cases`) and loops (if the number of iterations is not the same for all work-items). In some cases, divergent control-flow can impact performance so negatively that offloading computations to the GPU is no longer profitable [Adv15, Section 3.7.1].

Figure 2.13 shows an OpenCL kernel where divergent control-flow may occur. If some values in the `in` buffer are positive and others are not, then only some work-items will enter the `if` statement. In this case, it is not possible to statically determine whether the control-flow is divergent. For instance, it is possible that all elements in the `in` buffer are positive, in which case there is no divergence.

On NVIDIA pre-Volta [NVI17a, Section 12.1] and AMD AMD GCN [Adv15, Section 1.5] GPUs, however, work-items in the same warp/wavefront share the same program counter, so if some work-items enter the `if` statement, the work-items that do not have to wait until the execution of the `if` is over. Similarly, the time to execute a loop depends on the *maximum* number of iterations a thread in the warp/wavefront

```
1 kernel void foo(global float* in, global float* out) {
2     int index = get_global_id(0);
3     if (in[index] > 0) {
4         out[index] += 1;
5     }
6 }
```

Figure 2.13: OpenCL example with possible branch divergence.

executes. In NVIDIA Volta [DGHS17], each work-item has its own program counter, but each warp can still only execute code from one branch case at a time, so divergence is still an issue.

Branch divergence is strongly connected to the concept of *data divergence*, in which the same variable or expression has different values for different threads in the same warp. In the example in Figure 2.13, `index` and possibly `in[index]` are divergent, while `in` and `out` are not. Data divergence can not always be determined at compile-time (e.g., `in[index]` is not data divergent if all elements of `in` have the same value), but static analysis can still be used to some extent. For instance, Sampaio et al. [SSCP14] use static divergence analysis to improve a register spiller for GPUs and achieve speedups of 26.21%.

CPU architectures are generally not based on the concept of warp, so divergent control-flow should not be an issue. However, Shen et al. [SFSV13] measured the impact of branches that depend on the work-item ID (and therefore can diverge) on the Intel CPU implementation of OpenCL and found that they can negatively impact performance. Specifically, they studied the interactions between Intel's auto-vectorization feature (see Subsection 2.5.5) and divergent branches. When the compiler is unable to determine that all work-items in a thread will enter the same regions, it must mask the SIMD operations to ensure correctness. They conclude that the presence of these branches can cause the auto-vectorized version to perform worse than the version without auto-vectorization.

2.5.5 Vector Types

OpenCL [Khr16, Section 6.1.2] and CUDA [NVI17a, Section B.3] both includes built-in vector types and operations. Using vectors, programmers can perform the same computation on multiple values in a single operation.

Figure 2.14 shows an OpenCL program that adds two vectors using explicit vector types. On certain devices, such as CPUs, the vector instructions are compiled to SIMD instructions. AMD recommends using vector types to more efficiently use the vector Arithmetic Logic Units (ALUs) of CPUs [Adv15, p. 1-28].

However, explicitly using vector types does not guarantee performance improvements. Intel's OpenCL compiler features an optional auto-vectorizer optimization, which packs multiple work-items (the number of work-items depends on the width of the SIMD unit) into a single thread with SIMD operations. According to Shen et al. [SFSV13], this feature leads to speedups of 2 times in comparison to the version without the automatic vectorization running on the same devices. The auto-vectorizer does not always lead to speedups, though. The authors note that for a K-means benchmark without swapping, the vectorized version takes twice as much time as the non-vectorized version.

```
kernel void vectoradd_vec4(global float8* out, global float8* a, global float8* b) {
    int index = get_global_id(0);
    out[index] = a[index] + b[index];
}
```

Figure 2.14: OpenCL example computing the sum of two vectors, using explicit vector types.

This means that the potential for performance improvements for the auto-vectorizer depends on the order of the memory accesses. The swapped version performed better than the non-swapped version because the former had a row-major access order and the later had a column-major order. Auto-vectorization also caused slowdowns in the PathFinder benchmark [MS09]. Unsurprisingly, the authors recommend turning on and off auto-vectorization and measuring which version performs best. Manual vectorization is in many cases an option to consider. However, Intel advises against manually vectorizing kernels on the Xeon Phi [Int14].

Pennycook et al. [PHW+13] acknowledge that vector data types can help with optimization, but downplay this due to the existence of Intel's auto-vectorizer and the fact that AMD is capable of generating quality code for its VLIW architectures. The authors expect auto-vectorization to further improve in the future, and as such do not consider vector types to be very important. However, they note that Intel's OpenCL implementation is unable to auto-vectorize their kernels.

On AMD GPUs, vectorization is generally not desired [Adv15, Section 2.9] and may negatively impact performance. For instance, reads to 64-bit data types are not coalesced.

However, on some GPUs, vector types can still lead to performance improvements. For instance, Luitjens [Lui17] notes that on NVIDIA GPUs vectorizing memory accesses can lead to improved memory read bandwidth.

According to ARM, on Bifrost GPUs, vector types for 8-bit and 16-bit types should be used, but scalars should be used for larger data types [ARM17a, Section 9.6]. Similarly, on some NVIDIA GPUs, such as Tesla P100, using vector types can double the throughput on half-precision arithmetic [Har16].

2.5.6 Floating-Point Precision

Some programs that rely on floating-point operations can tolerate inaccuracies by a certain margin of error. In these cases, programmers may trade accuracy for performance [BHS13].

One form of improving performance by reducing accuracy is by using floating-point data types with less precision, such as single-precision or half-precision floating-point types instead of double-precision types. There are multiple reasons for the performance differences between different floating-point data types across devices:

- Certain GPUs do not coalesce memory reads to 64-bit data types, such as double [Adv15, p. 2-33], which can significantly impact performance (see Subsection 2.5.1).

- Smaller data types use less memory, and as such more data of smaller types can be simultaneously loaded with the same memory bandwidth [Har15].

- Some instructions have higher throughput when lower precision types are used. For instance, on an NVIDIA Tesla P100, half-precision arithmetic has twice the throughput of single-precision arithmetic [Har16].

- Some GPUs are capable of computing fewer double- or half-precision operations per clock cycle than single-precision ones. For instance, some GPUs can perform 128 32-bit floating-point add operations per cycle per multiprocessor[7], but only 4 for double-precision and 2 for half-precision [NVI17d, Section 5.4.1].

- Operations with half-precision floats are more likely to result in subnormal numbers [Har16].

Note that support for double- and half-precision floating-point types [Khr16, p. 161] is not required by the OpenCL standard. As such, using these types can impact portability, as not all devices support these types. For instance, Intel CPUs only introduced instructions to convert single-precision floats to half-precision floats and vice-versa in the 3rd generation of Intel Core CPUs [Kon12].

OpenCL exposes *native* and *half* versions of certain functions (e.g., `sin`). Native functions have an implementation-defined accuracy, but tend to be faster. For example, AMD notes that native functions can be 1.8× to 34.2× faster than their non-native counterparts on AMD Evergreen and Northern Islands GPUs, respectively [Adv15, p. 3-42]. Similarly, according to NVIDIA, on CUDA the performance of `__sinf` is substantially higher than that of `sinf`, at the cost of accuracy reduction [NVI17a, Section 11.1.5]. The *half* versions of functions have a specified minimum accuracy, but may still be faster than the base versions, as those requirements are lower. According to ARM, on their GPUs, the native versions of `sin`, `cos`, `tan`, `divide`, `exp` and `sqrt`, as well as `half_sqrt`, are faster than their precise counterparts, but no other native/half function is [ARM17a, Section 9.3].

2.5.7 Work-group Size

OpenCL and CUDA both allow tuning work-group/block size, that is, how many work-items each work-group consists of. This decision may have a significant impact on the performance of programs.

On NVIDIA hardware, it is best to choose a work-group size that is a multiple of 32, whereas on AMD it should be a multiple of 64 [Adv15, p. 2-29]. Intel recommends power-of-two sizes between 64 and 256 for kernels with barriers, and 32 for kernels without, for its GPUs [Int17e].

Agosta et al. [ABDFP15] tested the impact of the work-group size on cryptographic benchmarks and found that on AMD and NVIDIA GPUs, increasing the work-group size leads to improved performance except for the AMD HD 6850 GPU on the KeeLoq benchmark, where a work-group size of 64 resulted in better performance than a work-group size of 128. However, there is an upper limit to the work-group size that can be chosen, due to GPU resource limits.

[7] Hardware units that each work-group is assigned to

2.5.8 Shared Virtual Memory

OpenCL 2.0 introduced SVM [Khr15, p. 174], a feature that simplifies memory management. Not only can SVM reduce the need for marshaling data structures that contain pointers, but programs that use fine-grained sharing can omit data transfers entirely as these are managed transparently by the implementation.

ARM Bifrost GPUs do not support OpenCL 2.0, but SVM features are still available using the `cl_arm_shared_virtual_memory` extension [ARM17a, Section E.5.1].

Mukherjee et al. [MGY+15] note that using SVM provides consistent performance benefits. These results suggest that using SVM should be considered by as a potentially profitable optimization for compilers targeting OpenCL.

2.5.9 Overview

Different OpenCL devices have different characteristics, and non-trivial programs need to be transformed (either manually or automatically) in a manner that is aware of these differences for each target device.

Table 2.2 shows a number of factors that impact performance on various devices. We can see that there are significant differences between CPUs and GPUs, but even within each device category the best approach to optimize a program is not always clear.

Table 2.2: Factors that impact performance and their applicability to CPU and GPU devices.

Characteristic	CPU	GPU
Memory Coalescing	No	Yes
Bank Conflicts	No	Yes
Dedicated Local Memory	No	Yes
Dedicated Texture Memory	No	Yes
Branch Divergence Performance Degradation	Usually not[8]	Yes
Best floating-point type	Depends	Depends
Work-group size	Depends	Depends
Shared Virtual Memory (SVM)	Profitable, if available	Depends

2.6 Target-aware Optimizations

As described in the previous section, certain programming techniques and patterns have a different impact on different types of devices. As such, many optimizations need to be *target-aware*, that is, have some degree of knowledge over the target device for the application, in order to properly improve a given program. This section describes some of these optimizations.

2.6.1 Tiling

In certain programs, storing data in local memory (shared memory in CUDA terminology) can be used to optimize memory accesses by reducing the redundant data transfers

[8] On Intel CPUs, the ability to statically determine whether branches can diverge has an impact on auto-vectorization.

and/or coalesce memory accesses.

NVIDIA exemplifies this technique with matrix multiplication algorithms [NVI17a, Section 9.2.2.2]. A naive version results in wasted bandwidth, either in the form of redundant reads or fully serialized accesses that limit performance. Instead, work-items can load data in a coalesced manner to local memory and then load the data from the local memory to perform the computation. If the matrix is too large to store in memory, then this optimization can still work, by dividing the matrix in tiles and only storing a single tile (per matrix) in shared memory at once.

Figure 2.15 shows two versions of the matrix multiplication kernel (assuming square input matrices, have the same size and have a size that is a multiple of 16): the naive version in Figure 2.15a and the tiled version in Figure 2.15b.

In the naive version, if a work-group size of (128, 1) is used, then all work-items in the same work-group have the same value of y. All work-items access the same position of A, and the access to B is coalesced. Unfortunately, the broadcast access to A wastes bandwidth as only a single value of the entire cacheline is used. A work-group size of (1, 128) is even worse, as the fact that all work-items in a work-group share the x value means that the accesses to A will be fully serialized.

The version with tiling forces a work-group size of 16×16 (line 3) and divides the matrix into tiles of that size. Each work-item in a group loads a single cell per tile to the local memory, and then loads the data from the local memory to perform the computation. According to NVIDIA, on an NVIDIA Tesla K20X GPU, matrix multiplication using tiling is $\approx 2.26\times$ faster than the naive version [NVI17a, Section 9.2.2.2].

2.6.2 Loop Unrolling

Loop unrolling [ALSU06, p. 735] can improve performance because it increases the opportunities for instruction-level parallelism, reduces the overhead of having loops and improves data locality. On GPUs, loop unrolling can be particularly profitable, because branching operations are more costly than on CPUs and they may have hardware resources that support higher Instruction-Level Parallelism (ILP) levels. AMD recommends unrolling small loops (up to 32 instructions) on Southern Island GPUs for improved performance [Adv15, Section 2.8.7.3]. However, loop unrolling can also degrade performance, by increasing register pressure and program code size.

The use of loop unrolling to optimize cryptography GPU programs was studied by Agosta et al. [ABDFP15]. They examined the performance impact of loop unrolling and concluded that:

- For the Data Encryption Standard algorithm, the loop should be fully unrolled, because there are few iterations and using it results in a performance improvement;
- For the MD5-crypt algorithm, the authors tested the impact of performing unrolling up to a factor of 50, the maximum allowed by the compiler. For this algorithm, they concluded that there were small speedups for small unroll factors and slowdowns for unroll factors of 10 or more, in comparison to the original version with no unrolling.
- For the KeeLoq algorithm, they tested the impact of unrolling the main loop up to a full unroll. In this case, unrolling presents performance improvements, especially on platforms with an instruction cache. On AMD GPUs (R700 and R800 architectures), loop unrolling is only beneficial if the unroll factor is up to

```
kernel void matmul(global float* out, global float* A, global float* B) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    int size = get_global_size(0);

    float acc = 0;
    for (int i = 0; i < size; ++i) {
        acc += A[y * size + i] * B[i * size + x];
    }

    out[y * size + x] = acc;
}
```

(a) Naive matrix multiplication of square matrices.

```
#define TILE_SIZE 16

__attribute__((reqd_work_group_size(TILE_SIZE, TILE_SIZE, 1)))
kernel void matmul(global float* out, global float* A, global float* B) {
    int lx = get_local_id(0);
    int x = get_global_id(0);
    int ly = get_local_id(1);
    int y = get_global_id(1);
    int size = get_global_size(0);

    local float tile_a[TILE_SIZE][TILE_SIZE];
    local float tile_b[TILE_SIZE][TILE_SIZE];

    float acc = 0;

    for (int tile_start = 0; tile_start < size; tile_start += TILE_SIZE) {
        tile_a[ly][lx] = A[y * size + (tile_start + lx)];
        tile_b[ly][lx] = B[(tile_start + ly) * size + x];
        barrier(CLK_LOCAL_MEM_FENCE);

        for (int i = 0; i < TILE_SIZE; ++i) {
            acc += tile_a[ly][i] * tile_b[i][lx];
        }
    }

    out[y * size + x] = acc;
}
```

(b) Tile-based matrix multiplication of square matrices.

Figure 2.15: OpenCL matrix multiplication algorithm demonstrating how to implement tiling to coalesce memory accesses.

a quarter of the loop executions. Above that point, the code no longer fits in the cache and there is a performance loss.

Loop unrolling was also evaluated by Du et al. [DWL+12]. They ported CUDA programs to OpenCL in order to understand the necessary optimizations to achieve acceptable performance across multiple platforms. The OpenCL version was only competitive with the CUDA version in terms of performance after unrolling two loops. In contrast, the CUDA compiler automatically performed this optimization.

Note that both CUDA [NVI17d, Section B.23] and OpenCL [Khr16, Section 6.11.5] provide language mechanisms that serve as compiler hints to unroll loops. Figure 2.16 demonstrates the preferred mechanism for loop unrolling as of OpenCL 2.0. In this example, the attribute `loop_unroll_hint` [The13] can be used to indicate to the OpenCL implementation that 2 is the preferred loop unroll factor. On NVIDIA's OpenCL implementation, the `cl_nv_pragma_unroll` extension [WAG09] is available, allowing the use of `#pragma unroll <FACTOR>` to achieve the same result.

```
1 kernel void unroll_example(global float* out, global float* in, int n) {
2     size_t pos = get_global_id(0);
3     float acc = 0;
4     __attribute__((opencl_unroll_hint(2)))
5     for (int i = 0; i < n; i++) {
6             acc += in[pos * n + i];
7     }
8     out[pos] = acc;
9 }
```

Figure 2.16: Demonstration of OpenCL loop unroll hints.

2.6.3 Task Parallelism

Task parallelism consists of executing multiple tasks simultaneously, and it can be used to improve the performance of some programs. Brodtkorb et al. [BHS13] suggest two approaches to enable some instances of task parallelism:

- Simultaneous use of the CPU and the GPU. In this case, the CPU performs operations while waiting for the GPU to finish.
- Execution of multiple different kernels on the same GPU. Albeit limited, GPUs can indeed execute multiple kernels at the same time. This enables limited task parallelism at the GPU level.

ARM advises avoiding function calls to `clFinish`, as well as synchronous functions, to ensure that the host and the device can work in parallel [ARM17a, Section 8.2].

2.6.4 Thread-Coarsening

Thread-coarsening [MDO14] consists of grouping N work-items into a single one, so that each thread performs more work (where N is called the *coarsening factor*). If $N = 1$, then no coarsening is applied.

There are multiple reasons that explain the performance improvements associated with this optimization. For instance, when each work-item recomputes certain values

that are the same across all work-items, thread-coarsening reduces the amount of redundant work. In addition, it also reduces the number of threads to be executed at runtime. However, thread-coarsening can also lead to slowdowns if the wrong coarsening factor is used.

Magni et al. [MDO14] developed an automated model to predict the best coarsening factor. In their tests, they found that their model could result in speedups ranging from 1.11× to 1.33× for the evaluated architectures (i.e. NVIDIA's Fermi and Kepler and AMD's Cypress and Tahiti).

Shen et al. [SFSV13] mention an optimization which they call *MergeN*. Although they never specifically use the expression *thread-coarsening*, the two optimizations are fundamentally the same. They studied the impact of thread-coarsening on CPUs (both for AMD and Intel implementations of OpenCL) with coarsening factors of 4 and 16 for the PathFinder benchmark. On the AMD platform, they obtained a speedup of 26% for a factor of 4, with an additional speedup of 7% for a factor of 16. The Intel version has a performance improvement of 5% for a factor of 4, but a slowdown of 2% for a factor of 16. The explanation given for the improvements, however, is different. According to the authors, the reason for the speedup is that thread-coarsening improves cache utilization. The slowdowns are explained by the register spills that the additional workload causes.

Pennycook et al. [PHW+13] studied the impact of *work-item* and *work-group* distribution. This technique consists of keeping the same number of work-items per work-group, but changing the number of total work-groups (and, by extent, changing the total number of work-items). As such, each work-item must perform more work in the *coarse-grained* version than the *fine-grained*. If there are $(i_{max}, j_{max}, k_{max})$ tasks to perform, then the thread coarsening factor is:

$$\text{Thread Coarsening Factor} = \frac{i_{max} \times j_{max} \times k_{max}}{\text{SIMD width} \times \text{compute units}} \tag{2.1}$$

On AMD's OpenCL platform for an Intel X5550 CPU, this optimization was essential [Int09]. The coarse-grained version of the tested program completed in less than 10 minutes, whereas the fine-grained version took over one hour for the same test. However, the authors found that the fine-grained version is better for GPUs.

2.6.5 Overview

This section discusses several optimizations that may improve the performance of OpenCL and CUDA programs. Achieving good performance requires knowledge of the target device, as optimizations that are profitable on some devices may degrade performance on others. Fortunately, many of these transformations can be performed automatically by a compiler, to reduce the number of program versions and device knowledge that are required.

2.7 Summary

This chapter discusses the relevant background relevant to this thesis, such as the MATLAB and OpenCL programming languages, and the Z3 SMT solver. Since a significant component of this thesis is target-aware optimization, this chapter also covers the targeted OpenCL-compatible devices, their performance characteristics, and related optimizations.

3

Related Work

Contents

This chapter describes existing approaches that solve problems related to the work presented in this thesis. These include MATLAB compilers and libraries for GPU programming, as well as the MATISSE compiler framework, that was used as a starting point for our own prototype.

3.1 The MATISSE Compiler Framework

MATISSE (*A MATrix(MATLAB)-aware compiler InfraStructure for embedded computing SystEms*) [BPN+13] is a framework designed to compile high-level matrix programs. This compiler framework includes support for source-to-source MATLAB transformations, low-level C code generation and two separate OpenCL backends: an initial prototype, MATISSE CL V1 [BRC15a], and the prototype developed during this thesis, MATISSE CL V2. An overview of the architecture of MATISSE, focused on the C backend and MATISSE CL V2, is included in Chapter 4.

MATISSE's source-to-source MATLAB transformations are based on a Domain-Specific Language (DSL) named LARA [BPN+13], in which programmers write scripts to analyze and transform the input applications. Figure 3.1 shows a simple LARA example, in which the code is modified to print a message before every execution of functions named `matmul`. The most important elements of this script are the `select` construct in line 2, which finds a pattern in the code (in this case, a loop inside a function), the `condition` construct in line 6, which further filters the results (in this case, by limiting the search to functions with a name starting with `matmul`) and the `apply` section in lines 3 to 5, which lists the actions to execute for each instance of the pattern. On the C and OpenCL backends, LARA scripts can be used to override the inferred types of MATLAB variables.

```
1 aspectdef PrintOnLoopStart
2     select function.loop end
3     apply
4         $loop.insert before "fprintf('Loop Start');";
5     end
6     condition $function.name.startsWith('matmul') end
7 end
```

Figure 3.1: LARA program that adds a printing message before every loop execution, on functions with names starting with `matmul`.

MATISSE features a highly customizable C code backend originally designed to support a wide range of C compilers, including High-level Synthesis (HLS) tools [BPN+13, RBC16]. The original MATISSE C code backend traversed the MATLAB AST, inferring the variable types and generating the corresponding C code for each traversed node.

The MATISSE CL V1 prototype extends MATLAB with a set of OpenACC-inspired directives for parallelism, as shown in Figure 3.2. MATLAB loops to parallelize are annotated with the `%acc parallel loop` directive (and a matching `%acc end` directive at the end). MATISSE then analyses the `copyin` and `copyout` parameters to determine which data transfers to insert. MATISSE CL V1 can only parallelize loops, not matrix operations. No data carried dependency testing is performed, and the kernel is assumed to be correct. The only validation is checking that the referenced variables

exist and are appropriately declared. All matrix variables must be defined/allocated before the loop, even if they are marked as `copyout`, as their size information is used to determine the memory space to allocate on the OpenCL device.

```
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
    % acc end
end
```

Figure 3.2: Directive-annotated MATLAB program that, when compiled with MATISSE CL V1, generates OpenCL code for element-wise matrix multiplication.

Figure 3.3 demonstrates a MATLAB program using 2D parallelism. In this example, both loops are run in parallel, in a single kernel. MATISSE only parallelizes the inner loop due to the `%acc loop` directive. If this directive was removed, then MATISSE would generate a sequential inner loop.

```
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
        % acc end
    end
    % acc end
end
```

Figure 3.3: Directive-annotated MATISSE CL V1 program with 2D parallelism.

An in-depth description of MATISSE CL V1 is presented in [Rei14]. This backend performs the following stages to generate OpenCL code:

- Parsing of MATLAB code to generate an AST;
- Identification of directives, replacing AST comment nodes with AST directive nodes;
- Decomposition of Complex Expressions, generating temporary variables if needed;
- Aggressive function inliner;
- Directive cleaner to move directives if required due to the expression decomposer stage;

- Outlining of directive regions, using the `copyin` and `copyout` directive parameters to determine function arguments and returned values;
- OpenCL code generation for the outlined functions, and C code generation for the remaining code.

Table 3.1 lists the directives supported by the MATISSE CL V1 backend. The most significant of these are the `parallel loop` and the `end` directives, which mark the beginning and the end of a parallel section, respectively. Note also that the semantics of the directives is fairly low-level, to the point that one directive (`ignore`) directly maps to an OpenCL operation.

Table 3.1: List of MATISSE CL v1 directives

Directive	Description
`% acc parallel loop`	Indicates that a `for` loop should be compiled to OpenCL.
`% acc parallel`	Indicates that an inner `for` loop should be executed in parallel.
`% acc ignore`	Indicates that a section of code should be ignored by the MATISSE CL v1 backend, but still be executed by MATLAB and the remaining MATISSE backends.
`% acc end`	Indicates the end of a parallel loop, parallel or ignore section.
`% acc barrier`	Marks a local or global memory barrier (see Section 2.4).

3.2 MATLAB GPU APIs

This section describes existing MATLAB APIs/toolboxes for GPU programming.

3.2.1 MathWorks Parallel Computing Toolbox

MathWorks' *Parallel Computing Toolbox* [Mat13b] is an API for parallel computing, including a set of functions developed to execute certain operations on CUDA-compatible GPUs. Figure 3.4 shows a MATLAB program that runs a computation on the GPU, including the necessary data transfers. MATLAB allows programs to directly allocate memory on the GPU (e.g., with the `GPUArray.ones` function) or copy existing matrices to the GPU (using the `gpuArray` function). The results of these operations are *GPU arrays*. The programmer may use certain MATLAB functions/operators using values of this type to run the operations on the GPU. Finally, the `gather` function copies the results back to the CPU.

MATLAB also includes features for multi-core CPU parallelism, such as the `parfor` construct [Mat18b]. Figure 3.5 shows a MATLAB program that takes advantage of multi-core parallelism to compute an element-wise multiplication on the CPU in parallel. The outer loop (`parfor`) launches distributes the iterations across multiple *workers*, from a *thread pool*. However, MathWorks advises against using `parfor` on these simple computational tasks, as the time spent on transferring data to the workers may be longer than the time savings due to parallelization.

```
% Allocate matrix of ones on GPU
A = parallel.gpu.GPUArray.ones(1024, 1024);
% Allocate random matrix on the CPU and copy it to the GPU
B = gpuArray(rand(1024));

% Perform Computation on the GPU
C = A .* B;

% Copy result to CPU
C = gather(C);
```

Figure 3.4: MATLAB program that performs an element-wise multiplication operation on a GPU.

```
A = ones(1024);
B = rand(1024);

C = zeros(1024);

parfor j = 1:1024,
    C(:, j) = A(:, j) .* B(:, j);
end
```

Figure 3.5: MATLAB program that performs an element-wise multiplication on the CPU in parallel, using `parfor`.

The Parallel Computing Toolbox also includes features designed to support cluster parallelism. However, these features are out of the scope of this thesis.

3.2.2 GPUmat

GPUmat [GP-15] is an open-source library enabling GPU (CUDA) computations from MATLAB[1]. An in-depth description of how to use it can be found in [GP-12].

Figure 3.6 presents a program that computes an element-wise matrix multiplication on the GPU using the GPUmat API. As seen in this example, this API is very similar to the GPU capabilities of the Parallel Computing Toolbox, as both include functions to explicitly allocate or copy data between the CPU and the GPU, and reuses MATLAB functions/operators (e.g., `.*`) to perform GPU computations on GPU buffers.

3.3 Compiling MATLAB to Non-GPU Platforms

This section describes other tools that compile the MATLAB language (or similar) to low level programming languages such as C and FORTRAN, but without a focus on GPUs.

[1] As of the time of writing, GPUmat only supports CUDA 5.0 and has not been updated since 2015.

```
% Explicitly allocate data on the CPU and copy it to the CPU
A = GPUdouble(ones(1024));
% Or directly allocate the matrix on the GPU
B = rand(1024, 1024, GPUdouble);

% Compute data on the GPU
C = A .* B;

% Copy it back to the CPU
C = double(C);
```

Figure 3.6: MATLAB program that performs an element-wise multiplication on the GPU using GPUmat.

3.3.1 MathWorks Coder

MATLAB Coder [Mat18e] is a tool developed by MathWorks designed to compile MATLAB to C/C++. The generated code can be used as a library, integrated in C applications, compiled to an executable, or packaged as MEX files (native code libraries that can be called from MATLAB programs). Embedded Coder extends MATLAB Coder with features for embedded systems, such as an improved ability to customize the generated code and target-specific optimizations.

GPU Coder, an extension to Coder that generates CUDA code from MATLAB, is described in Section 3.4.6.

3.3.2 FALCON

DeRose et al. developed FALCON [DRP99], a MATLAB to FORTRAN 90 compiler. FALCON consists of three main systems: *Program Analysis*, *Interactive Restructuring* and *Code Generation*.

The Program Analysis System reads the MATLAB program and generates an SSA-based internal representation. The type/shape inference algorithm is applied on the SSA code. The Interactive Restructuring System performs transformations on the intermediate code. The Code Generation System generates the final Fortran 90 code.

FALCON deals with type/shape inference in two phases: static and dynamic. In the static phase, the types of variables are determined based on the semantics of statements and information about input variables. For variables of unknown type or shape, FALCON emits various code versions for each type possibility (FALCON considers only two, for simplicity: *real* and *complex*), and selects the code to execute at runtime.

To evaluate the compiler, the authors used 12 MATLAB benchmarks (listed in Table 3.2), and compared the generated FORTRAN 90 code with other versions, notably C code generated by the MATLAB Compiler[2]. The generated FORTRAN 90 outperformed the C code on 11 of the 12 benchmarks, with speedups from 1.8× to 179.7× (and a geometric mean of 6.87×).

[2] According to the MathWorks Support Team [Mat16], code generated by the MATLAB Compiler has the same performance as MATLAB, but the authors' results show that the generated C is clearly faster than MATLAB.

Table 3.2: List of benchmarks used to evaluate the FALCON compiler.

Short Name	Description
SOR	Successive Overrelaxation method
CG	Preconditioned Conjugate Gradient method
3D	Generation of a 3D-Surface
QMR	Quasi-Minimal Residual method
AQ	Adaptive Quadrature Using Simpson's Rule
IC	Incomplete Cholesky Factorization
Ga	Galerkin method to solve the Poisson equation
CN	Crank-Nicholson solution to the heat equation
RK	Two body problem using 4th order Runge-Kutta
EC	Two body problem using Euler-Cromer method
Di	Dirichlet solution to Laplace's equation
FD	Finite Difference solution to the wave equation

3.3.3 MC2FOR

MC2FOR [LH14] is a MATLAB to FORTRAN compiler integrated in the McLab umbrella project [Sab18].

MC2FOR uses the McLab front end to obtain an AST representation of all reachable functions, which is then processed by McSAF [DH12a], a code transformation engine designed specifically for MATLAB. McSAF outputs a lower-level AST and performs some initial analyses, such as distinguishing identifiers that refer to variables from those that refer to functions. After that, MATLAB Tamer [DH12b] translates this AST into a three-address code IR called *TamerIR*, which is used by the Fortran IR Generator. MC2FOR reuses Tamer's pre-existent analyses, but the authors also created new ones, such as shape analysis. Because TamerIR's representation produces several temporary variables, the authors consider that direct code generation would result in unreadable code. For this reason, they introduced Tamer+, a component that converts the TamerIR back to an AST representation, aggregating temporary variables in the process. The compiler then generates Fortran from the AST, which is passed to a pretty printer to produce the final code.

To evaluate MC2FOR, the authors used 11 benchmarks, listed in Table 3.3, and compared the execution time of the Fortran code compiled with GFortran of GCC 4.6.3 versus the original MATLAB running on MATLAB R2013a. The authors found that, aside from the `clos` benchmark, the Fortran version was 2.9× to 34.3× faster than the MATLAB version. The `clos` benchmark, however, performed 25× worse than the original MATLAB, due to GFortran's inefficient implementation of matrix multiplication. Overall, MC2FOR-generated code was 6.27× faster (geometric mean) than the original MATLAB.

3.3.4 MIX10

MIX10 [KH14] is a compiler, integrated in the McLab umbrella project [Sab18], that converts MATLAB programs into X10 [IBM18], a statically typed language designed for high-performance computing that can be compiled to C++ or Java.

MIX10 reuses the McLab front-end, McSAF [DH12a] and MATLAB Tamer [DH12b]

Table 3.3: List of benchmarks used to evaluate the MC2FOR compiler.

Short Name	Description
adpt	Adaptive Quadrature using Simpson's rule
bbai	Babai algorithm
bubl	Standard bubble sort algorithm
capr	Capacitance of transmission line
clos	Transitive closure of a directed graph
crni	Crank-Nicholson solution to the heat equation
dich	Dirichlet solution to Laplace's Equation
diff	Diffraction Pattern of monochromatic light
fiff	Finite-difference solution to the wave equation
mbrt	Mandelbrot set
nb1d	N-Body simulation

components, but extends them by adding concurrency constructs and X10-specific analyses. The TameIR code and analyses results are passed to the X10 code generator to generate the final code.

The authors found it challenging to generate X10 code with performance competitive with that of MATLAB Coder-generated C code (see Subsection 3.3.1) or MC2FOR-generated FORTRAN code (see Subsection 3.3.3). Notably, X10 supports two different types of arrays with different trade-offs between flexibility and performance, and the compiler must determine which one to use. Moreover, the authors found that casts of double-precision floating point values to integer types are extremely slow on the C++ backend of X10, because the C++ backend adds a value range check before every cast, and these casts are commonly injected to deal with situations where MATLAB allows double-precision values but X10 requires integer types. To deal with this issue, the authors developed an analysis to determine whether a double variable can be safely treated as an integer (specifically `Long`) type, i.e., if declaring a variable as an integer does not change program semantics. Finally, MIX10 implements MATLAB's `parfor` loops (parallel `for` loop execution) using X10's concurrency features.

To evaluate their compiler, the authors used a set of 17 MATLAB benchmarks (listed in Table 3.4), and compared the performance of the X10 code compiled with the C++ backend with the Java backend, the MATLAB runtime and MC2FOR. All static code generators outperformed MATLAB in most cases (except for `clos`, where it outperforms all but MATLAB Coder, and `mcpi` and `optstop`, where it outperforms Coder). Using the X10 C++ backend, the authors obtained a geometric mean speedup of 6.8× over MATLAB, above the 6.3× speedup of Coder over MATLAB, but below the 10.2× speedup of MC2FOR over MATLAB. Two benchmarks are problematic for MIX10: `clos` and `lgdr`. The `clos` benchmark heavily relies on matrix multiplication and MIX10 uses a naïve matrix multiplication algorithm, and `lgdr` repeatedly transposes a row vector to a column vector – an operation that is highly optimized on MATLAB and Fortran, but slow on MIX10.

3.3.5 MatJuice

MatJuice [FBH16] is a MATLAB to JavaScript compiler integrated in the McLab umbrella project [Sab18].

Table 3.4: List of benchmarks used to evaluate the MIX10 compiler.

Short Name	Description
bbai	Babai algorithm
bubl	Standard bubble sort algorithm
capr	Capacitance of transmission line
clos	Transitive closure of a directed graph
crni	Crank-Nicholson solution to the heat equation
dich	Dirichlet solution to Laplace's Equation
diff	Diffraction Pattern of monochromatic light
edit	Edit distance between two strings
fiff	Finite-difference solution to the wave equation
lgdr	Compute Legendre Polynomials up to a degree, and some derivatives
mbrt	Mandelbrot set
nb1d	N-Body simulation
matmul	Naïve matrix multiplication
mcpi	Calculate π using Monte Carlo method
numprime	Count the number of primes up to a given integer
optstop	Solution to the optimal stopping problem
quadrature	Quadrature approach for calculating integral

The compiler uses the McLab and Tamer [DH12b] projects to convert MATLAB source code into an IR called TameIR. The authors then perform analyses and transformations on TameIR before generating the final JavaScript code.

To implement matrices in JavaScript, the authors used typed arrays. However, operations on typed arrays, like ordinary JavaScript arrays but unlike MATLAB matrices, follow pass by reference semantics, rather than pass by value semantics. This issue can be resolved by inserting copies on every array assignment, but this comes at a performance and memory cost. A better approach is to only insert copies when needed. The authors developed an intra-procedural approach for this that injects copies only when needed, and is capable of detecting that copies are only necessary when the program enters certain branches of the code.

The authors compared MatJuice with the MATLAB runtime using 16 benchmarks, listed in Table 3.5. The authors found that MatJuice outperforms MATLAB on 10 of the tested benchmarks, but MATLAB is, on average, 20% faster than MatJuice. The authors attribute this difference to MATLAB's highly optimized matrix libraries.

3.3.6 MATLAB to C Targeting Application Specific Instruction Set Processors

Latifis et al. [LPD+16] present a MATLAB to C compiler that generates C code with function calls to intrinsics that represent custom instructions, notably SIMD operations, in Application Specific Instruction Set Processors (ASIP). The compiler takes as input an XML file with information about the available intrinsics and a MATLAB file with annotations (in the form of function calls) to indicate type and shape information of variables, as well as preferred vector sizes of SIMD operations in blocks of code.

The compiler parses the MATLAB code, constructs the AST, then performs type inference, instruction selection, conversion to a low-level IR (by decomposing complex

Table 3.5: List of benchmarks used to evaluate the MatJuice compiler.

Short Name	Description
babai	Babai algorithm
bubble	Standard bubble sort algorithm
capr	Capacitance of transmission line
clos	Transitive closure of a directed graph
collatz	Test the Collatz conjecture up to a given integer
dich	Dirichlet solution to Laplace's Equation
fdtd	Finite Difference Time Domain
fft	Fast Fourier Transform
fiff	Finite-difference solution to the wave equation
lgdr	Compute Legendre Polynomials up to a degree, and some derivatives
makechange	Compute the ways to make change for
matmul	Naïve matrix multiplication
mcpi	Calculate π using Monte Carlo method
nb1d	N-Body simulation
numprime	Count the number of primes up to a given integer

matrix expressions to simpler ones) and finally C code generation.

The authors evaluated their compiler on an ADRES ASIP (BoT template) with a set of 6 six fixed-point DSP algorithms with varying input sizes, comparing their result with the output of MathWorks Coder, and achieved speedups between 2× and 30×.

In [LPD+17], the authors presented an improved version of the compiler, that includes a dataflow analysis step after the low-level IR conversion and before code generation, designed to remove packing/unpacking operations. The authors evaluated this version with six benchmarks (FFT, CFO, FIR, mean, CORDIC and QR decomposition), and were able to achieve speedups between 2× and 97× over the MathWorks Coder compiler.

3.4 Compiling MATLAB to GPUs

This section describes tools that compile the MATLAB language (or Octave [Oct14a]) and output code in GPU programming languages or using GPU APIs. The MATISSE OpenCL backend MATISSE CL V1 has already been described in Section 3.1.

3.4.1 MATLAB Execution on GPU based Heterogeneous Architectures

MATLAB Execution on GPU based Heterogeneous Architectures (MEGHA) [PAG11, PG12] is a MATLAB to CUDA compiler proposed by members of the Indian Institute of Science in 2011. MEGHA incorporates a number of optimizations, including heuristics to determine when to offload code to the GPU and when to keep it running on the CPU. The authors implemented MEGHA using GNU Octave [Oct14a].

MEGHA implements the following compiler stages:

- *Code Simplification*: The compiler replaces complex expressions with simpler equivalent ones. This transformation does not fully simplify array accesses when

`:` and `end`-containing indices are used.

- *Semantics Preserving Transformations*: Replaces `:` and `end`-based indices into a simpler equivalent. For instance, `A(:)` becomes `A(1:length(A))`[3]. Additionally, MEGHA transforms MATLAB loops, so that the iteration variable is never modified in the loop body.

- *Static Single Assignment Construction*: Converts the transformed code into an SSA-based IR. Note that in this SSA representation, matrix assignments do not produce new SSA variables.

- *Type and Shape Inference*: Determines the intrinsic type (e.g., `integer`), shape (e.g., `3D-array`) and size (eg., $2 \times 3 \times 4$) of each program variable. It is possible for the programmer to override types of variables.

- *Kernel Identification*: MEGHA identifies *kernels*, blocks of code on which the scheduling (i.e., CPU vs GPU execution) decisions will be made. MEGHA identifies and groups sets of statements that are GPU friendly, tries to eliminate arrays by converting them to scalars when possible, and finally converts kernels into parallel loop nests.

- *Parallel Loop Reordering*: Determines the most efficient way to execute a given parallel loop nest (similar to determining the most efficient loop nest), in terms of memory locality (for CPUs) and memory coalescence (for GPUs). Since the most efficient iteration traversals for CPUs and GPUs are different and at this stage the compiler has not yet determined which kernels should be offloaded, this stage computes and returns the order for both cases.

- *Mapping and Scheduling*: MEGHA attempts to minimize the total execution time (including time spent on data transfers) using a scheduling heuristic.

- *Global Data Transfer Insertion*: The compiler inserts the necessary data transfers between the CPU and the GPU.

- *Code Generation*: MEGHA generates C++ code for the program sections mapped to the CPU and CUDA code for GPU kernels.

MEGHA only supports a subset of MATLAB. In particular, user-defined functions are not supported unless the compiler frontend is extended, not all data types are supported, the type inference algorithm is limited and auto-growing matrices are not supported.

To evaluate MEGHA, the authors selected a set of 10 benchmarks, listed in Table 3.6, with varying input sizes, and compared the execution time of the generated C++ code (with and without CUDA) with the MATLAB runtime on a quad-core Intel Xeon 2.83GHz with a GeForce 8800 GTS GPU and a Tesla S1070.

On the data parallel benchmarks, the MEGHA-generated pure CPU version outperformed the MATLAB version on the `fdtd`, `nb1d` and `nb3d` benchmarks, but was

[3]Usually, this would be an incorrect program transformation on matrices with more than one dimension (e.g., on 2×2 matrices, the length is 2 but all 4 elements should be included), even when the matrix access expression itself only has a single index, and the `numel` function should be used instead. The authors do not specify if and how they deal with matrices of more than 1 dimension on this transformation, particularly since at this stage type and shape inference has not yet been executed.

Table 3.6: List of bechmarks used to evaluate the MEGHA compiler.

Short Name	Description	Data Parallel
bscholes	Stock Option Pricing	Yes
capr	Line Capacitance	No
clos	Transitive Closure	Yes
crni	Heat Equation Solver	No
dich	Laplace Equation Solver	No
edit	Edit Distance	No
fdtd	EM Field Computation	Yes
fiff	Wave Equation Solver	No
nb1d	1D N-Body Simulation	Yes
nb3d	3D N-Body Simulation	Yes

slower than MATLAB on the `bscholes` benchmark (they did not present results for their CPU version of `clos`). On the 8800 GPU, they were able to achieve speedups over MATLAB from 2.7× to 172×, though some benchmarks/inputs failed to run due to lack of memory. On the Tesla GPU, they were able to achieve speedups over MATLAB from 2.7× to 191×, with a geometric mean speedup of 19.8×.

On the remaining benchmarks, the MEGHA-generated pure CPU version outperformed MATLAB in all cases, but the GPU-accelerated version performed roughly the same as the pure CPU version (performance variations $< 1\%$).

The authors also compared their GPU version with GPUmat, running the data parallel benchmarks on the Tesla S1070 (with only a single input size for each). MEGHA was able to consistently outperform GPUmat, with speedups ranging from 1.5× to 90×, geometric mean of 7.98×.

3.4.2 Chun-Yu Shei et al.'s MATLAB to CUDA compiler

Chun-Yu Shei et al. developed a MATLAB to C++ and CUDA compiler [SYRC11] as part of the HLLC/ParaM project [Cha13].

The authors focus on optimizing array statements by *scalarizing* them (i.e., converting array expressions to scalar loops) when converting to C++. The compiler only optimizes certain segments of code to C++/CUDA, leaving the rest of the code to be executed by the MATLAB runtime.

Their compiler stages are as follows:

- *Parsing*: The authors use the Octave parser to obtain an AST;
- *Conversion to the RubyWrite AST format*: As they use a Ruby-based DSL to perform AST manipulation (RubyWrite), the Octave-given AST is converted into a representation compatible with this language;
- *Expression Flattening*: All expressions are simplified by introducing temporary variables to store the results of subexpressions;
- *Type Inference*: This stage consists of the following steps:

    1. Variables are placed in the program representing the type of each variable at that point of the program (e.g., `iType_x` stores the type of `x`). These types

are modified when the variables they refer to are modified. The new type is based on the new value (e.g., given a statement `c = a + b`, the compiler injects the statement `iType_c = IXF_sum(iType_a, iType_b);`.

2. Conversion of the code to SSA form;
3. The compiler runs an *aggressive partial evaluator* that, whenever possible, computes the results of the type variables. According to the authors, the "vast majority" of types are evaluated by this mechanism. However, some types are only known at runtime.

- *Advanced Optimizations*: Now that most variable types are available, the compiler performs *code sequence optimization* and *type-based specialization*.
- *Backend*: The compiler selects the functions to use from available libraries and then generates C++ and CUDA code.

To evaluate their compiler, the authors performed two groups of experiments. One based on applications and kernels, described in Table 3.7. On three of the applications/kernels, by generating C++ with OpenMP, they were able to achieve speedups between 1.5× (for NBody3D, one thread) to nearly 17× (NASMG eight threads). On the remaining thee applications, however, they were unable to achieve speedups, due to data movement costs (either due to the copy-on-write semantics of array arguments on the CPU, or the data transfers between CPU and GPU).

Table 3.7: List of bechmarks used to evaluate Chun-Yu Shei et al.'s MATLAB to CUDA compiler.

Short Name	Description
N-Body	3D N-Body Simulation
NASMG	Multigrid benchmark from the NAS benchmark suite
FDTD	Finite Difference Time Domain
Heated Plate	Thermal simulation
Forward	Analyze stock market data (part of Black Scholes)
Shallow Water 1D	Solver for shallow water equations

3.4.3 Chun-Yu Shei et al.'s MATLAB to GPUmat compiler

The HLLC/ParaM project also includes an additional MATLAB compiler by Chun-Yu Shei et al. [SRC11] that performs MATLAB-to-MATLAB compilation to output code using the GPUmat library. Their approach attempts to predict CPU and GPU performance and tries to balance workloads and minimize data transfers.

Their compiler first performs *type inference*, using the same approach described in Subsection 3.4.2. The compiler then processes each MATLAB function, using the following steps:

- *Identification of schedulable statements*: A *schedulable* statement is a statement that may be executed on the GPU. Only functions supported by GPUmat may be offloaded to the GPU.

- *Cost estimation*: Estimate how long each schedulable statement would take to run on the CPU and GPU as well as data communication costs associated with each operand, based on previously obtained empirical data.
- *Partitioning*: Determine which type of processor (CPU or GPU) should run each schedulable statement, using heuristics to determine the most efficient mapping.
- *Reordering*: The compiler attempts to make CPU and GPU computations execute simultaneously by reordering the schedulable statements.
- *Code Generation*: The compiler generates MATLAB code with the additional operations and checks. It also performs a final partial evaluation and dead-code elimination pass.

To evaluate this compiler, the authors used the benchmarks listed in Table 3.8. The authors evaluated two approaches: *greedy*, that selects every schedulable statement to run on the GPU, and *heuristic*, that attempts to determine when to offload statements to the GPU. Their greedy approach can achieve speedups on certain benchmarks (more than 7× on high input sizes on FDTD), but also significant slowdowns on others (consistently more than 100× slower on Shallow Water, regardless of input size). With their heuristic-based approach, they were able to eliminate the worst slowdowns, while still achieving speedups very close to those of the greedy approach. Using the heuristic approach, the authors were able to achieve a worst-case slowdown of around 20%.

Table 3.8: List of bechmarks used to evaluate Chun-Yu Shei et al.'s MATLAB to GPUmat compiler.

Short Name	Description
Arnoldi	Find the eigenvalues of general matrices
Shallow Water (1D)	Solver for shallow water equations
Heated Plate	Thermal simulation
Krylov	Construct a Krylov matrix with columns normalized
N-body (1D)	1D N-Body Simulation
N-body (3D)	3D N-Body Simulation
NASMG	Multigrid benchmark from the NAS benchmark suite
Forward	Part of Black Scholes, a finance application
Binomial	Part of Black Scholes, a finance application
FDTD	Finite Difference Time Domain

3.4.4 Velociraptor

Velociraptor [GH14] is a compiler toolkit that aims to provide a reusable approach to compile array-based languages (e.g., MATLAB) to CPUs and GPUs. In order to achieve this, the authors developed:

- A high-level IR called VRIR;
- An optimizer and LLVM+OpenCL code generator for VRIR;
- A high-level task-graph API called VRRuntime, to manage task dispatch and data transfers to the GPU;

- An extension for a MATLAB JIT named McVM [CBHV10] (part of the McLab project [Sab18]) allowing MATLAB to GPU compilation;
- An add-on to CPython allowing Python to GPU compilation. Python is out of the scope of this thesis, so we do not focus on this frontend.

Velociraptor can parallelize multiple types of VRIR constructs, including parallel-for loops (but not nested parallel-fors, as inner loops are always executed sequentially), parallel map (i.e., element-wise operations), parallel operators (e.g., matrix multiply) and accelerated sections. In accelerated sections, multiple statements can be executed on the GPU, enabling optimizations that Velociraptor would not be able to perform when parallelizing each parallel statement individually.

To evaluate the McVM frontend of Velociraptor, the authors used 4 benchmarks, presented in Table 3.9, and added GPU annotations. For the GPU version, the Velociraptor runtime was up to 15% faster than the Velociraptor CPU version, and up to the 3.62× faster than the MATLAB JIT. The authors found that runtime asynchronous dispatch adds overhead that is significant for small kernels. The compiler optimizations have no impact in these benchmarks because the optimizations are designed for for-loops, and the benchmarks are written using vectorized operations.

Table 3.9: List of bechmarks used to evaluate Velociraptor with the McVM frontend.

Short Name	Description
clos	Transitive Closure
nb1d	1D N-Body Simulation
nb3d	3D N-Body Simulation
fdtd	EM Field Computation

3.4.5 StencilPaC

StencilPaC [SCS16] is a compiler that is capable of automatically generating parallel code for stencil computations written in a subset of MATLAB. StencilPaC generates C code that executes in parallel using OpenMP [Ope15] for multi-core CPUs, MPI [Mes15] for distributed systems, or OpenACC [Ope17] for GPUs. Since this thesis is focused on execution of code in accelerators (particularly GPUs), their OpenMP and MPI code generators[4] are not discussed here.

The supported subset includes booleans and numeric operations, ranges, vectors, matrices with up to 2 dimensions, control structures, user-defined functions and some MATLAB built-in functions. StencilPaC traverses an AST annotated with type attributes. Each node has an associated code template that is instantiated to build the final C code. As an optimization, the compiler combines matrix assignments into a single loop if possible.

StencilPaC can parallelize element-wise matrix expressions and calls to `arrayfun`. Other operations (notably reductions and user loops) are never parallelized. The compiler inserts the appropriate OpenACC directives for parallelization and data transfers.

The authors used two benchmarks, described in Table 3.10 to evaluate their compiler. Their OpenACC version of EasyWave was 187× faster than the MATLAB version, and

[4]OpenMP supports accelerators [Li17], but the authors use OpenMP specifically for multi-threaded CPU parallelism. OpenMP directives for GPU parallelism are distinct from those for CPU parallelism.

17% slower than a handwritten CUDA version. Using OpenACC, they were able to improve the performance of Celular Automaton by 69185× (no handwritten CUDA version was available).

Table 3.10: List of bechmarks used to evaluate the StencilPaC compiler.

Short Name	Description
EasyWave	Shallow water equations for tsunami early warnings
Celullar Automaton	2D grid of cells, updated based on state of adjacent cells

3.4.6 GPU Coder

GPU Coder [Mat18d] is a compiler developed by MathWorks and introduced in MATLAB R2017b [Edd17] that generates CUDA code from MATLAB programs. GPU Coder can be used alongside Embedded Coder (see Subsection 3.3.1).

Figure 3.7 shows a simple MATLAB function that can be compiled to CUDA by GPU Coder. The `coder.gpu.kernelfun()` command functions as a directive that indicates that the GPU Coder should compile this function, and the rest of the code is common MATLAB code. GPU Coder supports a subset of the MATLAB language, based on the subset supported by MATLAB Coder [Mat18c]. The `%#codegen` directive is optional: its usage allows the MATLAB Code Analyzer to properly understand the context where the function is used and produce more relevant diagnostics, but is otherwise ignored by the compiler [Mat18a]. It is possible to specifically indicate to the GPU Coder that a loop should be mapped to the GPU using the `coder.gpu.kernel` operation.

```
1 function acc = gpufunc(A) %#codegen
2     coder.gpu.kernelfun();
3
4     acc = A;
5
6     for i = 0:98,
7         acc = acc .* A;
8     end
9 end
```

Figure 3.7: Example of a GPU Coder function that computes the 100th power of a matrix.

Another parallelization approach supported by GPU Coder is shown in Figure 3.8. This example does not use `coder.gpu.kernelfun`. Instead, the inner function `computeAverage` is executed on the GPU. Stencil kernels receive as input a window of the passed matrix and are executed once per element, with the scalar result of each execution constituting an element of the output matrix.

```
function result = gpufunc2(image) %#codegen
    result = gpu.coder.stencilKernel(@computeAverage, image, [3, 3],
        'same');

    function out = computeAverage(window)
        out = 0;
        for i = 1:5,
            for j = 1:5,
                out = out + window(j, i);
            end
        end
        out = out / 25;
    end
end
```

Figure 3.8: Example of GPU Coder program that blurs an image, by replacing each pixel with the average of its 3×3 grid.

3.5 MATLAB Type Inference Strategies

As MATLAB is a dynamically typed programming language, in order to translate code to statically typed languages, a scheme for determining the types of variables is required. This section reviews some of the approaches used by existent MATLAB compilers.

The FALCON [DRP99] type inference (*static phase*) algorithm is based on two phases: first, FALCON performs a simple forward-only type inference based on the types of constants, expressions and built-in functions, that leaves most types as *unknown* and treats variables within loops as having the same types as before those loops (but marks them for later refinement). It then repeatedly traverses the statements of as-of-yet unclear type (i.e., *marked*) and iteratively refines the types until a stable point is reached. Variables that still have unknown type after this second phase are handled at runtime. The FALCON type inference assumes that all MATLAB values can be considered *complex* double-precision foating point values, and that treating *logical* and *integer* types as *complex* would be purely a performance issue, not one of correctness. The MEGHA [PAG11] and Latifis et al. [LPD+16] compilers use similar approaches.

Chun-Yu Shei et al. [SYRC11] developed a type inference mechanism based on giving each program variable a secondary *type* variable (i.e., a `x_Type` for every `x`) that gets updated every time the type of the program variable changes. The type inference itself consists of performing an aggressive partial evaluation, that turns a significant number of these variables into constant values.

The McLab [DH12b] framework includes an interprocedural analysis framework that can be used for many purposes, including type inference. The analysis supports function *contexts* (i.e., representations of the received function arguments). The authors developed a value analysis on top of the interprocedural analysis, to determine which values and kinds (i.e., MATLAB `class`) each program variable can have, by performing a *forward propagation* of MATLAB values. The MC2FOR [LH14] compiler extends this value analysis to also cover shape inference.

3.6 Summary

This chapter discusses relevant work that is related to the work presented in this thesis.

The most important related work is MATISSE [BPN+13], the MATLAB compiler framework used to build the compiler prototype presented in this thesis. Although MATISSE already featured a MATLAB to C/OpenCL compiler, this thesis proposes a new, substantially improved, C/OpenCL backend.

It is possible to perform GPU computations directly in MATLAB, using APIs such as the Parallel Computing Toolbox [Mat13b] or GPUmat [GP-15]. An alternative is to use tools that compile MATLAB programs to run on the GPU. Table 3.11 compares some of these tools.

Table 3.11: Overview of previous approaches that compile MATLAB to GPU languages/APIs.

Name	Output	Parallelization Approach
MEGHA [PAG11]	C++/CUDA	Fully automatic
Chun-Yu Shei et al. [SYRC11]	C++/CUDA	Fully automatic
Chun-Yu Shei et al. [SRC11]	MATLAB + GPUmat	Fully automatic
Velociraptor [GH14]	LLVM/OpenCL	Fully automatic[5]
StencilPaC [SCS16]	C/OpenACC	Fully automatic
GPU Coder [Mat18e]	C/C++/CUDA	Directive-driven
MATISSE [Rei14]	C/OpenCL	Directive-driven

Most of the covered approaches are fully automatic, using heuristics to determine the components to offload. The exception is GPU Coder [Mat18e], which opts for a more user-controlled mechanism based on directives (e.g., `coder.gpu.kernelfun`). All four approaches target CUDA directly or indirectly (since GPUmat is based on CUDA as well).

[5] Accelerated sections require manually placing begin/end markers.

4

Compiler Prototype Architecture

Contents

This chapter describes the architecture of the MATISSE-based compiler prototype, including the parallel programming model, main compiler phases, algorithms and Intermediate Representations (IRs).

4.1 Programming Model

This section describes the language features exposed by MATISSE for the programmer to build sequential and parallel programs.

4.1.1 Supported MATLAB Subset

MATISSE supports a non-trivial subset of MATLAB, but some features are only supported for sequential code generation.

The main limitation of the MATISSE compiler, both for C and OpenCL code generation, is related to type inference. MATISSE needs to know the types of all program variables in order to successfully compile the program. It can do this by receiving the explicit types from the programmer, by reading an example input file with definitions for those variables and by propagating types using type inference. However, the type inference algorithm has limitations and may be unable to determine the type of all variables. In those cases, MATISSE requires the user to either manually specify the types of those variables, or change the program to be amenable to type inference.

Table 4.1 lists an overview of several MATLAB features and current MATISSE support. Note that some features, such as memory allocation functions, are supported only in sequential code (i.e., column Support - C). The missing features were generally not implemented either due to their dynamic nature (e.g., `eval`) or because they require substantial engineering efforts that are out of the scope of this thesis.

Table 4.1: Overview of the MATLAB subset supported by MATISSE

Feature	Description	Support	
		C	OpenCL
Basic Control Flow	Such as `if`, `while` and `for` statements	Yes	Yes
Function Calls	Incl. multiple outputs, optional arguments	Yes	Yes
Matrix Operations	Such as addition, multiplication and division	Yes	Parts
Matrix Indexation	Access matrix positions, incl. logical indices	Yes	Parts
Complex Numbers	Numbers with a non-zero imaginary part	No	No
Test of arguments	Use of `nargin` and `nargout`	Yes	Yes
Globals	Variables shared by the entire program	Parts	No
Exceptions	Such as `try/catch` statements	No	No
Advanced Features	Incl. classes, function handles	No	No
Core Functions	Simple built-in functions (e.g., `sin`, `numel`)	Yes	Yes
Matrix Allocation	Built-in functions that allocate memory	Yes	No
Advanced Functions	Such as `padarray`, `sort`, `conv2` and `fft`	Parts	No
Input/Output	File handling and console I/O	Parts	No
Toolboxes	Packages of MATLAB code	No	No
Classes	Object-oriented programming	No	No

Despite these limitations, the supported subset covers the most relevant parts of

MATLAB, is sufficient for research purposes, and can be extended in the future if needed.

Some code features, such as lambda functions, cause compile errors even if they are in unreachable code regions, as MATISSE is unable to even generate IR for those features.

4.1.2 The Directive API

MATISSE uses directives to control several processes, including parallelization and optimization control. Some of these directives are exposed to enable certain MATISSE features (e.g., parallelization), while others are designed to facilitate debugging and experimenting with MATISSE (e.g., measure the impact of disabling certain optimization passes).

Let us start by describing the parallelization directive system. The goals when designing this approach were the following:

1. *Preservation of MATLAB semantics*: MATLAB (or MATLAB-compatible) runtimes shall be able to execute code annotated with directives, ignore them and still output the correct results.

2. *Ease of use*: The directive-based approach shall be as simple as possible, to assist the process of parallelizing legacy applications, to allow MATLAB programmers to understand annotated code even if they did not learn how to use our compiler, and to ensure the process of learning the directives is as simple as possible.

3. *Ability to control which code is offloaded*: Advanced users and other tools shall be able to modify the code to prevent certain regions from being parallelized. Similarly, there shall be compiler hints to allow MATISSE to parallelize code when it can not determine that doing so is correct. These hints may be provided by analysis tools.

4. *Integration with common MATLAB idioms*: The directive system is designed to extend, not replace, existent MATLAB idioms. MATLAB code can both be very high-level (matrix-based computations), very close to the equivalent C (loop-based computations) or a mix of both. The compiler shall be compatible with all of these code styles.

At first, an OpenMP/OpenACC-style directive system, where users annotate loops to parallelize (see Subsection 3.1), was considered. While this style fits properly with loop-based code, it was difficult to adapt it to matrix-based code styles, where no explicit loops exist. An API-based approach such as GPUmat [GP-12] would be similarly difficult to adapt to loop-based computations. As both programming styles are common among MATLAB programmers, neither approach would be optimal.

Instead, this thesis opts to use directives to identify code sections that *contain* code to parallelize, even if mixed with sequential code. The compiler searches for `%!parallel` and `%!end` comments and conservatively attempts to parallelize code within those regions. If a `%!parallel` directive is added on top of a function, then the entire function body is considered for parallelization. Code within these annotated regions that can not be parallelized, either because there is no parallel implementation or because the compiler could not determine the correctness of the parallelization strategy, is still executed sequentially, in C code.

Table 4.2: List of main MATISSE directives

Directive	Applies to	Description
`%!parallel`	Function, Section	Indicate that a section or function should be parallelized.
`%!end`	Section	Mark the end of a `%!parallel` section.
`%!serial_dimension`	Loop	Indicates that the loop should not be a kernel's parallel dimension.
`%!no_index_overlap`	Loop	Indicates that the loop may be parallelized, even if MATISSE could not prove lack of data dependencies.
`%!by_ref <arg>`	Function	Indicates that an argument should be passed by reference.
`%!assume_indices_in_range`	Function	MATISSE may assume that all matrix accesses are in-range.
`%!assume_matrix_sizes_match`	Function	MATISSE may assume that all functions with matrices have matching (non-scalar) sizes for both operands.
`%!export <name>`	Function	Indicates the final C name of the function, and disables any name mangling. On Windows, MATISSE also marks export-annotated functions as DLL exports. If `name` is not specified, then MATISSE uses the MATLAB name as the final C name.
`%!disable <opt>`	Function	Disable an optimization or optimization setting on a given function.
`%!infusible`	Loop	Prevent loop from being fused with other loops.

A summary of the directives supported by MATISSE is presented in Table 4.2.

Figure 4.1 shows how to use the `%!parallel`/`%!end` directives. In this example, MATISSE attempts to offload the `t1` (matrix-based) and `t2` (loop-based) computations on the GPU, but the final `y` computation is executed on the CPU, since it is outside the parallel region.

Programmers can further control the parallelization of the program by using the `%!serial_dimension` directive. The directive can be applied to loops (generally nested) and indicates that the annotated loop level does not correspond to a parallelized kernel dimension. That is, either the loop is executed sequentially, or it appears as an explicit loop in the OpenCL kernel. The `%!no_index_overlap` indicates that no loop iteration modifies a matrix position that is accessed by any other loop iteration. This is generally best used to deal with limitations of MATISSE's loop carried dependency analysis, and allows the compiler to offload code it would otherwise not be able to.

The compiler currently makes no attempt to determine the profitability of the parallelization of code sections and assumes that any loops it finds within a parallel region

```
function y = directive_example(A)
    %!parallel
        t1 = A .* 2;
        t2 = zeros(1, numel(A));
        for i = 2:numel(t2),
            t2(i) = t1(i - 1);
        end
    %!end

    y = t2 .* 2;
end
```

Figure 4.1: MATLAB program demonstrating the use of the `%!parallel` directives.

should be offloaded. Additionally, the compiler does not currently parallelize any built-in matrix functions that are not converted into a loop by optimization passes.

The `%!parallel` directive may take the following optional parameters:

- `local_size`: controls the homonymous OpenCL parameter.
- `schedule`: enables the programmer to force certain mappings between loop iterations and OpenCL work-items. A more in-depth description of schedules is given in Section 5.5.

Although the `%!parallel` directive supports a few more optional parameters, they are currently only used for testing/benchmarking purposes.

A fully automated approach where no directives are specified would also meet most of the requirements. However, the ability to control which code regions are offloaded is important to allow experiments with multiple options. A second reason for choosing directives is that this thesis does not propose schemes heuristics to determine which sections to offload. For this reason, a fully automated approach would likely result in substantial slowdowns across multiple benchmarks. As an alternative, the compiler relies on the MATLAB programmer to do so manually.

MATISSE also includes directives unrelated to parallelization. The most important is `%!by_ref`. Using this directive, programmers identify which function arguments should be passed by reference (see Section 5.4).

Additionally, directives such as `%!assume_matrices_in_range` and `%!assume_matrix_sizes_match` improve program performance by allowing MATISSE to ignore certain MATLAB edge cases. Combined, these two directives form what is herein called the *Unchecked Mode*.

Finally, certain optimizations can be disabled using `%!disable`. Although it is not expected that MATISSE users rely on this directive, it proved to be very valuable for debugging and to measure the impact of certain optimizations. On a similar note, the `%!infusible` directive indicates that the specified loop should never be fused (see Subsection 5.1.2) with any other loop.

4.1.3 Auxiliary LARA Files

As mentioned in Section 3.1, MATISSE supports MATLAB-to-MATLAB compilation using a domain-specific programming language called LARA [BPN+13, CBP+16]. Although MATLAB-to-MATLAB compilation is outside the scope of this thesis, the proposed MATLAB-to-OpenCL compiler is still compatible with LARA and this DSL is used internally, for two purposes:

- Its MATLAB-to-MATLAB capabilities are integrated with the OpenCL generation framework: LARA can be used to generate a modified MATLAB program (e.g., with added directives) and compile the LARA-generated MATLAB instead of the original source code. This use-case has been mostly used to automate certain parts of the tests.

- LARA is used to specify *target-aware properties* of each target device, such as conditions for certain optimizations to be considered profitable. This use-case is transparent to users of MATISSE, though it does allow advanced users to create *custom recipes* to target devices MATISSE does not know about.

However, users of MATISSE can safely ignore LARA, unless they specifically need its features.

4.2 Compiler Phases and Intermediate Representations

The MATISSE C and OpenCL backends are split into multiple phases, and most of them are shared by the two backends. This section describes these phases, as well as the Intermediate Representations (IRs) used.

Figure 4.2 shows the simplified compilation pipeline of the OpenCL backend. The only difference for the C backend is that the OpenCL code generation is missing, so the C code generator handles all the typed IR. The compiler starts by parsing the MATLAB source code to produce the Abstract Syntax Tree (AST), then converts it to a Static Single Assignment (SSA) IR that the compiler uses for type inference, optimization and finally code generation. Not shown in Figure 4.2 is a short set of transformation passes that are applied to the AST before SSA IR construction, and a final IR called the *C IR*, a final AST that maps C language constructs. Rather than generating C code directly, the C/OpenCL code generators produce this C IR representation. The compiler performs a series of final *cleanup passes* on the C IR and finally translate it to C/OpenCL code.

Note that the previous MATISSE version described in Section 3.1 (including MATISSE CL V1) does not follow the same flow. This thesis extends the original MATISSE with the SSA-based IR, type inference and optimization framework [RBC16] described in this section. However, the parser and some other components (e.g., parts of the C code generator) have been reused.

4.2.1 Parsing MATLAB

MATLAB is a programming language that tends to be very difficult to parse correctly, and the inexistence of a formal MATLAB grammar contributes to some of the difficulties. MATISSE's parser [BPN+13] has been in development since before this work started, and it has been gradually improved over time.

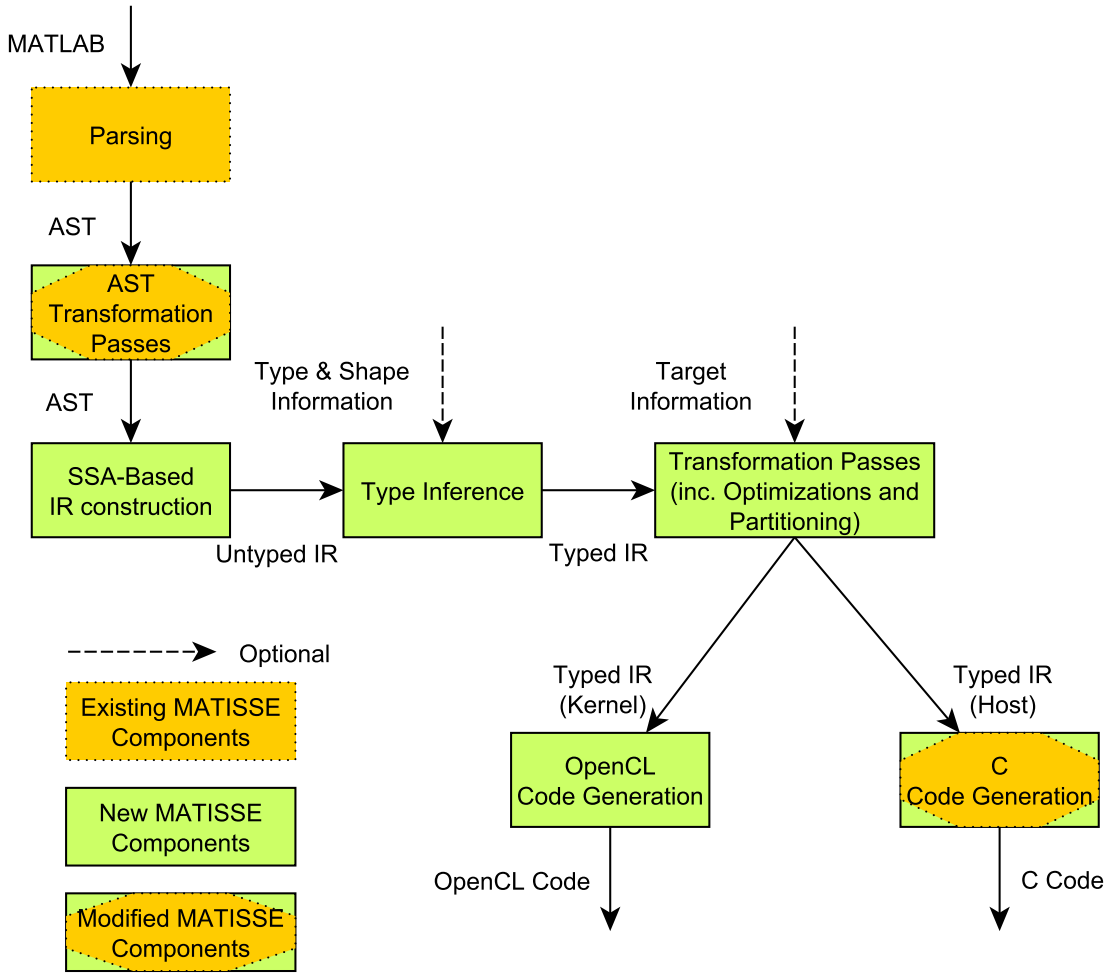


Figure 4.2: Overview of the MATISSE compiler phases, including the C and OpenCL backends.

At the moment, the parser is already capable of handling a wide range of difficult cases, including distinguishing matrix transposal operations (e.g., `A'`) from character arrays (e.g., `'A'`), and being able to properly process matrix elements (e.g., `[1 + 2]` is a matrix with a single element `3`, but `[1 +2]` has two elements, `1` and `+2`) and recognition of escaped single quotes in character arrays (e.g., `'a''b'`).

The MATISSE parser generates a custom AST that can be then analyzed and manipulated. Figure 4.3 shows a simple MATLAB program and its equivalent AST representation. Every MATLAB file has a *File* root node. In this case, this is a file containing user functions, so the root node is a *FunctionFile*. The first child of every function node is the *function declaration*, which includes information about the function name, inputs and outputs. The remaining children are the top-level (i.e., excluding those contained in other blocks) statements, in the program order. Statements such as loops and branches are represented as a *block* node. The first child of the block is the *block header*, which indicates the type of block (a `for` statement, in this case) and associated data. The remaining children of the block are the block statements. Note also that comments are treated as statements for AST purposes, and as such remain in the tree.

```
function y = f(x)
    y = zeros(1, x);
    for i = 1:x,
        y(i) = i * i;
    end
end
```

(a) Simple MATLAB program.

```
FunctionFile
    Function
        Statement: FunctionDeclaration (line 1)
            Outputs
                Identifier: y
                Identifier: f
            FunctionInputs
                Identifier: x
        Statement: Assignment (line 2;)
            Identifier: y
            SimpleAccessCall
                Identifier: zeros
                MatlabNumber: 1
                Identifier: x
        Statement: Block (line 5)
            Statement: SimpleFor (line 3)
                Identifier: i
                Operator: :
                    MatlabNumber: 1
                    Identifier: x
            Statement: Assignment (line 4;)
                SimpleAccessCall
                    Identifier: y
                    Identifier: i
                Operator: *
                    Identifier: i
                    Identifier: i
```

(b) Equivalent AST representation, in textual form.

Figure 4.3: Example of MATLAB element-wise expression and equivalent loop code.

4.2.2 AST Transformation Passes

Once the program AST has been built, MATISSE performs a number of transformations to simplify the next phases, particularly the SSA construction.

The transformations include:

- *Return Removal*: MATISSE's SSA IR does not use *return statements*, and every function returns only once, at the end. To deal with this discrepancy, we transform the AST of programs with return statements into the equivalent without returns.

- *Operator Replacement*: MATLAB defines a set of operators that have equivalent function calls (e.g., `plus(A, B)` for `A + B`). To avoid redundancy in later phases, both formats are normalized to use the function call form. The short-circuiting operators (i.e., `||` and `&&`) are not converted, as there does not appear to be a function that replicates these semantics, and there does not seem to be a way to implement short-circuiting logic in function arguments.
- *Matrix Replacement*: Normalizes matrix construction by replacing matrix expressions into the equivalent `horzcat`/`vertcat` function calls[1].
- *While Loop Simplification*: Simplifies `while` loops, so that the loop condition is always `1`, and instead code is injected at the beginning of the loop to check the condition and `break` the loop if it is false.

After these passes have been executed, the AST is in a *normalized* form, ready to be converted to the SSA IR.

4.2.3 Matrix-Based SSA IR – The Sequential Case

Between the AST and the code generation stage, an SSA-based IR is used by MATISSE to perform most transformations.

This section describes this IR in detail, including how MATISSE constructs it, and the parallelism-specific extensions.

SSA IR Description

Previous versions of MATISSE [BRC15b] had two intermediate representations: the MATLAB-based AST described in Subsection 4.2.1 and the C IR described in Subsection 4.2.7. The MATLAB IR was directly converted to C IR statement by statement, and type and shape inference occurred during this process. This limited the scope of many analysis and transformations, including type and shape inference. This is undesirable because one often wants to apply function-wide optimizations or code transformations that rely on type information but for which the C IR is too low-level.

To solve this problem, this thesis proposes a new IR for MATISSE [RBC16], with SSA semantics [RWZ88]. This SSA IR is used between the MATLAB IR and the C IR. The type inference mechanism, the optimizations and the C IR generator are now based on the SSA IR. Since it is built before type inference, it has both typed and untyped variants.

Figure 4.4 demonstrates the SSA code for the MATLAB function in Figure 4.3a, as generated by MATISSE before type inference has been executed. For simplicity, the code shown already has been processed by some initial transformation passes, notably dead code elimination, to remove unnecessary boilerplate code that is generated by MATISSE.

In this SSA representation, each function is divided into *blocks*. A *block* is a sequence of *instructions* that are executed one at a time. Blocks are identified by a block ID, with the # prefix. Transformations may safely assume that no control flow occurs within a

[1] There is an exception to this. If an empty matrix expression (i.e., `[]`), optionally inside parenthesis, is the right-hand side of an assignment, MATISSE does not perform the replacement. These are removal statements and replacing the `[]` literal with an expression returning an empty matrix changes the behavior of the program. MATISSE does not currently support removal statements, but it was still decided to disable the transformation in those cases, as it would be incorrect.

```
1  Function f                                                    % function y = f(x)
2  block #0:
3      line 1
4      x$1+2+3 = arg 0
5      line 2
6      $zeros_arg1$1 = 1                                         % y = zeros(1, x);
7      y$2 = untyped_call zeros $zeros_arg1$1, x$1+2+3
8      line 3
9      $start$1 = 1                                              % for i = 1:x,
10     $interval$1 = 1
11     for $start$1, $interval$1, x$1+2+3, #1, #2
12 block #1:
13     line 3
14     y$3 = phi #0:y$2, #1:y$4
15     i$2 = iter
16     line 4
17     $mtimes$1 = untyped_call mtimes i$2, i$2                  % i * i
18     y$4 = set y$3, i$2, $mtimes$1                             % y(i) = ...
19 block #2:
20     line 5
21     y$ret = phi #0:y$2, #1:y$4                                % end
22     line 6
```

Figure 4.4: MATISSE SSA code for the example in Figure 4.3a, before type inference has been applied. The code at the right side indicates which parts of the original MATLAB code generated the given SSA code, and is *not* part of the SSA representation MATISSE uses.

block, as all control-flow-related instructions (e.g., `for` and `break`) can only be present as the very last instruction of their block.

In this IR, control flow is still fairly high-level, as concepts such as *if statements* and *for loops* still exist (with the `branch` and `for` instructions, respectively). An instruction such as `for A, B, C, #D, #E` (line 11 of Figure 4.4) means: perform a ranged `for` loop (with the MATLAB semantics for the interval `A:B:C`, so starting at A, ending at C, with an interval B). The body of this loop starts at block `#D`, and after the loop is completed proceed by executing block `#E`. The iteration variable of the loop is given by the `iter` instruction (line 15 of Figure 4.4).

At the end of a block, control-flow does not *automatically* proceed to the next block. Instead, the current control flow context (e.g., `for` loop body) ends. If there are no control flow contexts pending (as is the case at the end of `block #2`), the execution of the function ends and the function returns the variables ending with `$ret`.

As with any SSA representation, every variable can only be assigned at one point of each function [SJGS99], so every MATLAB variable assignment must result in a new SSA variable being created. The proposed IR goes one step further: even *indexed* assignments (e.g., `A(1) = 2;`) result in a new SSA variable being created.

If a variable's value diverges in a branch or loop, the appropriate value can be obtained with a ϕ (phi) operation. For instance, in `$out = phi #1:$value1, #2:$value2`, the value of `$out` is `$value1` if the control flow came from block `#1`,

and `value2` if the control flow came from block `#2`.

The proposal attempts to make the instructions very simple, so complex expressions (such as `A + B .* C`) must be dealt with by introducing temporary variables. By convention, SSA variables starting with a `$` (e.g., `$A$1`) are compiler temporaries, whereas SSA variables starting with an identifier (e.g., `A$1`) correspond to variables that exist in the original MATLAB program.

There are no dedicated operators for `&&` and `||`. Instead, MATISSE injects an if/else statement (as a `branch` instruction) to implement the semantics of the short-circuited operators.

Appendix B presents the list of MATISSE SSA instructions and their semantics.

The proposed SSA representation includes both high-level instructions and low-level ones. The reason for this is that this SSA IR is produced from a nearly 1-to-1 representation of MATLAB (the AST), but is used all the way to a level very close to C.

Functions and `for` loop instructions can be annotated with properties to store additional information. Some directives (e.g., `%!serial_dimension`) are implemented using this feature.

Construction of the SSA Representation

As previously mentioned, the SSA IR is constructed from a transformed AST representation. This is performed one function at a time, even in files with multiple functions. It is also in this stage that MATISSE processes *directives* (see Subsection 4.1.2).

MATISSE starts by traversing the AST to determine which identifiers refer to *variables* are which are *functions*. In MATLAB, functions can be called without using parentheses (when the function call takes no arguments) and variables can have the same name as functions, so this step is fundamental to properly process the code. MATISSE determines that if an identifier is a variable at *any* point of the function, then it is a variable at *all* points of the function, and any identifier that does not refer to a variable must refer to a function. In order to perform name resolution, the compiler uses the following queries:

- Is it an argument or returned value of the function that is currently being processed? If so, then it is a variable.
- Is it used to store the result of an assignment (such as `A = B`, `A(i) = B` or `for i = 1:N`)? If so, then it is a variable.
- Is it used in a global declaration statement? If so, then it is a global variable. However, global variables are outside the scope of this thesis.
- All other identifiers refer to functions.

Notably, identifiers referenced by directives are not recognized as variables unless they are defined somewhere else in the function, because in the AST directives appear as normal comments.

Given this, MATISSE generates, at the start of block `#0`, a *stub* definition for all variables. Arguments are assigned using the `ARG$1 = arg <N>` instruction, and other variables are given *undefined* values using the `VAR$1 = !undefined` instruction. Many of these stub definitions are later removed by dead code elimination.

The compiler then constructs the initial *block context*. Block contexts store information necessary for SSA construction that may change at multiple points of the program, such as:

- Which SSA variables contain the most recent values of each MATLAB variable at that point of the program (initially, the compiler uses the aforementioned stub definitions);
- Current program line;
- Information about the matrix accesses that the SSA builder is currently traversing (e.g., to know that in `A(sqrt(end))`, `end` is used as the only index of an access to A);
- Information necessary to determine the target of `break` and `continue` statements;
- Active SSA block (initially, it refers to block `#0`).

MATISSE processes the tree nodes in depth-first order to generate the equivalent IR statements. At the end of each branching point (i.e., `if`, `while` or `for` blocks), MATISSE reads the data from the block contexts resulting from each possible source and produces ϕ nodes, if necessary.

When a comment is processed, MATISSE analyses it to determine whether it matches any known directive, and generates code as appropriate for it.

Finally, when MATISSE reaches the end of a function, it produces the `$ret` SSA variables, which contain the final values of all returned MATLAB variables.

4.2.4 Type Inference

MATLAB is a dynamically typed language. However, both C and OpenCL are statically typed. This means that, to convert MATLAB to C/OpenCL, MATISSE has to discover the types and shapes of MATLAB variables. This operation is known as *type and shape inference*.

Accurate and precise type and shape inference is important because many MATLAB operations have different semantics depending on the types of the inputs, so knowing which one should be used helps reducing the number of runtime validity checks or generating specialized optimized versions of MATLAB functions. For instance, the operation `sum(X)` can be implemented without memory allocations if X is known to be a row or column matrix.

MATISSE currently performs type inference at the SSA level, so each MATLAB variable can have different inferred types at different points of the program, as long as each SSA variable only has one. Algorithm 4.1 shows an overview of the type inference algorithm and its handling of some simple instructions. For cases without control flow, the type inference functions in a very straightforward manner, by visiting one instruction at a time.

Branching instructions are also relatively simple. MATISSE first inspects the condition to determine whether it is known to be true, known to be false, or neither. Based on this, MATISSE may avoid having to visit the *then* or the *else* case. If all possible taken branches intercept executions (i.e., `markUnreachable`), then the `branch` itself intercepts the point after the branches and type inference does not run on the *end*

```
function inferFunctionTypes:
    Input: currentFunction, argumentTypes
    Output: types
    rootContext ← newRootContext(argumentTypes)
    inferBlockTypes(rootContext, currentFunction, 0)
    types ← rootContext.types
end
function inferBlockTypes:
    Input: context, function, blockId
    instructions ← getFunctionBlock(function, blockId)
    foreach instruction ∈ instructions do
        inferInstructionTypes(instruction, context, function)
        if ¬isReachable(context) then
            break foreach
        end
    end
end
function inferInstructionTypes:
    Input: instruction, context, function
    if instruction.type = assignment then
        type ← getVariableType(context, instruction.input)
        addVariableType(context, instruction.output, type)
    else if instruction.type = φ then
        if hasPhiSource(context) then
            sourceVarName ← getSourceFrom(instruction,
              getPhiSource(context))
            type ← getVariableType(context, sourceVarName)
        else
            type ← getCombinedType(getAllSourceTypes(context, instruction))
        end
        addVariableType(context, instruction.output, type)
    else if instruction.type = continue then
        markContinuePoint(context, instruction.location.blockId)
        /* Statements after continue (not in the same block)
           are not executed.                                    */
        markUnreachable(context)
    else
        /* Rules for other instruction types                   */
        ...
    end
end
```

Algorithm 4.1: Overview of the type inference algorithm. Many instruction types, notably related to control-flow, have been omitted for brevity.

block. Otherwise, type inference proceeds to the end block. Note that if only one of the cases reaches the end without being intercepted, then the type inference takes that into account when inferring the types of the phi nodes that merge the `if/else` values.

The most complicated case for type inference occurs in loops, shown in Algorithm 4.2.

The concern that MATISSE deals with is that the types of variables may change each iteration, so MATISSE repeatedly performs type inference on the loop body until the inferred types stabilize. The compiler does this by storing the partial inferred information in an object called the *loop information sink*. Every time a continue statement is found, this information is added to the sink, which combines the inferred types and, if any change is found, takes note of the need to rerun the type inference algorithm. To start the algorithm, MATISSE treats the beginning of the loop as a *continue* point, coming from the block containing the `for`/`while` instruction.

```
function inferForInstructionTypes:
    Input: instruction, context, function
    loopSink ← new LoopInformationSink(LoopType.For)
    doContinue(loopSink, instruction.location.blockId, ∅)
    while hasPendingBlocks(loopSink) do
        sourceBlock ← nextPendingBlock(loopSink)
        types ← getVariablesStartingFrom(loopSink, sourceBlock)
        loopContext ← newWhileContext(context, sourceBlock, loopSink, types)
        inferBlockTypes(loopContext, instruction.loopBody)
    end
    if isReachable(context) then
        addBreakPointTypesToContext(loopSink, context)
        inferBlockTypes(context, instruction.endBody)
    else
        /* Infinite loop.                                          */
        markUnreachable(context)
    end
end
function doContinue:
    Input: loopSink, sourceBlockId, newTypes
    currentTypes ← getVariablesStartingFrom(loopSink, sourceBlockId)
    changedTypes ← combineTypes(currentTypes, newTypes)
    if changedTypes then
        addPendingBlock(sourceBlockId)
    end
    setVariablesStartingFrom(loopSink, sourceBlockId, changedTypes)
    if loopSink.loopType = LoopType.For then
        /* For loops may end at any continue.                      */
        doBreak(newTypes)
    end
end
```

Algorithm 4.2: Type inference algorithm to deal with `for` instructions.

This type inference mechanism has some limitations. Most notably, because type inference for each function call is performed by inferring the types of the function (based on the input arguments), MATISSE does not support recursive functions.

In practice, this approach was sufficient for solving most of the considered MATLAB code.

As an example, consider the SSA function presented in Figure 4.4, which represents a computation in a simple `for` loop. Table 4.3 indicates the order the blocks are

processed, and displays the resulting types of some function variables as an example. MATISSE first processes the entirety of block `#0`, until it reaches the loop. Once it reaches the loop, it uses the types of block `#0` as some of the possible types of block `#2` (as the loop may have 0 iterations), and processes block `#1` as *coming from* `#0`. Since, at this point, block `#1` can only have come from `#0`, the `y$3 = phi #0:y$2, #1:y$4` instruction copies the type of `y$2` to `y$3` and ignores `y$4`. The rest of the loop is processed normally until the end is reached. At the end of the loop, an implicit `continue` statement is passed, meaning that MATISSE will use the types at the end of the first loop iteration as some of the possible types of block `#2`, and repeat the type inference for the loop, this time as *coming from* `#1`. MATISSE infers the types for block `#1` once again, but this time when it reaches `y$3 = phi #0:y$2, #1:y$4`, it uses the type of `y$4` (inferred from the previous iteration) instead of the `y$2`. At the end of the loop, MATISSE once again uses the inferred types as some of the possible types of block `#2` and compare the results of the type inference with those obtained in the previous iterations. Since they are the same, MATISSE stops inference for block `#1`, and proceeds to block `#2`. Finally, MATISSE performs type inference on block `#2`.

Table 4.3: Order in which the type inference algorithm processes blocks, and results of some of the types inferred at the end of those blocks, for the example in Figure 4.4.

Block	Iteration	Source Block	`y$2`	`y$3`	`y$ret`
#0	N/A	N/A	double[1, ?]	undefined	undefined
#1	1st	#0	double[1, ?]	double[1, ?]	undefined
#1	2nd	#1	double[1, ?]	double[1, ?]	undefined
#2	N/A	N/A	double[1, ?]	double[1, ?]	double[1, ?]

4.2.5 SSA Transformation Passes

MATISSE uses its SSA IR to run a wide range of transformation passes. In total, more than 100 passes are executed, though some of these are repeated. Most of the passes are applied after type inference, as type information is critical for high-quality optimization, and some passes are applied multiple times.

Before the compiler passes are executed, the SSA IR code very closely resembles the original MATLAB code, using high-level instructions such as matrix-operations. Many of these passes translate the high-level unoptimized operations into gradually more low-level optimized equivalent. By the end of the compilation stage, the IR instructions are very close to the C equivalent.

It is also in the SSA transformation stage that the code patterns for parallelization are detected, and computations are offloaded to one or more kernels.

Specific SSA passes are discussed in more detail in Subsection 4.2.6 and in Chapter 5. A list of all available SSA passes and their default order of execution is available in Appendix D.

4.2.6 Parallelization

MATISSE is capable of performing auto-parallelization, based on instructions provided by directives. Overall, this is done by the following stages:

- *Parallel Region Identification*: Identifies sections marked by the user as parallel (either function-wide, or ending in a `%!end` directive) and moves them to a custom code block. It is executed before most other passes, including sequential optimizations.
- *Parallel Section Extraction*: Extracts the parallel regions to a separate IR container, in a manner similar to outlining. It is executed before most other passes, including sequential optimizations.
- *Parallel Section Analysis*: Analyzes parallel sections and extracts loops that should be offloaded to *kernels*. When doing so, MATISSE also naively introduces data transfers to ensure correctness, without concern for efficiency. Because this phase only applies to parallel sections, all identified loops are known to be in a section marked as `%!parallel`.
- *Recombination*: MATISSE moves the code from the parallel section back to the containing function, now without the offloaded loops but instead with kernel calls and data transfers, in a manner similar to inlining.

MATISSE's parallelization strategies are based on `for` loops. The compiler searches for `for` loops within parallel sections, verifies whether they follow a certain set of criteria and, if so, offloads them. It also supports parallelization along multiple loops (i.e., *loop nests*).

Consider the example in Figure 4.5. To determine whether a group of loops can be offloaded to OpenCL, the tool searches for potential loop-carried dependences in `phi` instructions (e.g., `y$3` is used to build `y$4`).

```
1  block #0:
2      % y$2, A$1, B$1 defined outside parallel region
3      $start$1 = 1
4      $interval$1 = 1
5      $end$1 = call numel [INT] X$1
6      for $start$1, $interval$1, $end$1, #1, #2
7  block #1:
8      y$3 = phi #0:y$2, #1:y$4
9      % y$3 used to build y$4
10     % No other loop-carried dependences.
11     $i$2 = iter
12     $A_value$1 = simple_get A$1, $i$2
13     $B_value$1 = simple_get B$1, $i$2
14     $y_value$1 = call plus [DOUBLE] $A_value$1, $B_value$1
15     y$4 = simple_set y$3, $i$2, $y_value$1
16     % y$4 constructed safely
17 block #2:
18     y$5 = phi #0:y$2, #1:y$4
```

Figure 4.5: SSA code for parallel region of function computing vector add, annotated with information about loop-carried dependences.

For each potential dependence, the compiler tracks how it is used and checks if it meets the requirements of known safe *reductions*. A loop can only be parallelized if *all*

potential loop-carried dependences can be safely handled. In addition, if there is any operation that is unsupported by the OpenCL backend (such as a matrix allocation), then the loop is not offloaded. The compiler treats matrix set operations as a form of reduction. For a matrix set reduction to be valid, an additional number of conditions must be met:

1. All matrix access operations within the loop must be known to be in-bounds. Notably, this means that there can be no matrix resizes. In practice, this means that all matrix accesses must be through the `simple_set`, `simple_get` or `get_or_first` SSA instructions (see Appendix B).

2. The matrix can only be used for size operations (e.g., `numel`), gets or sets;

3. The final reduction variable must be constructed from the initial reduction variable. A is constructed from B as `A = simple_set B, ...`, possibly with other intermediate variables. Once a variable is used to construct another variable, it must no longer be referenced (though in a branch, both cases can each reference the same variables). One of the implications of this is that all SSA variables for a matrix reduction can be combined into the same final matrix variable.

4. There must be no read-after-write, write-after-read or write-after-write dependences across loop iterations between any accesses to the matrix.

Other forms of reductions must also fit a condition equivalent to 3 (but with different formats of construction).

In order to evaluate whether any data dependence described in (4) occurs, the compiler uses the Z3 SMT solver (see Section 2.2). For every pair of potentially problematic accesses, it verifies if the following condition is possible:

$$\bigvee_i L_i \neq M_i \wedge \bigwedge_i A_i = B_i \tag{4.1}$$

Where L and M represent the list of loop nest indices for two iterations and A and B are the lists of access indices. If the condition is true, then there is a data carried dependence. If A and B are not the same size, then the compiler conservatively assumes that there might be a data dependence. In other words, two different loop iterations cannot access the same position of the matrix, unless both are matrix reads. The programmer may add the `%!no_index_overlap` directive to a loop to explicitly indicate that no iteration writes to a position accessed by another iteration, in which case MATISSE assumes that the parallelization is correct without using Z3. Note that if multiple nested loops are intended to be parallelized as dimensions of a kernel, then the directive must be added to all of them. Similarly, the `%!serial_dimension` prevents a loop nest from being parallelized along that specific dimension. Thus, if all loops in a loop nest are marked with this directive, MATISSE does not parallelize that nest.

Currently, the only forms of reductions MATISSE supports are matrix sets and loop carried sums/subtractions, but the compiler has been designed so that it is simple to add other forms of reductions and patterns (e.g., min, max).

The parallelization of loop nests often has the problem that some parallelization options are mutually exclusive. For instance, consider the program in Figure 4.6. Both loops can be parallelized: the outer loop has no loop-carried dependences, the inner loop has a trivial reduction. However, parallelizing one prevents the parallelization of the other. In these scenarios, MATISSE prefers to parallelize the outermost loop nest.

```
1  %!parallel
2  function y = f(A)
3      y = zeros(1, size(A, 2));
4      for i = 1:size(A, 2),
5          acc = 0.0;
6          for j = 1:size(A, 1),
7              acc = acc + A(j, i);
8          end
9          y(i) = acc;
10     end
11 end
```

Figure 4.6: MATLAB program with two mutually exclusive parallelization strategies.

Once MATISSE has obtained a list of loop nests to parallelize, it verifies which of those loop nests (if any) consist of a single loop that perform a matrix set with the loop iteration as the access index and a constant value (e.g., `A(1:x) = k;`). If this is the case, MATISSE treats this as a parallel *ranged set*, as OpenCL has functions to implement this specific operation. Otherwise, the compiler treats this as a *kernel*, and generates the IR code accordingly.

After these operations are completed, MATISSE performs additional transformations, mostly in the form of data transfer optimizations.

4.2.7 Code Generation

In order to generate C/OpenCL code from its SSA IR, MATISSE first converts the SSA IR (at this point using only fairly low-level instructions, because high-level idioms have been transformed in the previous stages) to a custom C-like AST structure (the *C IR*), and then generates the C code from the C IR. This subsection describes how this conversion is done.

Final Variable Allocation

There are two main difficulties in translating the SSA IR code to C: C variables are not single-assignment, and in fact directly expressing them as such can be highly inefficient (e.g., matrix variables would need to be repeatedly copied on each assignment), and C has no equivalent to ϕ nodes.

To solve these issues, the compiler uses a standard out-of-SSA algorithm by Boissinot et al. [BDR+09], adapted to fit MATISSE's needs. The implementation skipped the parts of the algorithm designed to improve compilation performance because this stage is not the bottleneck of MATISSE.

In order to convert code out of the SSA representation, the compiler executes the following operations:

1. Convert the IR to Conventional SSA (CSSA) form, by adding parallel copy instructions, as described in Method I of Sreedhar et al. [SJGS99];
2. Determine which SSA variables should be grouped into a single C variable;
3. Generate the C variable name for each group of SSA variables;

An SSA program is said to be in CSSA form if, and only if, for all ϕ nodes, all ϕ inputs and the output can be assigned to the same final variable without changing program semantics. The CSSA property is important because, when all SSA variables in ϕ nodes are assigned to the same final variable, that ϕ node can be deleted (it becomes a *no-op* instruction).

In order to determine which SSA variables should be grouped together, the compiler computes each variable's liveness set and builds the interference graph, as described by Boissinot et al. [BDR+09], but with a key difference: their variable liveness analysis algorithm does not work for MATISSE. The outputs of an instruction may interfere with its inputs, and some variables are alive in points beyond their last usage.

In order to clarify the first problem, consider the MATLAB function in Figure 4.7. Assume that MATISSE correctly detects that the assignment is fully in-range and that X is a vector variable. The compiler implements these functions in C by passing the outputs as function arguments (pointers). So if X is passed as an argument of a given type `T`, Y is passed as an argument of type `T*`. Calls to `hazard1` are implemented in C as `hazard1(X, &Y)`. Any call to `hazard1` must assign X and Y to separate variables. If not, MATISSE would dynamically detect that the variable for Y is already large enough and refuse to allocate memory at the call to `zeros` (line 3). Writes to Y (line 4) would therefore also modify the elements of X, in violation of MATLAB semantics. In order to prevent this issue, on any call to a function implemented with outputs passed as function arguments, MATISSE considers that the lifetime of the input variables lasts for *at least* as long as the instruction, unless they are scalars or they are marked as `%!by_ref` (see Section 5.4).

```
1 % Assume that X is a vector (e.g., row matrix)
2 function Y = hazard1(X),
3     Y = zeros(1, numel(X));
4     Y(2:end) = X(1:end-1);
5 end
```

Figure 4.7: MATLAB function showing a variable liveness hazard in a function call instruction.

A second issue occurs in the `iter` instruction, used to obtain the current iteration of a `for` loop. MATISSE uses the output variable of the `iter` instruction to determine the name of the variable to use to store the loop iteration. Consider the example in Figure 4.8. Standard MATLAB semantics dictate that the loop must have 10 iterations, so the final value of y should be 10. However, in the equivalent C code, if `i` is used as the iteration value and `j` is assigned to the same C variable, then the loop will execute only once, producing incorrect results. To fix this issue, the lifetime of the `iter` output variable must last *at least* as long as the entire loop body.

From the interference graph, the compiler creates groups of SSA variables that should be assigned together. The primary concerns here are (1) to ensure that all variables in

```
function y = hazard2()
    y = 0;
    for i = 1:10,
        j = i + 10;
        y = y + 1;
    end
end
```

Figure 4.8: MATLAB code demonstrating variable liveness hazard in loop iterations.

ϕ nodes are assigned to the same final C variable and (2) if possible, reduce the number of matrix copies. MATISSE uses the following approach:

- First, it assigns all variables in each ϕ node to a single group, because this is required for correctness per Boissinot et al.'s algorithm [BDR+09];
- Then, it visits all instructions, and tries to assign the input and output SSA variables to the same C variable in assignments, parallel copies and `simple_set` instructions;
- It visits all instructions again, and tries to group together `%!by_ref` arguments and the corresponding output, the input and output of `complete_reduction MATRIX_SET` and `set` instructions, and the input and output Shared Virtual Memory variables of `invoke_kernel` and `set_gpu_range` instructions;

Grouping matrix variables is important because any matrix assignment instruction with different input and output matrices must be implemented with a matrix copy first. Grouping scalar variables is not a concern, because C compilers will perform their own highly-efficient register allocations [ALSU06].

Finally, MATISSE determines the name of final C variables based on the matching SSA variables. The mechanism to do so is shown in Algorithm 4.3.

SSA variables are split into two categories: *user SSA variables* that start with a letter character (e.g., `A$1`), representing SSA variables that were obtained from the MATLAB source code, and *temporary SSA variables* that start with a $ character (e.g., `$A$1`), that were generated by transformation passes. If a variable group has both user and temporary variables, MATISSE ignores the temporary variables and use only the user variables to determine the name of the final C variable. MATISSE obtains the *base names* (e.g., `A` for `$A$1`) of each variable, and combine them to produce the final name. As a special exception, for groups of temporary variables, MATISSE gives preference to more informative names, by prioritizing them over names that are known to be of little relevance (e.g., `one`).

C IR

The final compiler stages of MATISSE are performed on a custom C-like AST IR – the C IR. This representation supports the subset of C functionality that MATISSE requires, but with certain concepts that are still higher-level than C concepts. For any given C IR code, there is a direct C equivalent.

```
function chooseFinalName:
    Input: ssaVariables, allocatedNames
    Output: name
    candidates ← ∅
    hasUserVariable ← ∃var ∈ ssaVariables: isUserVariable(var)
    foreach ssaName ∈ ssaVariables do
        baseName ← getBaseName(ssaName)
        if isUserVariable(portion) ∨ ¬hasUserVariable then
            candidates ← candidates ∪ {baseName}
        end
    end
    hasGoodStrengthPortions
     ← ∃candidate ∈ candidates: ¬isLowStrength(candidate)
    if ¬hasUserVariable ∧ hasGoodStrengthPortions then
        candidates ← {candidate ∈ candidates|¬isLowStrength(candidate)}
    end
    /* At this point, candidates is never empty           */
    name ← join(candidates, "_")
    if startsWithDigit(name) then
        /* Ensure names do not start with digits          */
        name ← concat("x", name)
    end
end
```

Algorithm 4.3: Simplified algorithm to determine final name of generated C variables.

The representation is divided into top-level *instances* (e.g., functions, structs, and typedefs). Each instance can have a *declaration*, which is placed in a header file, and an *implementation*, which is placed in a C source file. Some instances have neither declarations nor implementations – they are the *inline functions*. An example of an inline function is the operator "A + B", which is represented in C IR as a function call with two arguments. Each instance can also have *implementation*, *definition* and *call* dependences. For instance, given a function `void func(tensor_d* arg)`, the instance that defines `tensor_d` is a *declaration dependency* of `func`, because if `tensor_d` is not included first then the declaration of the function is invalid.

Function definitions include:

1. A C function *name*, which is usually not the same as the name of the MATLAB function – we perform *name mangling* to allow each MATLAB function to be instanced multiple times (once per combination of input types and number of output variables);

2. The *file*, a string indicating the file name that contains the function, not applicable for inline functions;

3. A *return type*. For functions with no returned values or returning a single scalar value, the *return type* is clear: it is `void` or the type of the returned value. On other functions it is more complicated, as C does not support multiple returns and we want to let the function dynamically know if memory for its outputs has

already been allocated. To account for these cases, C IR includes the concept of *outputs-as-inputs*, function arguments that are pointers where MATISSE will store the results of the function. Figure 4.9 shows an example of using outputs-as-inputs to implement the MATLAB function `eye`.

4. Zero or more *arguments*. These include the inputs of the function, as well as any *outputs-as-inputs*.

5. The *documentation comment* is a `/** ... */` comment that appears before the function declaration, meant to be used by documentation parsers.

6. A *statement list*, containing the body of the function.

```
1 // Definition
2 tensor_d* eye_d_2(int dim_1, int dim_2, tensor_d** restrict t);
3
4 ...
5
6 // Call
7 eye_d_2(20, 20, &out1);
```

Figure 4.9: Example of outputs-as-inputs, with a call to function `eye(20, 20)`, returning the result in a variable named `out1`.

Figure 4.10 shows an example of a C IR representation. The function shown computes the sum of all elements of a matrix, and the C equivalent can be seen in Figure 4.11.

All C statements correspond to `InstructionNode` nodes, containing a single child: the specific type of statement, or a block node. Statements such as `if` branches are implemented as *blocks* (MATISSE always generates brackets on branches and loops). These blocks are statements with children statements. The first child of the block is the *block header*, which indicates the type of block and condition or expressions. C comments can be represented in one of two forms: documentation comments that are part of the function signature, or comments in the body that appear as if they were C statements. Note that functions such as `numel_alloc_d` and `get_tensor_inline_d_1` do not appear to be called in the C code, because they are C IR *inline functions*. The corresponding C expressions are `X->length` and `X->data[iter]`, respectively.

Code Clean-up

Once MATISSE has constructed the full C IR code, it performs a few simple *clean-up* operations on this IR to make the generated C code appear closer to what a C programmer would usually write:

- *Constant Propagation*: Finds cases where constant variables are given literal values and replaces uses of those variables. Useful because MATISSE generates variables for all numeric constants;

- *For Simplifier*: Attempts to change MATLAB-style loops (which typically start at 1) into C-style loops (which typically start at 0) by finding loops with the format `for (INT_TYPE i = 1; i <= N; ++i)`, where `i` is only used in expressions

```
Function test_tdd_undef_1:
    InstructionNode
        AssignmentNode
            VariableNode: X_numel
            FunctionCallNode numel_alloc_d
                FunctionInputsNode
                    VariableNode: X
    InstructionNode
        AssignmentNode
            VariableNode: y
            CNumberNode: 0.0
    InstructionNode
        BlockNode
            InstructionNode
                ReservedWordNode: for
                AssignmentNode
                     VariableNode: iter
                     CNumberNode: 1
                FunctionCallNode LessThanOrEqual
                     FunctionInputsNode
                          VariableNode: iter
                          VariableNode: X_numel
                AssignmentNode
                    VariableNode: iter
                    FunctionCallNode Addition
                        FunctionInputsNode
                             VariableNode: iter
                             CNumberNode: 1
             InstructionNode
                 AssignmentNode
                     VariableNode: y
                     FunctionCallNode Addition
                         FunctionInputsNode
                             VariableNode: y
                     FunctionCallNode get_tensor_inline_d_1
                         FunctionInputsNode
                             VariableNode: X
                             FunctionCallNode Subtraction
                                 FunctionInputsNode
                                     VariableNode: iter
                                     CNumberNode: 1
    InstructionNode
        ReturnNode
            VariableNode: y
```

Figure 4.10: C IR code for a function that computes the sum of all elements of an array.

`i - 1` and modifying the loop so that `i` starts 1 value before. Figure 4.12 shows

```
double test_tdd_undef_1(tensor_d* X){
    int X_numel;
    int iter;
    double y;

    X_numel = X->length;
    y = 0.0;
    for(iter = 1; iter <= X_numel; ++iter){
        y += X->data[iter - 1];
    }

    return y;
}
```

Figure 4.11: C code generated from the C IR code in Figure 4.10.

```
double test_tdd_undef_1(tensor_d* X){
    int X_numel;
    int iter;
    double y;

    X_numel = X->length;
    y = 0.0;
    for(iter = 0; iter < X_numel; ++iter){
        y += X->data[iter];
    }

    return y;
}
```

Figure 4.12: Code generated from the C IR code of Figure 4.10 after the For Simplifier clean-up pass.

C code generated after this pass has been applied to the code in Figure 4.10.

- *Short-Circuited Expression Builder*: Converts simple `if/else` statements with certain patterns of variable assignments into equivalent short-circuited logical operations.
- *While Condition Builder*: The code generator only produces `while(1)` conditions (the loop condition is checked with `if-break` statements). While this is valid, it is better to instead place the loop condition directly in the `while`, so this pass performs that conversion.
- *Elseif Builder*: Converts `else` regions consisting solely on an `if` statement into `else if` regions. In other words, it replaces `else { if (cond) body }` constructs with `else if (cond) body`.
- *Redundant Return Removal*: Removes empty return statements (i.e., `return;`)

at the end of functions.

- *Empty Else Elimination*: Eliminates empty else blocks.

In combination, these passes make the generated C code closer to what hand-coded C looks like.

C and OpenCL Code Generation

MATISSE generates C code by first converting its SSA IR into the C IR, and then converting the C IR to the C source code and header files. The compiler performs C IR generation on a per-function basis, one instruction at a time. After all C IR functions have been constructed, MATISSE creates the C *project*, by determining which files each function belongs to and adding the `#include` dependences, as necessary. This is a simple stage because the C IR has been designed to be trivially convertible to C.

OpenCL code generation uses the same strategy, except that it does not generate header files and writes to separate `.cl` source files. Figure 4.13 shows the generated OpenCL code for the example in Figure 4.1. The directive in line 3 is necessary to use double precision on older OpenCL versions, but causes warnings in recent versions. Thus, MATISSE adds lines 1 and 4 to conditionally determine when to add the directive. In this example, MATISSE generates two kernels: one for the `t1 = A * 2;` kernel, and one for the explicit `for` loop. The call to `zeros(1, numel(A))` could cause the generation of a third kernel (to fill the output matrix with zeros), but on AMD GPUs, MATISSE is configured to use the `clEnqueueFillBuffer` function instead. In this example, MATISSE is able to eliminate all bounds checking without any additional directives or compiler settings.

4.2.8 Overview

MATISSE uses multiple compilation stages to process the input MATLAB code and to generate the resulting C/OpenCL program. Figure 4.4 lists the main representations of code used by the compiler. It parses a subset of MATLAB and generates standard C/OpenCL code, but all the main IRs were designed and built specifically for MATISSE. MATISSE parses MATLAB code to an AST representation, from which an SSA IR is constructed. It is on this IR that type inference is executed. MATISSE then performs the bulk of the optimizations, before generating a C-like AST called C IR. Finally, MATISSE cleans the obtained C IR and generates the final C and OpenCL code.

4.3 Compiler Validation

In order to test MATISSE, a combination of manual and automated testing was used. The automated testing approach is primarily composed of 3 components:

1. A set of small unit tests designed to test specific features, in as much isolation as practical.

2. A set of larger tests (e.g., benchmarks) that test the compiler. These tests consist of MATLAB programs along with respective inputs and expected outputs. The programs are compiled with MATISSE and executed, and their results are compared with the expected values, to verify that MATISSE generates correct code.

```
#if __OPENCL_VERSION__ < 120
// Since OpenCL 1.2, the pragma is no longer necessary and in fact may trigger warnings.
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

__attribute__((reqd_work_group_size(128, 1, 1)))
__kernel void directive_hello_world_1_2(global double* t1, global double* t2, uint N_1)
{
   size_t global_id0_1;
   size_t global_size0_1;
   size_t group_id0_1;
   size_t local_id0_1;
   size_t local_size0_1;

   global_id0_1 = get_global_id(0U);
   global_size0_1 = get_global_size(0U);
   local_id0_1 = get_local_id(0U);
   local_size0_1 = get_local_size(0U);
   group_id0_1 = get_group_id(0U);
   if(global_id0_1 < N_1){
      int i;
      i = global_id0_1 + 1 + 1;
      t2[i - 1] = t1[i - 1 - 1];
   }

}

__attribute__((reqd_work_group_size(128, 1, 1)))
__kernel void directive_hello_world_1_1(global double* A, global double* t1, uint N_1)
{
   // Variable declarations/initializations omitted for brevity,
   // as they are the same as the ones used in the previous kernel.

   if(global_id0_1 < N_1){
      int iter;
      iter = global_id0_1 + 1;
      t1[iter - 1] = A[iter - 1] * 2;
   }

}
```

Figure 4.13: MATISSE-generated OpenCL code for the code in Figure 4.1, when MATISSE is configured to target a generic AMD GPU and use the direct schedule.

3. A tool that automatically builds MATISSE and runs the relevant tests whenever

Table 4.4: Overview of main languages and IRs used in MATISSE.

Code Representation	Type	MATISSE-specific	Operations
MATLAB Source Code	Code	Custom Subset	Parsing
MATLAB-based AST	AST	Yes	Pre-processing SSA IR construction
Untyped SSA IR	Block-based	Yes	Type Inference Transformations
Typed SSA IR	Block-based	Yes	Optimizations Parallelization C IR construction
C IR	AST	Yes	Clean-up Code generation
C / OpenCL Source Code	Code	No	MATISSE output

changes are committed to the source code repository, as well as periodically. This tool only covers the sequential C code generation.

In combination, these components help improve MATISSE by signaling regressions in the compiler.

4.4 Summary

In this chapter, the overall architecture of the MATISSE compiler was discussed, including the programming model users are expected to use, the supported features and directives, the main stages and IRs of the compiler and the validation of the compiler.

The compiler was built to process MATLAB code with as few modifications as possible, using a lightweight directive system for parallelization. Directives were added for advanced features, such as the ability to disable optimizations on specific functions, but MATISSE can be used without resorting to these features.

Table 4.5 lists the stages of the compiler, as well as their inputs and outputs. MATISSE parses MATLAB code to generate an AST, converts it into an SSA IR that is used for type inference and optimization, then converts the SSA IR into another tree-based representation called CIR and finally generates C/OpenCL code.

In order to validate the compiler, multiple small test cases were automatically executed whenever changes have been committed to the source code repository. In addition, the compiler was also validated using larger test cases, such as benchmarks.

Table 4.5: List of MATISSE compiler stages.

Compilation Stage	Input	Output
Parsing	MATLAB Code	AST
AST Transformation Passes	AST	AST
SSA IR Construction and untyped SSA passes	AST	Untyped SSA IR
Type Inference	Untyped SSA IR	Typed SSA IR
Transformation Passes	Typed SSA IR	Typed SSA IR
CIR Generation	Typed SSA IR	CIR
Cleanup Passes	CIR	CIR
Code Generation	CIR	C/OpenCL

5

Optimizations

Contents

This chapter describes the optimizations proposed and implemented in MATISSE in order to generate efficient code. This includes standard optimizations such as loop fusion, as well as several optimizations to deal with MATLAB semantics, and finally parallelization-related transformations.

5.1 Loop Conversion Passes

MATISSE includes a set of optimization passes designed to convert matrix operations into optimized loops. The advantages of these optimizations are twofold: generate efficient code with better cache locality and few branches and, more importantly, produce fewer auxiliary matrices, e.g., by not generating them at all, or by eliminating them with subsequent optimizations. Eliminating auxiliary matrices is important as it can substantially reduce memory usage of the programs, and reduce the number of memory access operations.

5.1.1 Element-wise Operation Elimination

The element-wise operation elimination consists of identifying *element-wise* operations and replacing them with the equivalent `for` loops.

A function call or operator is considered element-wise if it meets all the following requirements:

1. It has no side effects;
2. It has a single output variable R;
3. It receives input variables $A_1, A_2 ... A_n$ that are all matrices, where a scalar is considered a 1×1 matrix;
4. All non-scalar input matrices have the exact same size/shape, which is the size/shape of the returned variable. If all inputs are scalars, then the output is also a scalar;
5. The value of R_i is computed solely based on the value of the scalar inputs and the value at the same index of the non-scalar inputs. The function/operation to do so is the same as the operation to compute the entire matrix. For instance, in `A = B + C`, `A(i) = B(i) + C(i)`. The operation + is a function call because these operators have equivalent function calls in MATLAB.

Figure 5.1a shows a MATLAB function with a computation that could be performed in a single loop (see Figure 5.1b) instead of the shown matrix expression. If we consider the operation to be a function call, then the element-wise operation elimination pass has effect similar to inlining, albeit for a more targeted purpose.

The optimization searches for function calls that are known to be element-wise (e.g., `sin`) by traversing every SSA instruction in order and replaces them with:

- A memory allocation for the operation result;
- A loop iterating through all input elements, with its body reading the input elements, performing the same function call on the obtained scalars, and storing the result in the output matrix.

```
y = sin(A);
```

(a) Element-wise operation.

```
y = zeros(size(A));

for i = 1:numel(A),
    y(i) = sin(A(i));
end
```

(b) Equivalent loop version.

Figure 5.1: Example of MATLAB element-wise operation and equivalent loop version.

For element-wise operations involving more than one matrix, determining if one of them is a scalar at compile time can be a difficult problem. This issue is avoided by using the `get_or_first` SSA instruction (see Subsection 4.2.3). This instruction receives 2 inputs (a matrix A and an index i), and return A if the matrix is a scalar, or A_i if it is a non-scalar (in this case, i is assumed to be in range of the matrix). The loop code to access the matrix then just uses this instruction, without checking for size.

Figure 5.2 illustrates the result of the element-wise operation elimination pass (excluding code to verify that A and B have compatible sizes, at runtime).

```
block #0:
    % Omitted: Code to compute the size of y
    $y$1 = call matisse_new_array [DynamicMatrixType(DOUBLE, shape=[Matrix
    Shape: [], Dims: -1])] $size$1
    $numel_result$1 = call numel [INT] $y$1
    $start$1 = 1
    $step$1 = 1
    for $start$1, $step$1, $numel_result$1, #1, #2
block #1:
    $y$3 = phi #0:$y$1, #1:$y$2
    $iter$1 = iter
    $A_value$1 = get_or_first A$1, $iter$1
    $B_value$1 = get_or_first B$1, $iter$1
    $y_value$1 = call plus [DOUBLE] $A_value$1, $B_value$1
    $y$2 = simple_set $y$3, $iter$1, $y_value$1
block #2:
    y$2 = phi #0:$y$1, #1:$y$2
```

Figure 5.2: Generated SSA code for the `y = A + B` assignment after element-wise operation elimination.

Note that operations such as `sum(A)` are *not* element-wise operations, as they use multiple input values (i.e., matrix elements) to produce a single output scalar.

5.1.2 Managing and Optimizing Loop Generation

This subsection describes two techniques for managing and optimizing loop generation which impact the use of temporary matrices: *Loop Fusion* and *Direct Combined Element-Wise Loop Generation*.

Figure 5.3 shows a MATLAB function with a computation that could be performed in a single loop instead of the shown matrix expression. The translation of each subexpression (e.g., `A(:)`) to a loop is trivial, as these expressions are all *element-wise* (see Subsection 5.1.1). However, a direct translation results in 6 loops being generated in the final code and 5 temporary matrices are needed to store the results of the subexpressions. A more efficient code generation scheme would recognize that the example can be compiled to a single efficient loop without any memory allocation. In order to achieve this, the element-wise operation elimination pass can be modified to directly combine multiple subexpressions when transforming the code (this approach was named the *Direct Combined Element-Wise Loop Generation*), or the compiler can later combine multiple generated loops into a single one using *Loop Fusion*. In MATISSE, a mix of both approaches is used to achieve the best results.

```
1 %!assume_matrix_sizes_match
2 function y = example(A, B, C)
3     y = sum(A(:) .* B(:) .* C(:));
4 end
```

Figure 5.3: MATLAB program that benefits from temporary matrix elimination.

Loop Fusion

Loop Fusion [KA01], also known as *Loop Merging*, is an optimization that combines multiple loops into a single one. Although loop fusion itself does not eliminate temporary matrices (i.e., there is no single explicit *scalar replacement* pass), it exposes opportunities for a subsequent optimization passes (notably *read after write elimination* and *dead code elimination*) that are capable of doing that.

The MATISSE Loop Fusion pass has two separate *modes* of operation:

1. *Standard Loop Fusion*: Detects two loops with the same number of iterations and fuses them, if certain conditions are met (see below);
2. *Variable Nesting Fusion*: Detects two loops where one is a loop nest and another is a simple one ($nesting = 1$), such that the product of the number of iterations of the loop nest is equal to the number of iterations of the simple loop and certain other conditions are met (see below).

Given two loops L_1 and L_2, MATISSE imposes the following restrictions on the fusion of L_1 with L_2, for the standard loop fusion:

- L_1 and L_2 have the same start, interval and end values, so by extension they must have the same number of iterations;
- Neither L_1 nor L_2 have `break` or `continue` operations (note that, at this stage, MATISSE no longer includes the concept of `return` statements);

- If L_1 and L_2 both have operations with side-effects, the loops can not be fused;
- Instructions between the two loops (the *middle block*) must be movable to either before L_1 or after L_2;
- For any matrix M modified in L_1 that is accessed in L_2, each iteration i of L_1 and L_2 must only access M_i;
- The two loops must be *related*, that is, they must access (i.e., read or write) at least one common matrix.

Given two loop nests L_1 (with iteration variables $i_{1,1}...i_{1,N}$ and L_2 (with iteration variable i_2), MATISSE imposes the following restricts for variable nesting fusion:

- MATISSE only attempts Variable Nesting Fusion if Standard Loop Fusion is not valid;
- L_1 is the loop nest, and L_2 is a single loop (i.e., the loop nest must be executed before the single loop);
- All start and interval values must be 1.
- L_2 must consist of a single basic block (i.e., no `break`, `continue`, `branch` or loop instructions);
- i_2 can only be used as part of `simple_get` or `simple_set` instructions with a single index;
- There must be at least one matrix get in L_2 referencing i_2;
- The size of the matrices accessed with i_2 must match the end values of the loop nest in L_1, in the correct order such that replacing the access $M(i_2)$ with $M(i_{1,1}, i_{1,2}, ...)$ should refer to the same value;
- The same side effect and middle block instruction restrictions that apply to Standard Loop Fusion also apply to Variable Nesting Loop Fusion.

To perform Variable Nesting Loop Fusion, MATISSE converts L_2 into a loop nest with the same number of loops as L_1, and then performs Standard Loop Fusion.

Figure 5.4a shows a MATLAB example that benefits from Standard Loop Fusion. Without loop fusion (see Figure 5.4b), MATISSE generates two loops: one for the `X(:)` computation, and another for the `sum(...)` computation. With loop fusion (see Figure 5.4c), MATISSE can generate a single loop, and avoid the allocation for the temporary matrix `sum_arg1`, that stores the result of `X(:)`.

In contrast, Figure 5.5a shows a MATLAB example that benefits from Variable Nesting Loop Fusion. Without loop fusion (see Figure 5.5b), MATISSE generates two loop nestings: a 2D nesting for the `X(2:end, 3:end)` computation, and a simple loop for the `sqrt(...)` computation. Standard loop fusion does not fuse the nestings, as they have different depths and number of iterations. However, the *Variable Nesting Loop Fusion* proposed in this thesis detects that `range_size_1 * range_size == numel_result` and is able to combine the two nests into a single one, as shown in Figure 5.5c.

```
function y = loop_fusion_test(X)
    y = sum(X(:));
end
```

(a) MATLAB source code

```
int X_numel;
int iter;
int iter_1;
tensor_d* sum_arg1 = NULL;
int sum_arg1_numel;
double y;

X_numel = X->length;
create_td_2(X_numel, 1, &sum_arg1);
for(iter_1 = 0; iter_1 < X_numel; ++iter_1){
    sum_arg1->data[iter_1] = X->data[iter_1];
}

sum_arg1_numel = sum_arg1->length;
y = 0.0;
for(iter = 0; iter < sum_arg1_numel; ++iter){
    y += sum_arg1->data[iter];
}

tensor_free_d(&sum_arg1);

return y;
```

(b) Generated C source code, without loop fusion.

```
int X_numel;
int iter;
double y;

X_numel = X->length;
y = 0.0;
for(iter = 0; iter < X_numel; ++iter){
    y += X->data[iter];
}

return y;
```

(c) Generated C source code, with loop fusion.

Figure 5.4: MATLAB program that benefits from Standard Loop Fusion.

However, there are circumstances in which both variants of loop fusion fail to combine two consecutive loop nests. Consider the example in Figure 5.3. If the `%!assume_matrix_sizes_match` directive is removed, there is no longer any guar-

```
function y = loop_fusion_test2(X)
    y = sqrt(X(2:end, 2:end));
end
```

(a) MATLAB source code

```
range_size = X->shape[0] - 2 + 1;
range_size_1 = X->shape[1] - 2 + 1;
create_td_2(range_size, range_size_1, y);
for(iter_2 = 1; iter_2 <= range_size_1; ++iter_2){ // X(2:end, 2:end)
    X_index_1 = iter_2 + 2 - 1;
    for(iter_1 = 1; iter_1 <= range_size; ++iter_1){
        (*y)->data[(iter_1 - 1) + (iter_2 - 1) * (*y)->shape[0]] =
            X->data[(iter_1 + 2 - 1 - 1) + (X_index_1 - 1) * X->shape[0]];
    }
}
numel_result = (*y)->length;
for(iter = 0; iter < numel_result; ++iter){ // y = sqrt(...)
    (*y)->data[iter] = sqrt((*y)->data[iter]);
}
return *y;
```

(b) Generated C source code, without loop fusion.

```
range_size = X->shape[0] - 2 + 1;
range_size_1 = X->shape[1] - 2 + 1;
create_td_2(range_size, range_size_1, y);
for(iter_1 = 1; iter_1 <= range_size_1; ++iter_1){
    X_index_1 = iter_1 + 2 - 1;
    for(iter = 1; iter <= range_size; ++iter){
        (*y)->data[(iter - 1) + (iter_1 - 1) * (*y)->shape[0]] =
            sqrt(X->data[(iter + 2 - 1 - 1) + (X_index_1 - 1) *
            X->shape[0]]);
    }
}
return *y;
```

(c) Generated C source code, with loop fusion.

Figure 5.5: MATLAB function that benefits from Variable Nesting Loop Fusion. Variable declarations were omitted for brevity.

antee that the loops constructed to compute `A(:)` and `B(:)`, or `B(:)` and `C(:)`, have the same number of iterations, as one of the matrices may be a scalar. Thus, in this example, only 2 loops are fused: the one for the `C(:)` operation and the one for the final `sum` operation. However, MATISSE can still generate a single loop for this example by also using the *Direct Combined Element-Wise Loop Generation* optimization, described in the next section.

Direct Combined Element-Wise Loop Generation

The Direct Combined Element-Wise Loop Generation optimization is an extension of the *Element-wise Operation Elimination* described in Subsection 5.1.1 to recognize *chains* of element-wise operations (i.e., element-wise operations that are used only as operands of other element-wise operations). The goal is to directly generate a single loop for the entire chain without temporary matrices.

This optimization can be disabled by using the `%!disable element_wise_combine_loops` directive.

The optimization is applied as follows:

1. Identify element-wise computations (see Subsection 5.1.1);
2. Identify which of these operations should be embedded in other element-wise computations. If this optimization is disabled, then this set is empty, otherwise a matrix should be embedded if all of these conditions apply:
   (a) It is never referenced in a non-element-wise instruction;
   (b) It is only used once;
   (c) It is not a return variable.
3. Compute the data dependence graph for these computations;
4. When an element-wise operation that should be embedded is found, remove it and generates no replacement code;
5. When an element-wise operation that should not be embedded is found, replace it with the equivalent element-wise loop code.

The main limitation of this optimization is that it only works on element-wise operations. In particular, operations such as `sum` and `A(:)` are not considered element-wise, the former because `sum` uses multiple values to compute a single one (so the shape of the output is not the same as that of the input) and the latter because `(:)` is not a function. These limitations explain why this optimization alone is insufficient to generate highly efficient loops. For better code generation, Loop Fusion is still executed after this pass.

5.2 Bounds-checking Elimination

A significant number of MATLAB operations have edge cases that are costly to deal with. For instance, every write to a matrix can potentially trigger a matrix resize so MATISSE must check whether any index is out-of-range. Even if those resizes never happen at runtime, the mere presence of the checks in the generated code can incur a significant overhead inhibit valuable compiler optimizations, including automatic parallelization.

MATISSE handles these situations in two ways:

1. MATISSE features an *Unchecked* mode that turns off most of these checks on a per-function or per-program basis, at the cost of compatibility (the supported MATLAB subset naturally becomes smaller). For an overview of how to use the per-function *unchecked* mode, see Subsection 4.1.2.

2. Static analysis to determine whether these checks are actually necessary, using a solver.

The static analysis attempts to determine whether complex and inefficient SSA instructions (e.g., `set`) can be translated into fast simpler ones (e.g., `simple_set`)[1].

The solver code consists of two components: the *shape solver* and the *scalar solver*. The *scalar solver* receives function-wide information such as `i = a - 2;` to determine whether statements about scalar values (e.g., `i < a`) are necessarily true. The *shape solver* keeps track of the relative shapes of variables and finds statements such as `A = zeros(N, M);` to discover facts such as `size(A, 1) == N`. However, this component is unable to determine how scalar values relate to each other (e.g., `N > N - 1`) without relying on the *scalar solver*.

5.2.1 Scalar Solver

MATISSE features two different scalar solvers: a fast naive one that is mostly used for test purposes, and a second advanced solver based on a third-party library. Symja [Sym16] was initially considered for this purpose, but it was found unsuitable for the purposes of this thesis. For instance, it failed to identify that `a < a + 1`, or that `a < b && b < c` implies `a < c`.

For these reasons, MATISSE uses Z3 [DMB08], a full-fledged theorem prover by Microsoft Research that can be used to check theorems for satisfiability. MATISSE needs to determine whether certain expressions are *necessary*, not *satisfiable* (possible). Testing the necessity of an expression X is equivalent to testing the non-satisfiability $\neg X$. MATISSE only uses a small subset of Z3's features. Notably, it does not currently use any of Z3's array or bitvector operations. In addition, MATISSE relies on Z3's soft timeouts to prevent excessive compilation times.

In order to use Z3 for bounds-checking, MATISSE first builds Z3 *assertions* for each function. To do so, the compiler iterates through SSA IR instructions, and generates the proper assertions, if possible. For instance, given the example in Figure 5.6a, MATISSE builds the assertions specified in Figure 5.6b. Currently, MATISSE instructs Z3 to process floating-point values as real numbers, by declaring floating-point variables as `Real`.

In order to deal with for loops, MATISSE adds the assertion that, if the interval is known to not be negative, the iteration variable must be less or equal to the end of the loop range. MATISSE does not produce code to conditionally deal with branches, so MATISSE does not detect that, in `if(A <= 0) y = A;`, y is necessarily positive.

5.2.2 Shape Solver

The shape solver uses the chosen scalar solver as a base and adds features intended to determine how matrix sizes relate to each other.

Each matrix A has a *size category*, which includes the following information:

- *numel*: the value returned by the `numel(A)` function;
- $dims_x$: value returned by the `size(A, x)` function;
- $dimsSince_x$: The result of $\prod_{i=x}^{N} dims_i$, where N is the number of dimensions.

[1] A list of MATISSE SSA IR instructions is presented in Appendix B.

```
A = 4;
B = user_function(A);
C = A + B;
```

(a) MATLAB code

```
; Assuming that A is an integer, and B and C are double-precision values.
(declare-const A$1 Int)
(assert (= A$1 4))
(declare-const B$1 Real)
; No assertion is built for B = user_function(A)
; As the Z3 scalar solver does not know how to deal with user functions
(declare-const C$1 Real)
(assert (= C$1 (+ A$1 B$1)))
```

(b) Generated Z3 assertions

Figure 5.6: Example of MATLAB code and generated Z3 assertions.

The number of dimensions N is determined by the type and shape inferencer which is executed before the shape solver. As the shape solver does not compute N, it is not part of size categories. Also of note is that since the shape solver is executed at the level of the SSA IR, there is no need to deal with the case of matrices that change size (as each resized matrix has a separate SSA variable).

In order to understand the reason for $dimsSince_i$, consider the example in Figure 5.7. In this example, A is a matrix with 3 dimensions, but only 2 are used in line 4. This means that the value of `end` is `n * n`, and that is the number of elements of $A(1, 1 : end)$. The two matrices `A(1, 1:end)` can be computed separately, and have a size given by two temporary variables. The *dimsSince* information allows MATISSE to determine that those two variables must necessarily have the same value.

The *numel*, *dims* and *dimsSince* information is grouped in a *size category*. When two matrices are known to have the same size, they share the same *size category*.

This information alone is, however, not sufficient to determine the size of certain common cases. To understand why, consider the example in Figure 5.8. In this example, A and y have the same size, since the size of y is directly obtained from A. However, the existence of an intermediate step makes it difficult to determine that these two variables should have the same size category. MATISSE deals with this problem by keeping track of size matrices ($x \rightarrow A$ in this example). When a new matrix is constructed with a function such as `zeros(matrix)`, MATISSE searches the size matrices and reuses size categories, as appropriate.

In order to use this information, MATISSE first has to compute it. In code without loops, analyzing code instruction-by-instruction is sufficient to obtain the necessary information, but in code with loops that is not the case.

In programs that do have loops, such as the one in Figure 5.9, this becomes more complex. If `A$1`, `A$2`, `A$3` and `y$ret` have the same size, MATISSE should be able to discover this fact. This is the case if and only if `A$2` and `A$3` have the same size. Consider the first highlighted instruction. In order to determine that `A$2` has the same size as `A$1`, MATISSE must first determine that `A$1` has the same size as `A$3`. However, `A$3` has the size of `A$2`, which MATISSE does not know about at this

```
function y = f(n)
    A = zeros(2, n, n);
    B = A + 1;
    % ...
    y = A(1, 1:end) + A(1, 1:end);
end
```

(a) MATLAB code

```
Variables:
    A: [category A]
    B: [category A]
    tmp1: [category tmp1]
    tmp2: [category tmp2]
    y: [category y]
Categories:
    A: numel=#1, dims=[2, n, n], dimsSince=[N/A, #2, n]
    tmp1: numel=#2, dims=[1, #2], dimsSince=[N/A, #2]
    tmp2: numel=#2, dims=[1, #2], dimsSince=[N/A, #2]
    y: numel=#2, dims=[1, #2], dimsSince=[N/A, #2]
```

(b) Size categories

Figure 5.7: MATLAB program demonstrating the use case of *dimsSince*. The first *dimsSince* is always N/A because it is the same as the `numel`. Values starting with # represent values that have no corresponding SSA variable.

```
function y = f(A)
    x = size(A);
    y = zeros(x);
end
```

Figure 5.8: MATLAB program demonstrating the use case of the shape solver's size matrices.

point. If MATISSE assumes that `A$3` has the same size as `A$1`, then there is the risk that this is not the case (e.g., if the second highlighted instruction instead accesses an out-of-bounds position), but if MATISSE does not assume that, then this important fact about sizes is not discovered, hindering important optimizations (such as bounds-checking elimination).

MATISSE deals with this case by *speculatively* assuming that variables in loops are in fact not going to be resized, and verifying that the assumption is correct at the end of the loop. If it is not, then MATISSE rolls back and re-analyses the loop more conservatively.

As computing the size categories of matrices is expensive, compiler passes explicitly indicate whether previously computed size information becomes invalid after they are executed. Thus, MATISSE only recomputes this information when it needs to.

```
1  block #0:
2    A$1 = arg 0
3    $one$1 = 1
4    $end$1 = call numel [INT] A$1
5    for $one$1, $one$1, $end$1, #1, #2
6  block #1:
7    A$2 = phi #0:A$1, #1:A$3   % <-- 1
8    i$2 = iter
9    A$3 = set A$2, i$2, $one$1 % <-- 2
10 block #2:
11   y$ret = phi #0:A$1, #1:A$3
```

Figure 5.9: SSA function illustrating the issues of computing size group information instruction-by-instruction without backtracking.

5.3 Matrix Preallocation

MATLAB does not require programmers to initialize matrices before writing values to them, but doing so can still be tremendously beneficial for performance reasons. If a matrix set refers to a position out of index, then a resize operation is triggered. If this happens inside a loop, this can have a substantial negative impact on performance.

The MathWorks Code Analyzer identifies and warns against this performance anti-pattern [Shu12] but, nevertheless, it still appears in many examples.

MATISSE analyses each loop nest N where matrices are resized and preallocates them as appropriate. Note that this optimization has some restrictions:

- Only loops starting at 1 and with an increment of 1 are supported;
- For a given matrix access `A(x)` and given a variable x that may vary in each iteration, it must be possible to determine the maximum value of x within the loop nest:
  - Value x must grow ($x_{new} \geq x_{old}$) with each new iteration;
  - All variables defined within the loop nest that are necessary to compute the value of `x` and are not loop iteration variables of N must be possible to copy to just before N. In other words, they must not have side-effects.
- Matrices to preallocate must not be used inside the loop in SSA statements other than the `set` instructions. They must not be used in, e.g., `numel` expressions;
- Given a loop iteration i with a matrix set $X(expr)$, $expr$ must grow with i (i.e., $i_2 > i_1 \implies expr_2 > expr_1$).

For loop nests that comply with these restrictions, MATISSE injects the appropriate preallocation before the loop nest, either by allocating the matrix (if it was not allocated already), or by adding the resizing operations once before the loop (if it was). Even if MATISSE fails to optimize a full loop nest, it still tries to perform the optimization on its contents (i.e., inner loop nests).

Note that when MATISSE runs in Unchecked mode (see Section 5.2), all indices are assumed to be in-range so MATISSE assumes that no resizes may happen. For this

reason, matrix preallocation fails in unchecked mode and users must manually allocate the matrices instead.

The mechanism matrix pre-allocation can be seen in Algorithm 5.1.

```
function Prealloc:
    Input: function
    loopHierarchy ← getLoopHierarchy(function)
    undefVars ← getUndefinedVars(function)
    loops ← getLoops(loopHierarchy)
    foreach loop ∈ loops do
        convVars[loop] ← getConventionalVariables(function, loop)
    end
    usages ← getVariableUsageCounts(function)
    foreach loop ∈ loops (reverse order) do
        if getStart(loop) ≠ 1 ∨ getIncrement(loop) ≠ 1 then
            continue
        end
        foreach convVar ∈ convVars[loop] do
            /* Ensure that the variables are only being used
               in phi instruction, or in matrix set
               instructions                                  */
            if ¬hasCorrectNumberOfUses(convVar, loop) then
                continue
            end
            /* Traverse the loop nest upwards, as long as we
               find matching loop variables, to find the
               outermost loop that applies to the loop
               variable.                                     */
            nesting ← chooseNesting(loopHierarchy, loop, loopvar)
            optimized ← true
            while nesting ≠ [] ∧ optimized do
                /* Find matrix set instructions, and inject the
                   corresponding matrix allocation operation
                   before the loop.                          */
                optimized ← visitInstructions(function, loop, convVar)
                removeLast(nesting)
            end
        end
    end
end
```

Algorithm 5.1: Simplified pseudo-code describing the algorithm for matrix preallocation.

A loop variable is a tuple `[Initial, LoopStart, LoopEnd, AfterLoop]`), with *after loop* being optional, that meets the following constraints:

- There are "initial" and "loop end" SSA variables that appear in a phi instructions at the start of the loop, and, optionally, in the block following the loop. The two phi instructions use the same variables. The output of the former phi instruction

is the "loop start" variable, and the output of the phi instruction following the loop, if it exists, is the "after loop" variable;

- The has no `continue` or `break` instructions;
- No phi instructions in the block following the loop reference any conventional variables, other than the phi instruction outputting the "after loop" SSA variable.
- The four variables (initial, loop start, loop end, and after loop) must have the same type, with the exception of the initial variable, which can have an undefined type, and after loop, which may not exist.

5.4 Pass By Reference

In MATLAB, function arguments are passed *by value*, copied as necessary. Effectively, this means that there is an implicit copy of every matrix argument.

MATISSE can automatically eliminate matrix copies in functions that use an argument without modifying it. But, on certain programs, in order to achieve performance comparable to C, there is a need for a mechanism to eliminate copies even in cases where the matrix argument is, in fact, modified by the called function. For these cases, MATISSE supports the `%!by_ref VarName` directive, that indicates that certain arguments should be passed by reference. This approach was intended to both be compatible with MATLAB, consistent with our SSA IR semantics and resilient to user errors. Based on these constraints, the directive was designed to work as follows:

- If a function input X is passed by reference, then there must be a function output with the same name (the order of the inputs/outputs does not matter). Furthermore, the called function must not have more than one output named X;
- If a function has a `%!by_ref` output, then whenever it is called, its `nargouts` must be large enough to cover that output;
- The inferred type of the output must be compatible (in terms of C signature) with the type of the corresponding input;
- Arguments passed by reference *must* be simple variables, not complex expressions. For instance, if $f(X)$ has X as a by-ref input, then `f([1, 2, 3])` is not a valid expression and instead `X = [1, 2, 3]; f(X);` must be used;
- In the caller, the inputs passed by reference and their matching outputs must correspond to the same MATLAB variable;
- Currently, only matrix types are supported by the directive, as pass by reference does not yield performance improvements for scalars.

Even then, it is not always possible to avoid the use of a separate matrix. The compiler emits a performance warning when the variable allocator is unable to properly handle `%!by_ref`, and ensures that the generated code is still correct in these cases.

5.5 Execution Schedules

As mentioned in Subsection 4.2.6, MATISSE parallelization mechanism is based on loop nests, with loop iterations being executed in parallel. There is, however, more than one way to map loop iterations to OpenCL work-items.

Given a parallel loop nest with N dimensions and $I_T = I_1 \times ... \times I_N$ iterations, each of the I_T parallel iterations is herein referred to as a *task*. A mapping of tasks to work-items is called a *schedule*, based on the OpenMP concept with the same name [Ope15, Table 2.5]. MATISSE schedules differ from OpenMP in some significant aspects:

- In MATISSE, the schedule is an entirely static loop property (i.e., the schedule of the loop must be known at compile-time, and constant at run-time);
- The list of supported schedules of MATISSE and OpenMP are different. In particular, in OpenMP, the number of threads is controlled separately from the schedule, whereas in MATISSE the number of work-items and the schedule are intrinsically linked.

Figure 5.10 demonstrates a MATLAB program with an explicit user-specified execution schedule (i.e., `direct`). Without the `schedule(direct)` parameter, MATISSE would be free to assign tasks to work-items in any manner it found suitable, as it would if `schedule(auto)` was used. The explicit `direct` schedule forces a strict one-to-one mapping between tasks and work-items, where task i is processed by the work-item with global ID i.

```
1 %!parallel schedule(direct)
2 function y = f(X)
3     y = zeros(1, numel(X));
4     for i = 1:numel(X),
5         y(i) = X(i) * 2;
6     end
7 end
```

Figure 5.10: MATLAB program using an explicit `direct` schedule.

A full list of supported schedules is presented in Appendix C. Schedules can be grouped in 5 categories:

- The automatic schedule (i.e., `schedule(auto)`): The MATISSE is free to assign tasks to work-items in any manner. This is the default.
- The direct schedule (i.e., `schedule(direct)`): MATISSE uses a trivial one-to-one mapping of tasks to work-items. In general, `schedule(auto)` tends to select this schedule in the absence of reasons to select other schedules, as it is considered a *safe* default.
- Thread coarsening schedules (e.g., `schedule(coarse, N)`): Each work-item processes N tasks;

- Fixed work-groups schedules (e.g., `schedule(fixed_work_groups, N)`): MATISSE launches N work-groups (and, by extension, $N \times L$ work-items, with L being the local size), and tasks are evenly distributed across work-items. These schedules are similar to OpenMP's `static` schedule.

- Cooperative schedules (e.g., `schedule(cooperative)`): These schedules are discussed in Section 5.6.

The thread coarsening and fixed work-groups schedules are available in two different variants: the sequential schedules[2], where each work-item processes a contiguous chunk of tasks, and global rotation schedules, where consecutive tasks are assigned to consecutive work-items, up to the available work-items, after which the next task is assigned again to work-item 0. Global rotation schedules are recommended on devices that benefit from coalesced memory accesses, and sequential schedules are recommended for the remaining devices. However, sequential schedules may still be viable on GPUs, for loops with memory accesses that would not be coalesced anyway. If the specific schedule is not specified (e.g., when `schedule(coarse, 2)` is used instead of `schedule(coarse_sequential, 2)`), MATISSE chooses the appropriate variant based on target device information, which is currently `schedule(direct)` in most cases.

Table 5.1 demonstrates how different schedules impact the number of work-items and processed tasks, in an example where 10 tasks are processed and the local size is 2. When the automatic schedule is used, MATISSE makes no assurances whatsoever about the number of work-items, work-groups or tasks processed by each work-item. When the direct schedule is used, the number of work-items, work-groups and processed tasks is trivial to determine. When the coarse schedule (with a coarsening factor of 2) is used, the program launches 6 work-items instead of 5, because MATISSE requires the number of work-items to be the product of the local size and the number of work-groups.

Table 5.1: Example of how different schedules impact the number of work-items and which tasks a work-item processes, if 10 tasks are available and the local size is 2. FWG refers to Fixed Work-Groups. GR refers to Global Rotation.

Schedule	Variant	Number of		Tasks processed by Work-item #0
		Work-items	Work-groups	
Automatic	N/A	Depends	Depends	Depends
Direct	N/A	10	5	[0]
Coarse	Sequential (F=2)	6	3	[0, 1]
Coarse	Sequential (F=4)	4	2	[0, 1, 2, 3]
Coarse	GR (F=2)	6	3	[0, 6]
Coarse	GR (F=4)	4	2	[0, 4, 8]
FWG	Sequential (N=2)	4	2	[0, 1, 2]
FWG	GR (N=2)	4	2	[0, 4, 8]

[2]The *sequential* in the schedule names is not in opposition to parallelism, but rather to the way tasks are mapped. These schedules launch as many work-items as the global rotation variants.

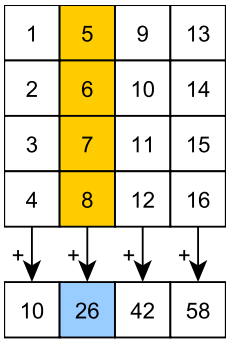


Figure 5.11: Example of a kernel computation with serialized memory accesses. In this example, each work-item accesses a column of data and computes its sum, which is stored in a second buffer.

5.6 The Cooperative Schedule

Section 5.5 discussed the concept of schedules – the mapping of threads to work-items. This section describes in detail a group of schedules herein called the *cooperative schedules*.

5.6.1 Motivation

As previously discussed, one of the most important aspects of GPU code optimization is memory coalescing, to the point where GPU kernels with serialized memory accesses can be slower than the CPU equivalent.

Unfortunately, it is not always trivial to write programs in a way that enables memory coalescing, and some algorithms tend to be particularly hostile to this optimization. Moreover, not all devices benefit from memory coalescing (e.g., CPUs typically do not) and some programmers may prefer to share one single program version across all devices.

For this reason, approaches that automatically optimize programs with serialized accesses into the equivalent with coalesced accesses are important, even if they are only capable of optimizing a subset of all programs.

One common example of non-coalesced accesses occur when a 2D column-major structure is traversed vertically, where each thread processes a column to perform some computation, such as MATLAB's `sum`. A visual representation of this computation is presented in Figure 5.11 (though in a more realistic case, the buffers involved would be significantly larger). At a glance, it would seem that the computation should be efficient – after all, each work-item processes a contiguous chunk of data in a highly predictable manner. However, on a typical GPU device, multiple work-items are executed in tandem, so the memory accesses to 1, 5, 9 and 13 are performed simultaneously, and these accesses are (for a sufficiently large buffer) not on the same cache line, so the accesses are not coalesced.

Another example of this problem occurs on naive matrix-vector multiplications, as presented in Figure 5.12. Once again, at a glance, the code seems to be efficient, as both the matrix and the vector are accessed in the order the elements are laid out in memory. However, consider that, when MATISSE parallelizes the code, the loop with i will be executed in parallel but the loop with j will be executed sequentially, so the access to `M(j, i)` is not coalesced (thread A accesses `M(j, A)` while the next thread accesses `M(j, A+1)`).

```
%!parallel schedule(direct)
%!assume_indices_in_range
function y = matvec_mult(M, V)
        y = zeros(1, size(M, 2));
        for i = 1:size(M, 2),
                acc = 0.0;
                for j = 1:size(M, 1),
                        acc = acc + M(j, i) * V(j);
                end
                y(i) = acc;
        end
end
```

Figure 5.12: Naive matrix-vector multiplication algorithm in MATLAB.

5.6.2 Description of the Optimization

This thesis proposes to fix these inefficiencies by introducing a new category of schedules: cooperative schedules. In a cooperative schedule, there is a many-to-many mapping between tasks and work-items, as each task is processed by multiple work-items *cooperatively*, but each work-item also processes multiple tasks. The host code itself only requires minimal modifications, and often works without any modifications whatsoever.

The total number of tasks processed by a work-group is the same as in the direct schedule (N, where N is the local size of the kernel), but each work-group processes each task one at a time, so *all* work-items in a work-group concurrently process the same task and, once that task is completed, they all move on to the same next task. Parts of the kernel can be executed:

- Redundantly: All work-items compute the same operation independently (used for simple computations);
- By a "leader work-item": A single work-item performs the operation, while the other work-items perform no operations (used for e.g., memory operations);
- By distributing the work across the work-items: Used in certain inner *for* loops (which is called the *cooperative execution loops*), with each work-item processing a separate iteration.

It is clear that the use of the cooperative schedule has an overhead. All work performed redundantly or by a single leader thread is effectively wasted compared to the direct schedule. The cooperative schedule becomes more relevant when the kernel contains inner loops with certain properties.

Consider the example in Figure 5.13. In this example, the outer loop (with iteration variable i) is parallelized, but the inner loop (with iteration variable j) is not, because MATISSE's multi-dimensional parallelization requires perfect nesting. However, the inner loop can still be parallelized at the local level. It is this property that the cooperative schedule exploits to achieve speedups. Loops that can not be parallelized must still be executed redundantly or by a leader work-item.

```
%!parallel
%!assume_indices_in_range
function y = f(W, n)
    y = zeros(1, n);
    for i = 1:n,
        acc = 0.0;
        for j = 1:n,
            acc = acc + W(j, i);
        end
        y(i) = acc;
    end
end
```

Figure 5.13: Example of a MATLAB kernel that may benefit from a cooperative schedule. This function corresponds to the operation shown in Figure 5.11.

Figure 5.14 shows the simplified generated code for the MATLAB code in Figure 5.13, with the direct and cooperative schedules. Note that the cooperative schedule introduces an outer loop (so that each work-item processes multiple tasks) and changes the iterations of the inner loop.

Not all loops can be cooperatively executed. For the most part, the mechanism to determine whether a loop can be cooperatively executed is the same that MATISSE uses to determine whether a loop can be offloaded to the GPU. The only exceptions are that the `%!serial_dimension` directive is ignored and that the supported reduction types are not necessarily the same.

When the MATLAB code does not manually specify the schedule, MATISSE attempts to determine which schedule is the most efficient based on static program analysis and information about the target device. MATISSE uses the following heuristic to determine when to use the cooperative schedule:

- The MATISSE target-aware information about the device must indicate that it is known to benefit from this schedule (e.g., CPUs are always excluded);
- One or more memory accesses in the kernel region are statically predicted to be serialized if the *direct* schedule is used;
- All memory accesses in the kernel region are statically predicted to be coalesced if the *cooperative* schedule is used;
- No *non-trivial* instructions are executed in the kernel outside cooperative execution loops. The instructions that are considered trivial are those in a *white-list*. Notably, this excludes kernels with any loops that are not cooperatively executed.

MATISSE only uses the cooperative schedule if *all* of the aforementioned conditions are met.

MATISSE also implements `subgroup_cooperative`, a variant of the cooperative schedule. The only difference between the two approaches is that this alternative version performs the cooperation on the level of OpenCL subgroups instead of work-groups. The reason for this variant is that, on many OpenCL-capable devices, memory coalescing

```
__kernel void f_1_1(double n, global double* W_data, uint W_dim1, global double* y)
{
    // Variable declarations and initializations omitted

    if (global_size0_1 < N_1) {
        i = global_id0_1 + 1;
        acc_1 = ((double)0.0);
        for(j = 1; j <= n; ++j){
            acc_1 += W.data[j - 1 + (i - 1) * W.dim1];
        }

        y[i - 1] = acc_1;
    }
}
```

(a) Direct Schedule

```
__kernel void f_1_1(double n, global double* W_data, uint W_dim1, global double* y)
{
    // Variable declarations and initializations omitted

    for(task_id0_1 = group_id0_1 * local_size0_1; task_id0_1 < (group_id0_1 + 1) * local_size0_1 && task_id0_1 < N_1; ++task_id0_1){
            i = task_id0_1 + 1;
            acc_1 = ((double)0.0);
            if(local_id0_1 != 0){
                acc_1 = ((double)0);
            }

        for(j = 1 + local_id0_1; j <= n; j += local_size0_1){
            acc_1 += W.data[j - 1 + (i - 1) * W.dim1];
        }

        // Reduction computation omitted

        if(local_id0_1 == 0){
            y[i - 1] = acc_1;
        }

        barrier(CLK_GLOBAL_MEM_FENCE);
    }
}
```

(b) Cooperative Schedule

Figure 5.14: Simplified generated OpenCL for the code in Figure 5.13, using two different types of schedules.

occurs on a level that is more fine-grained than work-groups (see Subsection 2.5.1). If the implementation chooses to map subgroups to this level, then the subgroup variant may exhibit better performance, as the overhead is lowered (i.e., there is less redundant or leader-only work on the subgroup variant) while the advantages are preserved. However, subgroups are only supported since OpenCL 2.0, and even when available, the two approaches should be compared and measured on a per-device basis, as the OpenCL specification does not in any way require memory coalescing and subgroups to be related.

On devices that do not support subgroups, but are still known to have a warp-like behavior, MATISSE implements *sub-groups as warps fallback*, a mode in which MATISSE simulates sub-group operations by explicitly dividing the local size in chunks of a constant size (the warp/wave-front size). This approach, however, has limitations due to the difficulty of emulating the semantics of `sub_group_barrier` using the stricter `barrier` function and, in practice, should be seen as a proof-of-concept workaround until high-quality sub-group implementations are available.

5.7 Data Transfers

When generating OpenCL without Shared Virtual Memory (see Section 2.4), MATISSE generates device data buffers and adds data transfers naively, meaning that the code is highly inefficient. Figure 5.15 shows a simple MATLAB program that can be parallelized by MATISSE and demonstrates where the data transfers would be inserted by MATISSE. As we can see, each loop iteration contains 3 data transfers, which could have easily been moved out of the loop – X is not changed between iterations, and y could be copied to the device in the beginning of the program and back to the host after the end of the loop.

```
1 %!parallel
2 function y = f(X, a)
3     y = zeros(size(X));
4     for i = 1:a,
5         % Data transfer of y and X to the device
6         y = y + X;
7         % Data transfer of y back to the host
8     end
9 end
```

Figure 5.15: Example demonstrating MATISSE's naive data transfer insertion.

To reduce the number of unnecessary data transfers, after the sequential optimizations and basic parallelization operations are performed, MATISSE applies a set of *data transfer optimizations*:

- If a buffer is copied from the device to the host, and then back to the device as a new buffer, after which the old buffer is never used again, MATISSE optimizes away the creation of the new buffer and reuses the old buffer instead (note that a later optimization often eliminates the copy to the host as well);
- If MATISSE detects that there are two constant buffers C_1 and C_2 that refer to the same data (i.e., they are created from the same host SSA variable), and that

C_2 is created in a code location that is in scope of C_1 (i.e., it is not possible to reach the instruction that creates C_2 without first passing through the instruction that creates C_1), then MATISSE eliminates the creation of C_2 and modifies all instructions that use it to use C_1 instead;

- If a device buffer is created from a host matrix that contains only uninitialized data, then MATISSE does not copy the data to the GPU and just allocates the buffer;
- If a device buffer is copied to the host, but used only to determine the size of a matrix (e.g., as an argument to a function like `numel`), then MATISSE eliminates the copy and uses the outdated host matrix instead, as the device will never change the shape of the matrix;
- MATISSE detects patterns of matrices that are copied to the host before a loop A_{before} and after each loop iteration A_{loop}, where the host is used only in `phi` instructions, and replaces the final `phi` instruction with a new copy to host operation, so that a subsequent dead code elimination pass can eliminate copies A_{before} and A_{loop}.
- If a matrix is copied to the device, only for the entire buffer to be overwritten (by a `set_gpu_range` instruction), then MATISSE replaces that copy with a simple allocation;
- Within a loop body, when a matrix B is a loop variable (i.e., defined with `B = phi A, C`), copied to the device at the beginning of the loop, never used in the loop (aside from the copy, the `phi` and potentially calls to size-related functions), its buffer is copied back to the CPU as matrix C and C is also never used in the loop (aside from the copy, the `phi` and potentially calls to size-related functions), then replace references to B with references to A, delete the initial `phi`, move the copy to the device to before the loop and the copy back to the host to after the loop replacing a possible final `phi` (if this `phi` does not exist, then do not insert this final copy);
- Within a loop body, when a read-only buffer is copied to the device or any buffer is allocated on the device, MATISSE attempts to move the copy or allocation to the position immediately before the loop.

5.8 Shared Virtual Memory Heuristics and Optimizations

In addition to the explicit data buffers, MATISSE is able to generate code that takes advantage of SVM *fine-grained buffers* (see Section 2.4), which allow the host to access data in device buffers without any explicit data transfers or memory mapping operations.

This thesis is focused on fine-grained buffers for performance reasons, as data transfers can represent a very significant portion of the total execution time, and fine-grained buffers can eliminate them. However, using SVM is not always profitable. For example, on AMD R9 Nano, a discrete GPU, unconditional use of SVM may lead to slowdowns.

Another instance where SVM is not profitable is the mechanism to store a repeated pattern of data on a buffer. OpenCL features the `clEnqueueSVMMemFill` and `clEnqueueFillBuffer` for this purpose, for SVM and non-SVM buffers, respectively. However, the former is much slower than the latter on the tested devices. In the

context of this thesis, this function is typically used to replace all elements of a buffer with a constant.

Based on MATISSE experiments, this thesis proposes the following heuristics and optimizations, that are selectively applied based on the target device:

1. Instead of `clEnqueueSVMMemFill`, generate an OpenCL kernel that performs the equivalent operations, effectively ignoring the built-in OpenCL function altogether;
2. Replace SVM buffers with dedicated buffers if doing so does not lead to any data transfers (e.g., a buffer is always used by the device, never the host);
3. Replace SVM buffers with dedicated buffers if all generated data transfers are out of a loop with kernel invocations;
4. Do not use SVM buffers if a kernel accesses that buffer in a non-coalesced manner (see Subsection 5.8.1);
5. Do not use SVM buffers if a kernel accesses that buffer in a non-sequential manner (see Subsection 5.8.2).

This thesis proposes the following set of target-aware guidelines, that allow the compiler to obtain performance improvements in most benchmarks evaluated (see Subsection 6.5):

- On an Intel CPU, use optimizations 1 and 2. This allows the compiler to call the `clEnqueueFillBuffer` function for non-SVM buffers (i.e., buffers that are only accessed by OpenCL code) and eliminate the remaining `clEnqueueSVMMemFill` function calls. Optimization 1 is responsible for most of the performance gains on this device.
- On an AMD integrated GPU ("Spectre" architecture, available in the AMD A10-7850K APU), use optimizations 1, 2, 3 and 4;
- On an AMD R9 Nano GPU (a discrete GPU with HBM memory), use optimizations 1, 2, 3, 4 and 5;

As of the time of writing, AMD's OpenCL platform does not support SVM on CPUs and some GPUs. Moreover, NVIDIA's OpenCL platform does not implement fine-grained buffers, so no heuristics were developed for those targets.

5.8.1 Coalesced Access Heuristic

Our most important heuristic is the one that determines whether all memory accesses to a buffer are coalesced. Our algorithm to do this is fairly simple, and it works by trying to check whether all work-items in a work-group either access the same matrix index, or access consecutive indices. In particular, `A(2 * i)` and `A(100 - i)` are treated as non-coalesced. If *any* matrix access is detected as non-coalesced, then that matrix is considered non-coalesced, regardless of any other accesses to the same matrix.

The algorithm keeps track of 3 sets: The set of *locally constant* (LC) variables, the set of *locally sequential* (LS) variables and the set of *non-coalesced accesses*. The compiler traverses each IR instruction exactly once (even instructions in loops or branches), in

the order that they are executed, to build these sets. A variable is locally constant if it is expected to have the same value across all items in a work-group. A variable is locally sequential if two consecutive work-items in the same work-group are expected to have consecutive values for the variable. On kernels with a single dimension, that dimension is the *work-group variable dimension*. On kernels with a local size of $(1, 1, 1)$, the first dimension is the work-group variable dimension. Kernels with other local sizes have no work-group variable dimension. The variables are classified as follows:

- Iteration variables for the work-group variable dimension are locally sequential if and only if the schedule is coalescence-friendly (i.e., `direct`, `coarse_global_rotation` or `fixed_work_groups_global_rotation`), and locally constant in a cooperative schedule (see Section 5.6);
- Iteration variables for dimensions that are not work-group variable are locally constant if and only if the local size of that dimension is 1;
- Iteration variables for kernel `for` loops with start s and interval i are locally constant if and only if:
$$s \in LC \bigwedge i \in LC \tag{5.1}$$
- Iterations variables for kernel `for` loops are locally sequential if and only if:
$$s \in LS \bigwedge i = 1 \tag{5.2}$$
- Constants and variables that are assigned out of the kernel are locally constant;
- On assignments, the assigned variable is locally constant if and only if the assigned variable is as well, and the same for locally sequential variables;
- On any function (including operators) with inputs I and outputs O, the outputs are locally-constant if all of the function inputs are locally constant:
$$\bigwedge_i I_i \in LC \implies \bigwedge_i O_i \in LC \tag{5.3}$$
- If $A = B + C \bigvee A = B - C$, where $B \in LS \bigwedge C \in LC$, then A is locally sequential;
- If $A = B + C$, where $B \in LC \bigwedge C \in LS$, then A is locally sequential;
- If $A = B - C$, where $B \in LS \bigwedge C \in LS$, then A is locally constant;

Matrix accesses (gets or sets) are considered coalesced if their first dimension is indexed with a locally constant or locally sequential variable and all remaining dimensions are indexed with a locally constant variable.

5.8.2 Sequential Access Heuristic

The purpose of this heuristic is to examine loops in the body of a kernel and determine whether memory accesses within them are sequential (i.e., each iteration accesses a consecutive position). This heuristic is particularly useful when combined with the coalesced access heuristic, because the accesses that pass both conditions have good spatial locality both from a work-item and from a work-group perspective. Since accesses

where the same position is repeatedly addressed also have good spatial locality, these accesses are treated as if they were sequential.

This thesis classifies all variables in three groups: *loop-constant*, *loop-sequential* and *random-access*. A matrix access in a loop is determined to be sequential if the first index (i.e. the one that is contiguous in memory) is *loop-sequential* or *loop-constant* and all remaining indices are *loop-constant*. If an outer loop has an iteration variable `out` with an inner loop with an iteration variable `in`, an access in the inner loop to `A(in)`, `A(in + 1)` or `A(in + out)` is determined to be sequential, whereas an access to `A(out)`, `A(1)` or `A(1, in)` is not. If any access to a matrix is not sequential, then the entire matrix is considered to be non-sequential. Matrix accesses outside of loops are ignored.

Our compiler categorizes variables in the following way:

- If a variable is assigned outside of a loop, then it is necessarily *loop-constant* (even if it was considered *random-access* in the outer block);
- A loop iteration variable is *loop-sequential*;
- The result of a sum of a *loop-sequential* and a *loop-constant* (in any order) is *loop-sequential*;
- Any constant is considered a *loop-constant*;
- The result of a subtraction of two *loop-sequential* values is *loop-constant*
- The result of a subtraction of a *loop-sequential* with a *loop-constant* is *loop-sequential*;
- The results of any function where all inputs are *loop-constant* are *loop-constant*;
- All other variables are considered *random-access*.

5.9 Summary

In this chapter, we described various optimizations we implemented in MATISSE to improve the performance of the generated code. Table 5.2 lists the optimizations, and describes which ones are general and which ones are specific to parallel code. Most of the optimizations work without user input of any sort, the exceptions being pass by reference and some of the execution schedules.

Table 5.2: Optimizations described in this section, and their applicability

Optimization Name	Applicable To		Programmer Intervention
	Seq. Code	Par. Code	
Loop Conversion	Yes	Assists parallelization	No
Bounds-checking Elimination	Yes	Assists parallelization	No
By-ref directive	Yes	No	Yes
Schedules	No	Yes	Optional
Data Transfer Optimization	No	Yes	No
Shared Virtual Memory	No	Yes	No

6

Experimental Results

Contents

This chapter describes the benchmarks and test devices used to validate the prototype of MATISSE and the proposed techniques, as well as the obtained results.

6.1 Experimental Setup

In order to determine the effectiveness of the proposed optimizations, we evaluated MATISSE with a number of benchmarks on different devices. In this section, we briefly describe each of those benchmarks and devices.

6.1.1 Benchmarks

MATISSE was evaluated using the following benchmarks:

- Complex Product [CDCP13]: A benchmark that computes the element-wise product of two complex matrices (single-precision). The real and imaginary components are stored in separate matrices. This benchmark has two equivalent variants: one in a loop-oriented code style, and another in a matrix-oriented code style. Different input sizes are used in different sections.
- Dilate [CDCP13]: A function from a stereo navigation application, with single-precision inputs. Different input sizes are used in different sections.
- Hypotenuse [BC17]: A benchmark that computes the length of the hypotenuse of triangles, from the lengths of their *catheti*. Operates with single-precision values. This benchmark is only used in Section 6.2.
- Matrix Multiplication [BRC15a]: Single-precision naïve matrix multiplication algorithm, with square matrices of size 1024×1024.
- Monte Carlo Option Pricing [BRC15a]: Monte Carlo simulation based on an example from MathWorks [Mat14c], to calculate prices for financial options, based on 100 096 simulations each with 50 time steps.
- N-Body 1D [JB07]: Performs an *N-Body* double-precision simulation of a gravitational system. The benchmark is referred to as "1D" because there are separate matrices for each spatial dimension (i.e., all matrices are 1D, storing X, Y or Z data). This benchmark is only used in Section 6.2.
- RGB to YUV [BRC15a]: Performs a conversion of three `uint8` matrices (of size 2048×2048, unless stated otherwise), each with a color component, from RGB to YUV color formats.
- Sub-band Coding [CDCP13]: Component of a MPEG2 encoder originally written in C, translated to MATLAB and generalized to work on multiple input sizes. The inputs are single-precision matrices, of varying size. Different input sizes are used in different sections.
- Disparity (Various versions) [VAJ+09]: *Disparity Map* benchmark extracted from the San Diego Vision Benchmark Suite (MATLAB version). The algorithm estimates the depth of each position of a scene, based on two stereo images. A discussion of the changed made to produce the various sequential versions of this benchmark can be found in Section 6.3. All MATLAB versions of this benchmark

use double-precision matrices representing two Full HD images. For the parallel versions, the following variants were used:

  - Disparity (V1): Based on *Disparity (Modified 1)*, but with manual function inlining and parallelization-related directives.
  - Disparity (V2): Based on *Disparity (V1)*, but with selective parallelization, so certain unprofitable sections are still executed on the host;
  - Disparity (V3): Based on *Disparity (V2)*, but with a parallel implementation of `padarray`.

- Tracking [VAJ+09]: *Feature Tracking* benchmark extracted from the San Diego Vision Benchmark Suite (MATLAB version). The algorithm extracts motion information from a sequence of double-precision matrices representing images. The benchmark was modified to fit in the benchmark system used in this thesis. In addition, the original benchmark depends on MATLAB features that MATISSE does not support (e.g., cell arrays), so the necessary changes for Tracking to run on MATISSE were performed. Finally, the original benchmark used the `sort` function to extract the N first elements of a given matrix (where N is much smaller than the size of the matrix). This function call was replaced with an explicit MATLAB loop to perform the same more efficiently, as MATLAB loops are not inefficient in MATISSE the way they are in MATLAB. The benchmark was tested to track the features across 4 Full HD (1920×1080) images.

- Transpose (1 and 2): Benchmarks designed to serve as a program with coalesced writes and serialized reads or serialized writes and coalesced reads, respectively. Both use single-precision matrices of size 1024×1024.

- Sum Vertical: A benchmark designed to test the cooperative schedule. Computes the sum of each column of a single-precision 4096×4096 matrix.

Moreover, MATISSE was also tested using the Polybench/GPU [GGXS+12], a benchmark suite with multiple GPU implementations (including OpenCL) of 15 algorithms:

- 2DCONV: 2D convolution kernel (input matrix size 4096×4096);
- 3CONV: 3D convolution kernel (input matrix size $256 \times 256 \times 256$);
- 2MM: 2 Matrix Multiplications (input matrix size 2048×2048);
- 3MM: 3 Matrix Multiplications (input matrix size 512×512);
- ATAX: Matrix Transpose and vector multiplication (input matrix size 4096×4096);
- BICG: BiCG Sub Kernel of BiCGStab Linear Solver (input matrix size 4096×4096);
- CORR: Correlation computation (input matrix size 2049×2049);
- COVAR: Covariance computation (input matrix size 2049×2049);
- FDTD-2D: 2-D Finite Different Time Domain kernel (input matrix sizes of around 2048×2048, 500 iterations);

- GEMM: Matrix-multiply $C = \alpha AB + \beta C$ (input matrix size 512×512);
- GESUMMV: Scalar, Vector and Matrix Multiplication (input matrix size 4096×4096);
- GRAMSCHM: Gram-Schmidt decomposition (input matrix size 1024×1024);
- MVT: Matrix Vector Product and Transpose (input matrix size 4096×4096);
- SYR2K: Symmetric rank-2k operations (input matrix size 2048×2048);
- SYRK: Symmetric rank-k operations (input matrix size 1024×1024);

To run these benchmarks with MATISSE, they were ported to loop-based MATLAB code annotated with directives and executed in unchecked mode. Although these benchmarks are designed to be able to run on various precision levels, they were all configured to all with single-precision.

Some of the benchmarks are designed to test MATISSE heuristics, namely those related to data patterns. In these cases, benchmark variants were used, taking a base benchmark, and modifying it to have a different parallelization strategy or data access patterns:

- No Interchange: The loop interchange optimization is disabled for these benchmarks, so the memory access patterns are entirely different. Notably, many coalesced accesses become serialized;
- Outer: Instead of 2D range kernel execution (i.e., nested loop parallelism), only the outermost loop is parallelized, and the inner loop is executed sequentially by each work-item. Again, the memory accesses patterns become substantially different.

The source code of these benchmarks is available at http://specs.fe.up.pt/matisse-cl2-benchmarks.zip.

6.1.2 Target Devices

The code generated by MATISSE was evaluated on the following computing systems:

1. A desktop computer running Windows 10 Enterprise (64-bits) with an AMD A10-7850K APU (@4.10 GHz), 8 GB of DDR3 RAM and 2 GPUs: an *integrated* Radeon R7 GPU (codenamed *Spectre*) that uses the system's DDR3 RAM, and a discrete AMD Radeon R9 280X GPU with 3GB of GDDR5 RAM (codenamed *Tahiti*).
2. A high-performance computer node running Windows 10 Education (64-bits), with two Intel Xeon E5-2630 v3 CPUs (@2.40 GHz), with 128 GB of DDR4 RAM and a discrete AMD R9 Nano GPU with 4GB of HBM memory.
3. A desktop computer running Kubuntu 18.04 (64-bits), with an Intel Xeon E5-1650 CPU (@3.6 GHz), 64 GB of DDR4 RAM and a NVIDIA GTX 1070 GPU.
4. An embedded Odroid XU+E [Hc14] containing an Exynos5 Octa CPU, with 4 Cortex A15 cores at 1.6 Ghz and 4 Cortex A7 cores at 1.2 GHz and 2 GB of LPDDR3 RAM, running Ubuntu 14.04.2 LTS.

6.2 Impact of Temporary Matrix Elimination

In order to demonstrate the impact of the proposed approaches for temporary matrix elimination (see Subsection 5.1.2), 5 MATLAB benchmarks were used that include temporary matrices when they are not optimized properly: *Hypotenuse*, *N-Body 1D*, *Complex Product*, *Disparity* and *Tracking*.

The results in this section are relative to sequential C code. Impact on parallelism is not evaluated on this section. Table 6.1 lists the input sizes used for the benchmarks in this section, some of which are different from those used in other sections.

Table 6.1: Input sizes used for the benchmarks to measure the impact of temporary matrix elimination.

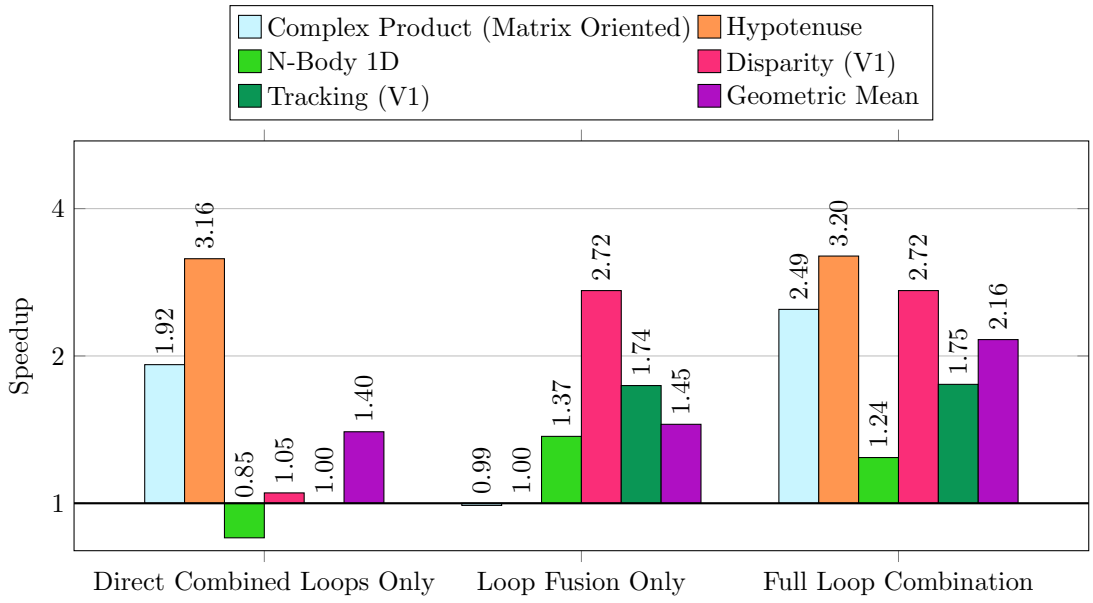
Benchmark	Input Size
Complex Product	$1024 \times 1024 \times 10$
Hypotenuse	512×1024
N-Body 1D	512 bodies, 15 iterations
Disparity	Full HD
Tracking	Full HD

The results are shown in Figure 6.1a. In general, both optimizations tend to improve (or at least not decrease) the performance of benchmarks, but their impact depends on the patterns used on each benchmark. Complex Product and Hypotenuse mostly benefit from the direct combined element-wise loop optimization, as they use patterns that are not easily handled by the loop fusion pass. In contrast, Tracking heavily relies on patterns that are optimized to loops but are not considered "element-wise". In the Disparity benchmark, the element-wise operations are not the bottleneck and use patterns where MATISSE can identify that the intermediate matrices have the same sizes, so loop fusion works reliably without the assistance of direct combined loop generation. The *N-Body* benchmark benefits from loop fusion. It is not clear why the direct combined element-wise loop optimization causes slowdowns.

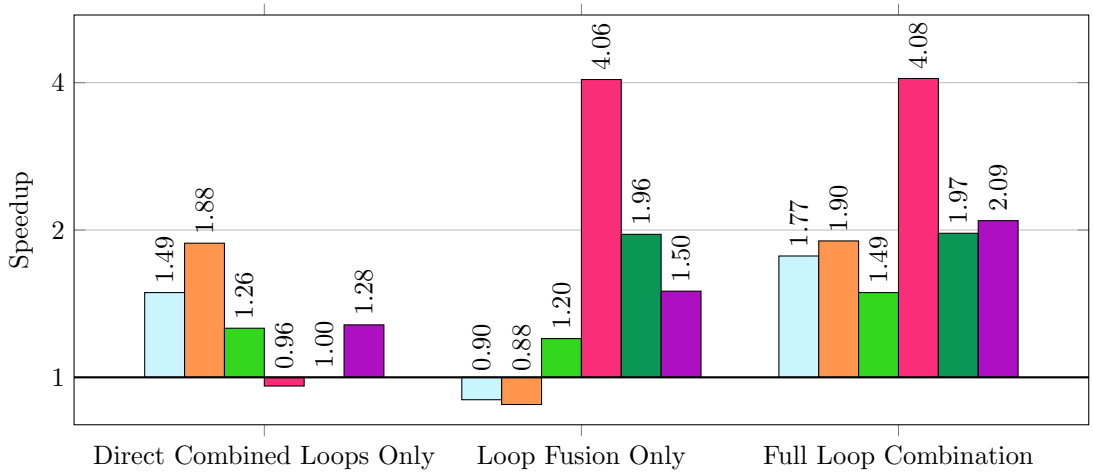
The Complex Product benchmark benefits from using both the direct combined loop generator, as it exposes opportunities that the Loop Fusion pass would otherwise not detect.

These benchmarks were also tested on an embedded system: an ODROID XU+E device [Hc14], with an Exynos 5 Octa Cortex-A15 (1.6 GHz) and Cortex A7 (big.LITTLE), and 2GiB of LPDDR3 RAM, running Ubuntu 14.04.2 LTS. The results are shown in Figure 6.1b. In general, the results are similar, as benchmarks that show speedups, slowdowns or roughly the same results on one platform tend to do so on the other as well. The exception is the N-Body benchmark, that shows improved performance on the direct combined loop generation version.

The memory consumption and estimated memory accesses and L1 cache misses were also measured, using Valgrind/Cachegrind [Val18]. Cachegrind was unable to automatically determine the parameters for the system, so they were manually set based on publicly available information about the processor. These tests assume that the total available cache size is 32 KiB and that the cache line size is 128 bytes. The results are presented in Table 6.2. A visual representation of the memory consumption reductions is presented in Figure 6.2.



(a) Results on a desktop system (computing system 1).



(b) Results on an ODROID XU+E system (computing system 4).

Figure 6.1: Speedups for optimization techniques for temporary matrix elimination.

Note that the allocated memory is the total for an entire program execution, and does not necessarily mean that all that memory needs to be available at the same time. For instance, the disparity benchmark can allocate up to 9 GiB, yet it can run on a device with only 2 GiB of RAM.

In general, the reductions to memory allocations, memory accesses, cache misses and execution times are correlated, with a few noteworthy exceptions. In the Tracking and Hypotenuse benchmarks, the loop fusion transformation actually increases the total allocated memory, despite reducing memory accesses. The reason for this is that the optimization eliminates some temporaries from the SSA IR and corresponding memory accesses, but these temporary matrices do not always correspond to C variables, as MATISSE attempts to combine multiple SSA matrices into a single C matrix.

Table 6.2: Allocated memory, memory accesses and estimated L1 cache misses for the combinations of benchmarks and optimizations, on an ODROID XU+E system, as measured by Valgrind. Two optimizations are evaluated here: Loop Fusion and Direct Combined Element-Wise Loop (DCEWL) optimization. The units Ki, Mi and Gi refer to 2^{10}, 2^{20} and 2^{30}, respectively, as defined by the International Electrotechnical Commission [IEC18].

Benchmark	Variant	Allocated	Accesses	L1 Cache Misses
Complex Product	None	320 MiB	242 M	7 M
	DCEWL Only	240 MiB	158 M	5 M
	Loop Fusion Only	320 MiB	242 M	7 M
	Both	240 MiB	116 M	3 M
Hypotenuse	None	8 MiB	6,893 K	149 K
	DCEWL Only	6 MiB	4,271 K	84 K
	Loop Fusion Only	10 MiB	7,942 K	149 K
	Both	6 MiB	4,271 K	84 K
N-Body 1D	None	119 KiB	264,255 K	2,228 K
	DCEWL Only	96 KiB	184,055 K	1,235 K
	Loop Fusion Only	101 KiB	216,069 K	1,257 K
	Both	85 KiB	128,108 K	145 K
Disparity	None	9 GiB	8,591 M	679 M
	DCEWL Only	9 GiB	8,060 M	663 M
	Loop Fusion Only	2 GiB	2,938 M	112 M
	Both	2 GiB	2,938 M	112 M
Tracking	None	610 MiB	4,599 M	59 M
	DCEWL Only	610 MiB	4,559 M	59 M
	Loop Fusion Only	705 MiB	797 M	60 M
	Both	705 MiB	797 M	60 M

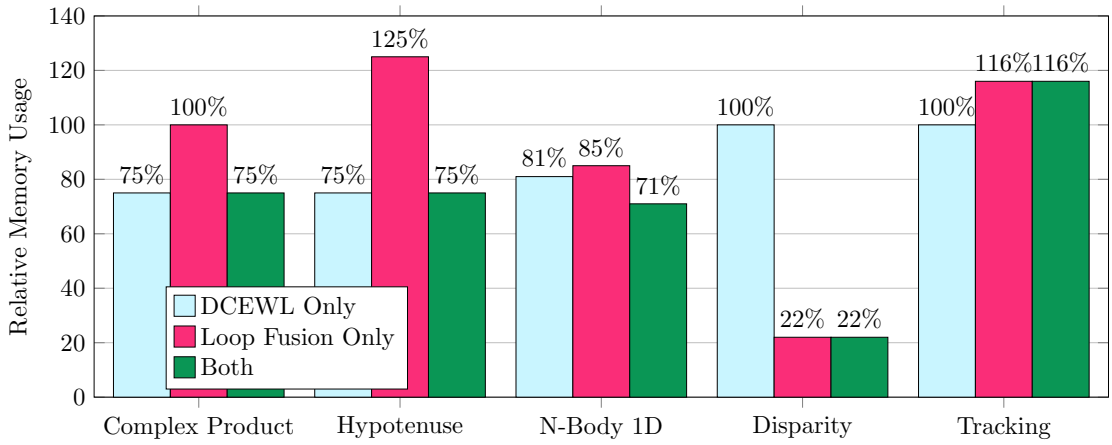


Figure 6.2: Memory usage of programs depending on loop combination optimizations, relative to the version without combined loops. The non-normalized values are presented in Table 6.2. The lower the value, the more effective the optimization(s).

6.3 Comparison of Sequential Versions of Disparity

The performance of MATISSE-compiled programs is heavily dependent on the effectiveness of transformations designed to optimize sequential code, even when parallelization is involved. The reasons for this are that even when code is parallelized, there are often sections that are still executed sequentially and those sections can be bottlenecks, and that many optimizations for sequential code (e.g., dead code elimination) also end up improving the performance of parallel code or even the ability for MATISSE to determine that a section can be parallelized. Thus, before evaluating the performance of the generated parallel code, this section measures the performance of the sequential code, including the various variants that MATISSE supports.

MATISSE is evaluated on two systems: a desktop AMD PC (system 1) and an Odroid XU+E (system 4). The experiments are focused on Disparity from the San Diego Vision Benchmark Suite [VAJ+09], as the original benchmark features both C and MATLAB versions, allowing for a comparison of MATISSE's code generation with C code written by humans. These results were published in [RBC16].

There are two significant differences between the original MATLAB and C versions: the C version uses single-precision floating point and integer values for computations for which the MATLAB version uses double-precision, and the MATLAB version stores the results of each iteration of the algorithm in a 3D matrix, to later convert to a 2D matrix using the `min` function, whereas the C version directly stores the partial results in a 2D matrix and performs a partial `min` operation after each iteration of the algorithm.

The MATLAB version of the *Disparity* benchmark is composed of three files, namely: `script_run_profile.m`, a function that behaves as an entry point for the program, `getDisparity.m`, that contains the actual algorithm implementation, and `refineDisparity.m`. This last file does not appear to be used by the entry point directly or indirectly and as such was excluded from further analysis. The `script_run_profile.m` file was modified to be better integrated with the testing environment used for this thesis, but no other code was modified, except in the code versions explicitly labelled as such. The original `getDisparity.m` file has a total of 47 non-empty lines of code, of which 3 are comments. This file relies on only a few built-in MATLAB functions. Aside from matrix allocation, type casting and built-in operators, only `padarray`, `min`, and `size` are used.

The multiple C versions and variants of the Disparity benchmark are the following:

- *Handwritten Original*: Original C version, as it appears in the San Diego Vision Benchmark Suite, with only minimal changes for integration with the benchmarking system;
- *Handwritten Double*: Same as *Handwritten Original*, but modified to use double-precision data types (instead of single-precision and integer types), for a fair comparison with MATLAB;
- *Original MATLAB (w/o Z3)*: Generated C code for the original MATLAB, with only minor changes to fit into our benchmark system (e.g., by removing file I/O), with runtime checks enabled and using a naïve solver (see Subsection 5.2.1);
- *Original MATLAB (w/ Z3)*: Same as *Original MATLAB (w/o Z3)*, but using the Z3-based scalar solver (see Subsection 5.2.1);

- *Original MATLAB (Unchecked)*: Same as *Original MATLAB (w/ Z3)*, but with some runtime checks disabled for faster performance (the *Unchecked Mode* mentioned in Section 5.2);
- *Modified 1*: Based on *Original MATLAB (Unchecked)*, but with some changes in the MATLAB code for improved efficiency: replacement of a 2D loop with the equivalent 1D loop, code modifications to MATLAB to more closely resemble the C version (including the `min`-related difference described above), and remove calculations for outputs that are known to be unused;
- *Modified 2*: Based on *Modified 1*, but using the `%!by_ref` directive (see Section 5.4);
- *Manually Improved A*: Based on *Modified 2*, but the generated C code was manually modified to remove 6 unnecessary matrix allocations;
- *Manually Improved B*: Based on *Manually Improved A*, but with a manual application of a loop interchange.

Table 6.3 presents code metrics for the MATLAB versions of Disparity (i.e., *Original*, *Modified 1* and *Modified 2*). The total number of lines that were modified is more than half of the lines of code, but most of those can be attributed to a single change: the replacement of the `min` function call with partial operations.

Table 6.3: Metrics of various MATLAB versions of Disparity.

Metric	Original MATLAB	Modified 1	Modified 2
Total Lines of Code (excluding comments and empty lines)	44	52	57
Number of Lines of Code that were added, removed or modified	0	32	45
Directives	0	0	2
Total Functions (in getDisparity.m)	3	3	3

Table 6.4 presents code metrics for the C versions of Disparity. The difference between the two handwritten versions is that the double precision replaces single-precision and integer functions with the double-precision equivalent, meaning that the number of allocation-related support functions is halved (from 6 to 3). Support functions that are not related to allocations (`padarray2` and `padarray4`) remain in the same number. The MATISSE-generated versions include a substantially higher number of support functions. The reason for this is that even simple MATLAB operations such as `size` are often implemented as functions, whereas this does not happen in the handwritten versions. Moreover, matrix allocation in MATISSE causes the generation of several C functions (16 in *Modified 2*), whereas *Handwritten Original* uses only 6 allocation functions.

The handwritten version also uses fewer lines of code for the computation parts of the program. The main reasons for this are:

- MATISSE-generated variable declarations consist of a single variable per line. In contrast, the handwritten versions declare multiple variables of the same type in a single line.

Table 6.4: Metrics of various C versions of Disparity.

Version	Functions		Lines of Code
	Computation	Support	(*.c Computation Files)
Handwritten Original	6	8	166
Handwritten Double	6	5	166
Original MATLAB (w/o Z3)	5	43	826
Original MATLAB (w/ Z3)	5	42	514
Original MATLAB (Unchecked)	5	42	481
Modified 1	4	33	300
Modified 2	4	28	301
Manually Improved A	4	21	258
Manually Improved B	4	21	258

- Several subexpressions are assigned to temporary variables in MATISSE, whereas in the handwritten version they tend to be part of more complex expressions.

Figure 6.3 shows the results for Disparity running on a desktop computer (system 1). Each benchmark was compiled with GCC 4.9.2 (both with `-O2` and `-O3`) 30 times and computed the average and standard deviation of the execution time. In these tests, the 95% confidence estimated error is negligible. The code generated from the original MATLAB achieved execution times around 72% slower than the handwritten version with double-precision. However, the MATLAB code can be manually optimized so that MATISSE achieves performance very close to that of the handwritten C. The only advantage of the handwritten C over the best manually improved MATISSE version is related to data-types. When the handwritten C is used to use the same types (i.e., double-precision floating point) as the MATLAB versions, then the manually improved versions can outperform the C versions ($3.98s$ for handwritten compared to $2.75s$ for the manually improved). With few exceptions, there the differences between using `-O2` and `-O3`, suggesting that the additional optimizations of `-O3` (e.g., vectorization) have little impact on the overall performance.

Similar results apply to Odroid XU+E with GCC 4.8.2, as seen in Figure 6.4. The original MATLAB-generated version without Z3 could not be executed on this device, due to lack of memory (as this version uses more temporary matrix variables). The main differences being that execution times in general are higher (as Odroid XU+E is less computationally capable than desktop systems), and that `-O3` makes an even smaller difference. Other than that, the same conclusions apply.

Now, it is analyzed whether the C compiler was able to vectorize the loops of the best 4 generated versions, as well as the handwritten one, using GCC's `-fopt-info-vec` diagnostics. Table 6.5 shows the results for vectorization of the loops of multiple functions (`computeSAD`, `finalSAD`, and `integralImage2D`). There is not a direct mapping of MATLAB and handwritten C functions, so the table was built based on rough correspondences. For multi-dimensional loops, the vectorization results refer to the innermost loops. All results are relative to GCC's `-O3`, as `-O2` does not vectorize by default.

GCC is able to vectorize the MATISSE-generated equivalent of `finalSAD`, even though it could not vectorize the handwritten version. The loops seem very similar, with the most significant difference being that the generated versions use double instead of single-precision floats and that some values are computed outside the innermost loop

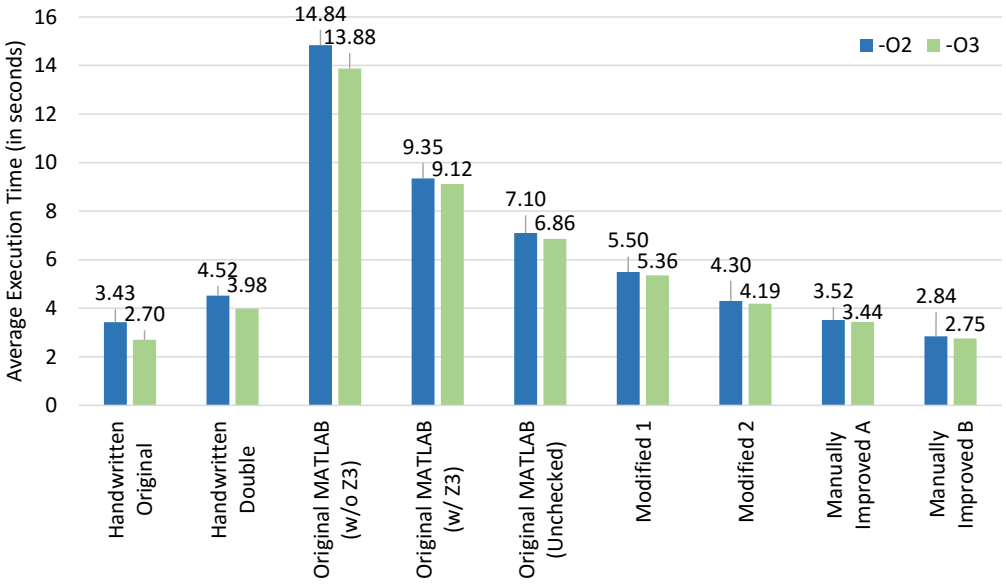


Figure 6.3: Average execution times for various versions of the Disparity benchmark, running on a desktop computer (system 1).

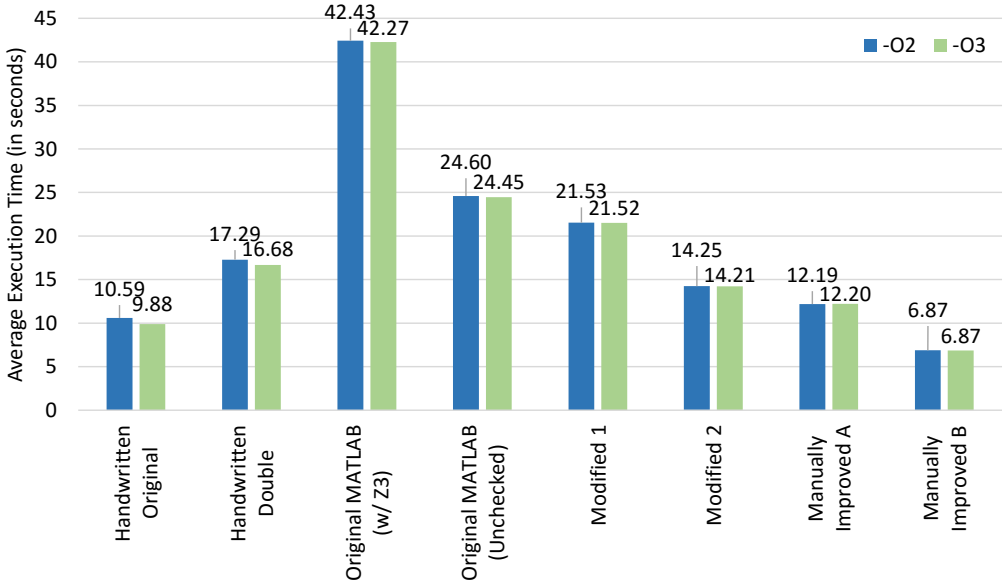


Figure 6.4: Average execution times for various versions of the Disparity benchmark, running on an Odroid XU+E (system 4).

(though loop invariant code motion should be able to do the same), so it is not clear what prevents GCC from vectorizing the loop.

The first loop in `integralImage2D` has no equivalent in the last 3 versions as it was made redundant due to changes introduced by `%!by_ref`. The reason the loops in `integralImage2D` that are vectorized in the generated versions are different from the handwritten versions is because MATLAB is column-major, whereas the C version is row-major. Since each of the loops reads values along a dimension, the two loops effectively swap places in the generated versions.

We also measured the execution time of the original version with MATLAB R2015a

Table 6.5: Vectorization of loops in various C code versions of Disparity.

	SAD		integralImage2D		
	Compute	Final	#1	#2	#3
Handwritten Original	Yes	No	Yes	Yes	No
Handwritten Double	Yes	No	Yes	Yes	No
Modified 1	Yes	Yes	Yes	No	Yes
Modified 2	Yes	Yes	N/A	No	Yes
Manually Improved A	Yes	Yes	N/A	No	Yes
Manually Improved B	Yes	Yes	N/A	No	Yes

on the desktop system 1, using the built-in profiler. According to the profiler, the MATLAB version takes approximately 261.8s to complete. Nearly all of that time (91.5%) is spent on a single line of code, a matrix set in a 2D loop (corresponding to the C function `computeSAD`). This happens because the matrix was not allocated before the loop, so the matrix is continuously resized (see Section 5.3).

To solve this issue, a single line of code was added to allocate the entire matrix before the loop. This new version takes approximately 28.0s to run. Other approaches to reduce the execution time in MATLAB were attempted, and it was found that the best approach was to remove the `computeSAD` loop altogether and use matrix operations instead. This last version takes about 12.2s to run.

As previously noted, these results were published in 2016 [RBC16], and MATISSE has changed since then. One of these changes is a difference in semantics in *unchecked mode*. At the time these results were published, matrices implicitly created inside matrix loops were automatically allocated by MATISSE using the matrix pre-allocator (see Section 5.3). However, our current semantics of unchecked mode are that MATISSE may assume that *all* matrix accesses are in-range. Because of this, matrices implicitly created in loops are undefined behavior in unchecked mode, and the pre-allocator only has an impact in checked mode. As a result of this change, the *Original MATLAB (Unchecked)* version no longer runs, by design. Instead, an explicit matrix allocation should be manually inserted in order to obtain the new equivalent of this version.

Another issue of interest to this thesis was to see how a current version of MATISSE performed, as used of our findings were used to guide optimizations in MATISSE. The current version of MATISSE was evaluated on the desktop computer (system 1), with the generated C being compiled with GCC 6.3.0 (`-O3`). Table 6.6 presents the new results, compared to recompiled versions of the code from 2016. MATISSE now performs fewer unnecessary matrix copies and allocations, and automatically performs loop interchange in some cases, resulting in some improvements to the *Original MATLAB (w/ Z3)* version. However, not all unnecessary copies were eliminated. One notable unnecessary matrix prevents the new *Modified 2* version from performing equivalently to *Manually Improved B* but, in general, fewer changes are required to achieve the performance of *Manually Improved B* from the new *Modified 2*. Overall, the newer version of MATISSE outperforms the older version from 1.16× to 1.50×.

Table 6.6: Comparison of current version of MATISSE with generated version from 2016.

Version	Execution Time (in seconds)		Speedup
	MATISSE 2016	MATISSE 2018	
Original MATLAB (w/ Z3)	9.58	6.39	1.50×
Original MATLAB (Unchecked)	7.41	6.40	1.16×
Modified 2	4.65	3.57	1.30×

6.4 Comparison with Previous MATISSE Backend

MATISSE featured a previous OpenCL backend (see Section 3.1), albeit one with some significant limitations. This section presents the results of a comparison between the two approaches. Note that this does not include any of the proposed work regarding target specialization (e.g., SVM or schedules).

Because the directive systems are different, so is the code that is compiled. Table 6.7 shows the difference in size between the two versions. In general, the differences between the two versions can be attributed mostly to the differences between directive systems. The more compact style of code using matrix operations that MATISSE CL V2 supports was not used, to keep the two versions as close as possible. The simpler style of directives of MATISSE CL V2 is generally not noticeable in these statistics, as they cover lines of code, not size of those lines. The exception being the Monte Carlo Option Pricing benchmark, where the original directives were so long, that they had to be split into multiple lines (using a multi-line comment) for readability.

Table 6.7: Comparison of size differences between the MATLAB benchmark versions for MATISSE CL V1 and MATISSE CL V2. Empty lines and non-directive comments are excluded.

Benchmark	MATISSE CL V1		MATISSE CL V2	
	Total	Directives	Total	Directives
Complex Product	15	2	15	1
Dilate	26	4	24	2
Matrix Multiplication	16	4	14	2
Monte Carlo Option Pricing	82	7	78	2
RGB to YUV Conversion	33	4	30	1
Sub-band Coding	23	2	23	2

Figure 6.5 shows two directives (one for MATISSE CL V1 and one for MATISSE CL V2), to demonstrate the difference between the two approaches. The `copyin` and `copyout` parameters are absent from the newer approach, as they can be automatically inferred by the compiler.

Figure 6.6 shows the speedups of the MATISSE CL V2 over MATISSE CL V1, with the same local size. The Complex Product benchmark operates on matrices of size $2048 \times 2048 \times 20$, Dilate operates on matrices of size 2048×2048 and Sub-band Coding operates on a matrix of size 128×65536. The proposed compiler prototype was able to achieve speedups in most benchmarks, though the reasons for this differ. An explanation

```
%{
acc parallel loop
copyin(initial_seed, riskFreeRate, dividend, volatility, timeToExpiry,
    sampleRate, stockPrice, strike, N)
copyout(finalStockPrices, optionPrices)
local_size(128)
%}
```

(a) Example of MATISSE CL V1 directive.

```
%!parallel local_size(128)
```

(b) Equivalent MATISSE CL V2 directive.

Figure 6.5: Example of MATISSE directive, used in the Monte Carlo Option Pricing benchmark

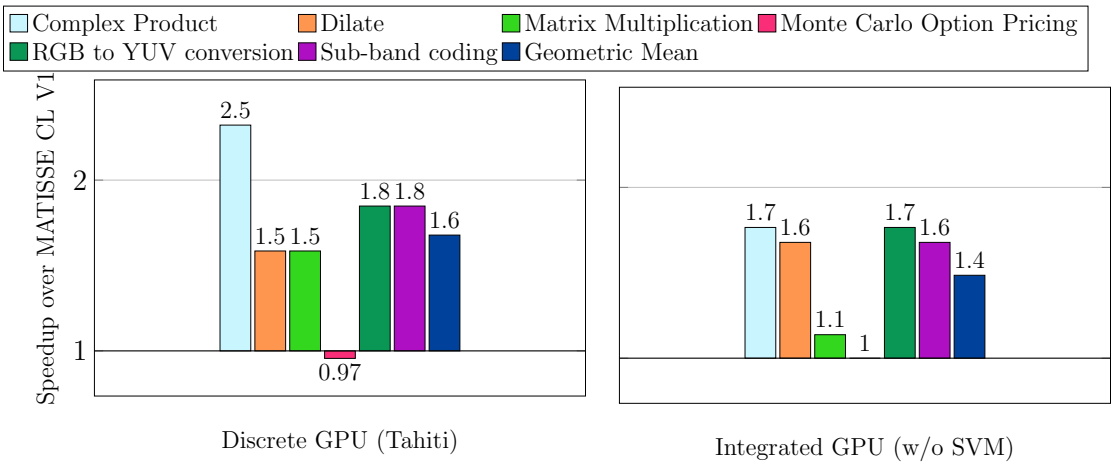


Figure 6.6: Speedups for the total execution time in comparison to MATISSE CL V1.

of these differences follows:

- On the *Complex Product* and the *Sub-band Coding* benchmarks, our previous backend performs unnecessary initializations and copies on the outputs. Effectively, each output buffer is traversed three times by the host – once to initialize it with zeros, once to copy data from the device to the host and once to copy the data from an intermediate buffer to the output. In contrast, MATISSE CL V2 generates efficient initialization code that does not fill the buffer with zeros on allocation because it statically detects that doing so is unnecessary. Additionally, it does not fill the output buffers (as all positions are overwritten) and copies the results from the device to the final buffer directly.
- On the *Dilate* benchmark, the previous backend once again performs a copy of the outputs to the intermediate buffer that the new backend does not. In addition, the new backend performs two loop interchanges.
- The previous versions of the *Matrix Multiplication* and *RGB to YUV Conversion* benchmarks also feature unnecessary copies of the output matrices and missing

loop interchange opportunities.

- The *Monte Carlo Option Pricing* has few data transfers, so the differences between the two backends are less important. The generated OpenCL code is very different, because the previous version performs aggressive inlining which the new one does not, but this does not seem to have an impact on performance. Moreover, the previous version copies its outputs to a buffer and computes the sum of that buffer on the CPU. In contrast, the new backend features a simple per-work-group sum reduction to reduce the number of data transfers and CPU computations. This, too, has little impact on performance as the reduction represents only a very small part of the benchmark execution time. It is noted, however, that there is potential for further improvement of sum reductions.

The proposed prototype was able to outperform the older MATISSE OpenCL backend by 1.6× (geometric mean) on a discrete GPU, and by 1.4× on an integrated GPU.

6.5 Analysis of Shared Virtual Memory (SVM)

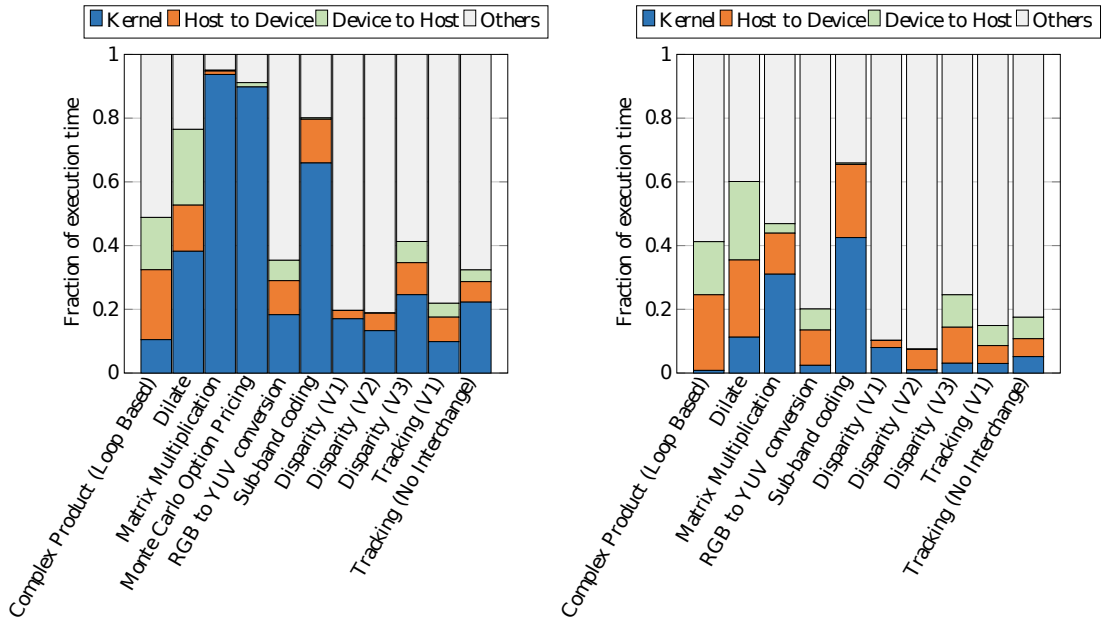
On devices that share their memory with the host (e.g., integrated GPUs that use the same memory as the CPU), Shared Virtual Memory (SVM) can be used to eliminate potentially costly data transfers. This section examines the impact of using SVM. All benchmarks in this section were tested using the direct schedule (see Section 5.5).

Figure 6.7a shows the fractions of execution time spent on each part of the code, on the integrated GPU without using SVM, measured using OpenCL's profiling capabilities, from the start to end of each command (ignoring queue times). This figure shows that the data transfer times represent a substantial portion of the total time on most benchmarks. In particular, the benchmarks with the most significant slowdowns (*Complex Product* and *Sub-band Coding*) spend more time performing data transfers than computing data. For instance, the *Complex Product* benchmark has a speedup of 5.44× when only kernel times are considered. The "Others" part represents the remaining time, which corresponds to time spent in sequential C code and also the overhead of the OpenCL driver. However, when OpenCL code is executed asynchronously, the CPU portions that are executed simultaneously with the parallel portions are excluded.

Figure 6.7b shows the fractions of time for a discrete GPU (AMD R9 Nano), also without SVM. Since this GPU has more computing power than the aforementioned integrated GPU, kernel computation times represent shorter portions of the total execution time.

Despite improving data transfer times (by eliminating most of them), SVM also appears to degrade performance for the kernel computations themselves. In some cases, the additional kernel computation time is greater than the removed data transfer times, which is why some benchmarks are slower with SVM. The loss of performance appears to be related to the memory accesses in the kernel itself. As described in Section 5.8, a set of heuristics and optimizations is used to improve the use of SVM in the compiler.

The heuristics were evaluated with a set of benchmarks from various sources, including some developed or modified for this thesis. *Monte Carlo Option Pricing* was not evaluated, because memory transfers/memory accesses represent a negligible portion of this benchmark. We tested our heuristics on multiple OpenCL devices. The Complex Product benchmark was tested with matrices of input size $1024 \times 1024 \times 10$, Dilate with 4096×4096 and Sub-band Coding with 256×131072.



(a) Integrated GPU (system 1)

(b) AMD R9 Nano (Discrete GPU in system 2)

Figure 6.7: Fraction of time spent on each part of the code.

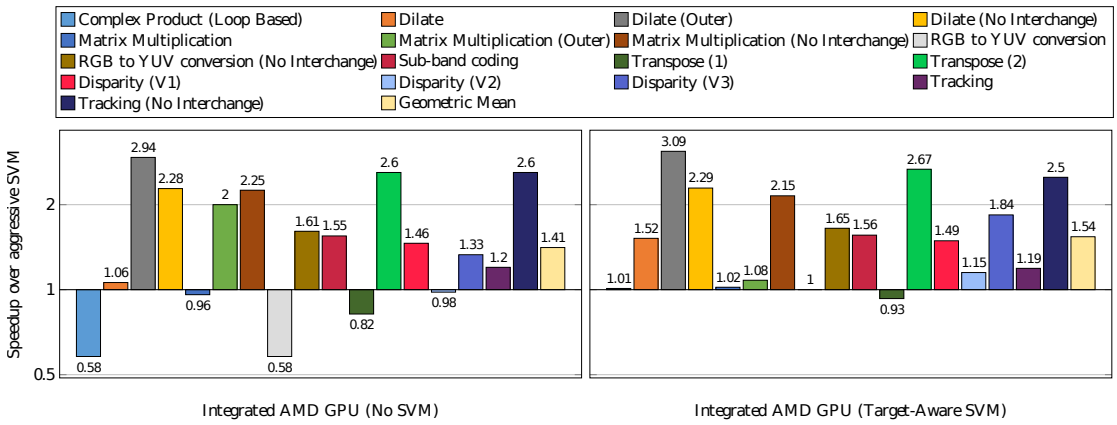
The ODroid XU+E (system 4) and the discrete Tahiti GPU in system 1 were excluded because they do not support SVM, while the NVIDIA GPU of system 3 was excluded because at the time of this thesis the NVIDIA drivers do not support fine-grained buffers.

Figure 6.8 shows the impact of different memory strategies (no SVM, aggressive SVM, aggressive SVM without `clEnqueueSVMMemFill` and target-aware optimized use of SVM), on an integrated AMD GPU (Spectre). The proposed target-aware approach is able to retain the speedups associated with SVM while avoiding most slowdowns. As such, MATISSE was able to achieve geometric mean speedups of 1.54× over the aggressive SVM version and 1.09× over the version without SVM. Only one benchmark (*Matrix Multiplication (Outer)*) showed slowdowns, and even there the heuristics were able to mitigate the impact of SVM.

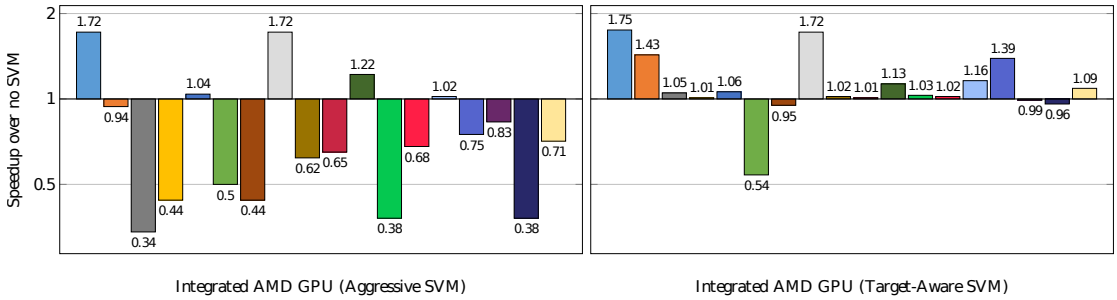
Figure 6.9 shows the equivalent results for an Intel CPU. As expected, SVM is nearly always profitable, but MATISSE can still achieve speedups using the heuristics by avoiding `clEnqueueSVMMemFill` and using `clEnqueueFillBuffer` when possible. The SVM heuristics allow MATISSE to achieve speedups of 9% (geometric mean) over the aggressive SVM approach, with no significant slowdowns on any benchmark.

Figure 6.10 shows the impact of SVM on an AMD R9 Nano GPU. Using the heuristics, MATISSE is able to eliminate the slowdowns associated with using SVM, while retaining nearly all speedups. Overall, MATISSE was able to achieve geometric mean speedups of 1.8× over aggressive SVM and 1.19× over the versions without SVM.

On this last device, our heuristics were unable to predict the best performance for *Transpose (1)* and *Matrix Multiplication (Outer)* on an integrated GPU, *Matrix Multiplication (Outer)* on an Intel CPU and *Transpose (1)* on a Discrete AMD Nano GPU. This suggests that there is potential for further improvement. However, the current heuristics were still able to achieve geometric mean speedups both over no SVM and over aggressive use of SVM, while also preventing all instances with significant slow-



(a) Speedup over aggressive use of SVM.



(b) Speedup over no SVM.

Figure 6.8: Analysis of the impact of Shared Virtual Memory (SVM) and target-aware heuristics, on an integrated AMD GPU (system 1). The Y axis uses a logarithmic scale.

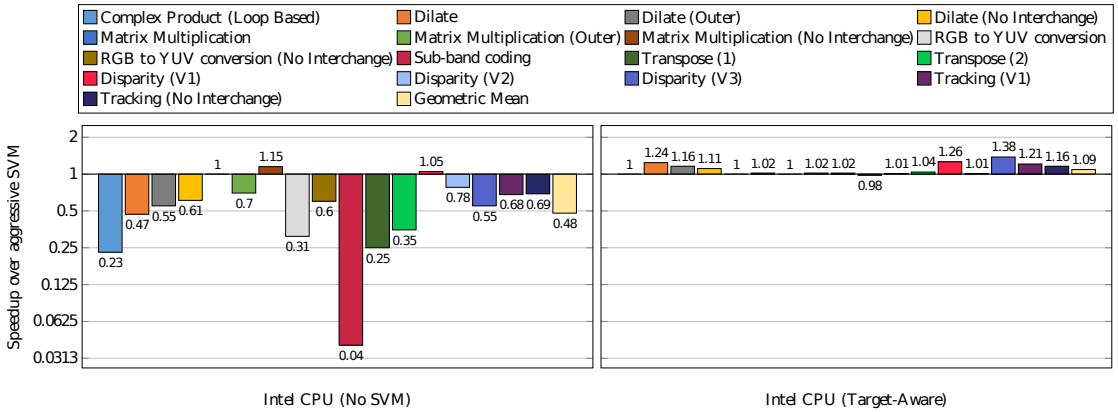
downs, all while using fairly simple heuristics.

6.6 Impact of Parallelization

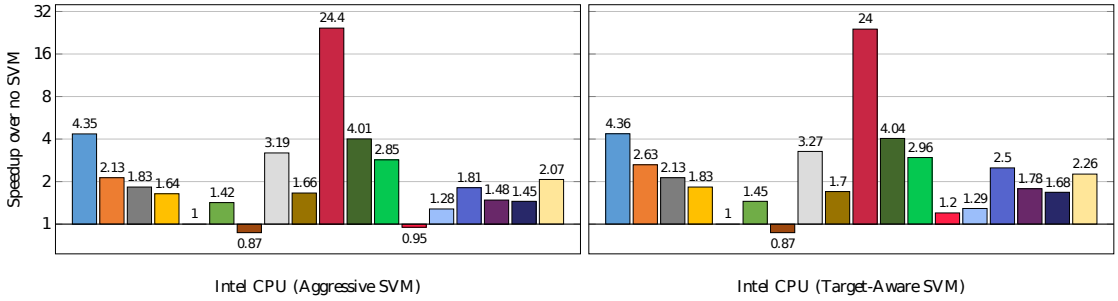
Although parallelization has the potential to improve performance, it is not guaranteed to do so. This section compares the performance of MATISSE-generated C and OpenCL code with that of MATISSE-generated C code (i.e., parallel MATISSE versus sequential MATISSE) and sequential MATLAB, on the San Diego Vision Benchmark Suite benchmarks (*Disparity* and *Tracking*). These two benchmarks contain code that is best executed on the GPU along with code that is best executed on the CPU, and as such allows for an exploration of the implications of parallelization. The Disparity V1 and V2 versions were compared with the same sequential version, as the only differences between the two versions are related to which sections are offloaded to the GPU.

This section only evaluates the benchmarks using the `direct` schedule, but it does measure the impact of SVM. Three OpenCL devices were tested: two GPUs and a CPU, all running on the same computer (system 1).

Figure 6.11 shows the results, when all execution time (i.e., kernel, data transfers and host computations) is considered. In general, offloading is not profitable on the CPU. The added data transfers eliminate any speedups gained by the somewhat limited parallelism of the CPU, but on other devices, parallelization can and often is effective. Of the multiple benchmark variants, only *Disparity (V1)* has consistent slowdowns, as this variant offloads code that would be best executed on the GPU (loops that have very



(a) Speedup over aggressive use of SVM.



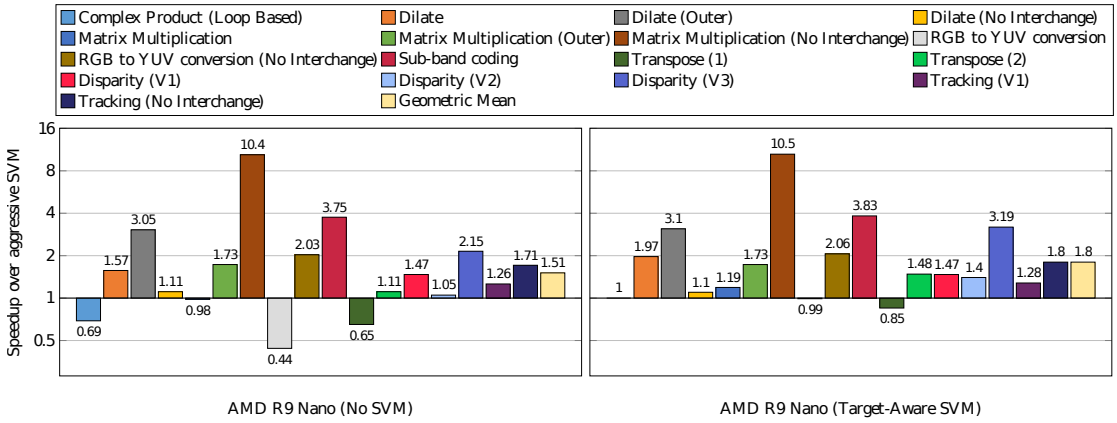
(b) Speedup over no SVM.

Figure 6.9: Analysis of the impact of Shared Virtual Memory (SVM) and target-aware heuristics, on an Intel CPU, with Intel's CPU platform (system 2). The Y axis uses a logarithmic scale.

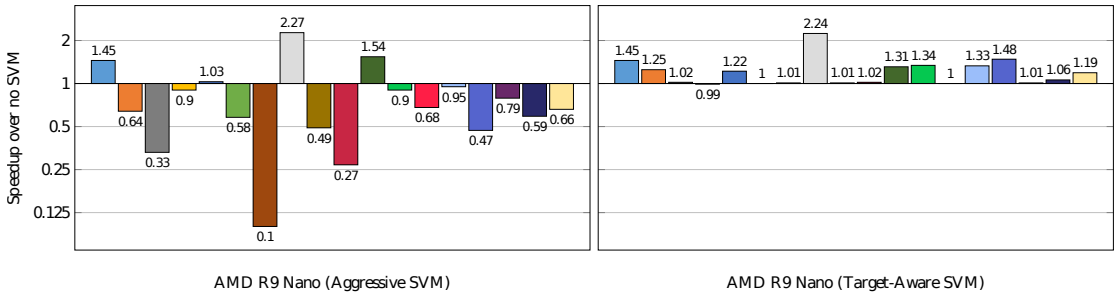
limited parallelism and in one case non-coalesced memory accesses) In this benchmark, SVM does not help performance, as data transfers are not the main issue to begin with – the improved versions (*Disparity (V2 and V3)*) actually have *more* data transfers. The improved performance of Disparity (V2) is caused by the more selective offloading of sections, and Disparity (V3) further improves performance since sections of the code were rewritten to now be recognized by MATISSE as parallelizable. In combination, MATISSE can achieve speedups of more than 2× on both the integrated and the discrete GPU. On the Tracking benchmark, MATISSE can achieve speedups on both the discrete and integrated GPU, but use of SVM is not profitable. When loop interchange is disabled, the Tracking benchmark performs worse in general, but the sequential version seems to be more severely impacted than the GPU version, which is why the speedups are greater. SVM in general becomes unprofitable as the kernel memory accesses are no longer coalesced.

The execution times include measurements of the OpenCL kernel for the discrete GPU, as well as data transfer times. These results can be used to estimate the maximum theoretical speedup over the current generated C/OpenCL code for these benchmarks. The maximum theoretical speedup without any data transfers is defined as:

$$S_{Theoretical} = \frac{T_{Total}}{T_{Total} - T_{DataTransfers} - T_{Kernel}} \tag{6.1}$$



(a) Speedup over aggressive use of SVM.



(b) Speedup over no SVM.

Figure 6.10: Analysis of the impact of Shared Virtual Memory (SVM) and target-aware heuristics, on an AMD R9 Nano GPU, with HBM memory (system 2). The Y axis uses a logarithmic scale.

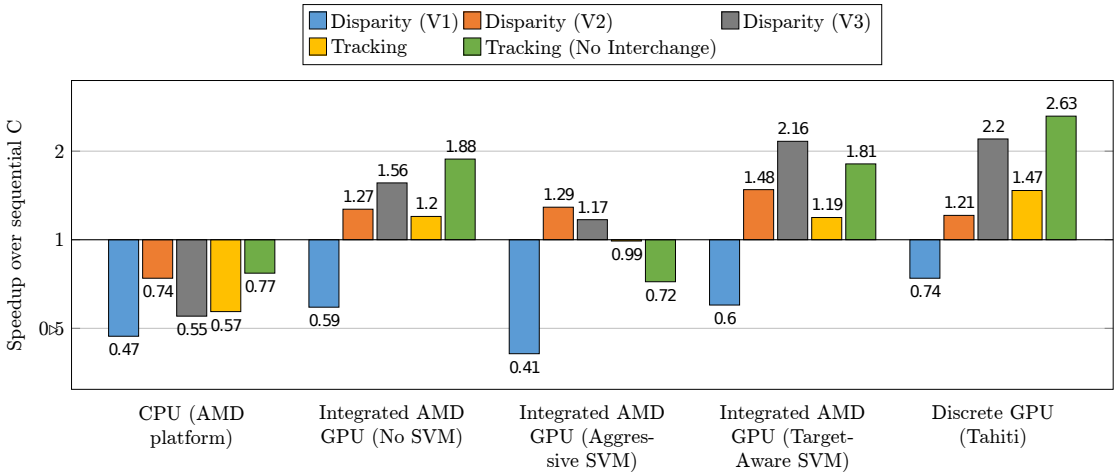


Figure 6.11: Speedups for the generated C/OpenCL versions, compared to the sequential automatically-generated C versions. The Y axis is in a logarithmic scale.

The theoretical speedup with data transfers is defined as:

$$S_{Theoretical(w/DataTransfers)} = \frac{T_{Total}}{T_{Total} - T_{Kernel}} \tag{6.2}$$

For the *Disparity* (V1, V2 and V3) benchmarks, the maximum theoretical speedups

without any data transfers[1] are 1.14×, 1.20× and 1.54×, respectively. With data transfers, the maximum theoretical speedups are 1.07×, 1.01× and 1.04×, respectively. For the Tracking (V1 and No Interchange) benchmark, the maximum theoretical speedups without data transfers are 1.17× and 1.24×, respectively. With data transfers, the maximum theoretical speedups are 1.01× and 1.07×, respectively. This means that our results are close-to-optimal, and improvements to the generated kernel code are unlikely to lead to significant performance improvements. However, there is potential for improvements by optimizing data transfers.

We also compared the performance of the sequential and parallel versions with the original MATLAB. The performance of MATISSE-compiled programs varies substantially across benchmarks, as matrix operations and built-in functions tend to be relatively fast in MATLAB, whereas explicit loops operating on matrices tend to be very slow. For instance, *Complex Product (Loop Based)* and *Complex Product (Matrix Oriented)* benchmarks (with inputs of size $1024 \times 1024 \times 10$) have practically the same performance on MATISSE, but the later is around 6.3× faster than the former in MATLAB, though even the idiomatic version is around 1.15× faster on MATISSE C. The *Disparity* and *Tracking* benchmarks combine both approaches. The original *Disparity* benchmark takes on average 17.7 seconds to run. The fastest MATISSE version is *Disparity (V3)* both for sequential C and parallel OpenCL code generation, taking 2.94 and 1.34 seconds on average, in C and OpenCL (Discrete GPU) mode, respectively. For comparison, the best *Disparity* version for MATLAB runs in approximately 12.2 seconds (see Section 6.3). For the *Tracking* benchmark, the sort removal optimization results in code that runs slower in MATLAB, so it is compared the Tracking (V1) benchmark running on MATISSE with the MATLAB equivalent using the original sort function. On MATLAB, *Tracking* takes on average 387.65 seconds to run. The code generated by MATISSE takes 0.93 and 0.63 seconds to run in sequential C and parallel OpenCL (Discrete GPU) modes, respectively.

6.7 Comparison with Manually Coded OpenCL

In order to test the quality of the proposed OpenCL code generator, the generated code was compared with manually coded OpenCL using the Polybench/GPU benchmark suite [GGXS+12], which implements OpenCL versions of simple algorithms that were rewritten in MATLAB using the MATISSE directives. The performance of the resulting MATISSE programs was compared with the PolyBench/GPU originals. The list of benchmarks in this suite is presented in Subsection 6.1.1.

There are some difficulties in comparing the MATISSE and original versions of the benchmarks, notably in the manner that the benchmarks are structured, as the original Polybench/GPU versions initialize GPU buffer data outside of the timed region, whereas MATISSE includes all data transfers. For improved homogeneity, the Polybench/GPU benchmarks were modified to use OpenCL profiling capabilities in the same manner MATISSE does, but this has the side-effect that CPU-only computations are not timed. Fortunately, there are very few CPU-side computations on all benchmark versions. MATISSE was explicitly instructed to use the same local sizes as the original Polybench/GPU version, and always use the `direct` schedule. All Polybench/GPU

[1] Measured using OpenCL's event profiling info (from `CL_PROFILING_COMMAND_START`). Time waited until a submitted command is actually executed on the device is excluded. Note also that MATISSE's support for asynchronous OpenCL command execution is very limited.

buffers were initialized to zero (instead of leaving them with undefined data), but these initializations were performed on the CPU-side, which were not measured.

For each benchmark, there are multiple ways to offload computations to the OpenCL device. The MATLAB versions were written in an manner that attempted to preserve the "spirit" of each benchmark, but some parallelization strategies simply do not map well to MATISSE. Regardless, even when the MATLAB sources imitates the C++ code closely, MATISSE optimizations can lead to different OpenCL being generated. Notably, MATISSE is capable of optimizing repeated assignments to a matrix position in a loop to assignments to a scalar accumulator variable that is only assigned to the matrix at the end of the loop, in a simple form of *scalar replacement*.

A comparison of the generated source code between the original and MATISSE versions follows:

- 2DCONV and 3DCONV: These benchmarks contain a single kernel invocation (and respective data transfers) each. The MATISSE versions were programmed as the equivalent loops and added the parallelization directives but, due to automated transformations (i.e., the `loop-start-normal` pass mentioned in Appendix D.1), the generated OpenCL is not exactly equivalent. Additionally, in 2DCONV, MATISSE initializes a GPU buffer to zeros by generating a kernel to do so or performing an OpenCL range set (it does not remove the initialization because the kernel does not modify the border of the buffer), whereas the original Polybench/GPU keeps the buffer uninitialized due to the domain knowledge that the contents of the border of that buffer after the kernel call are irrelevant. This is not the case in 3DCONV, where the border is set to 0 in the body of the kernel, so MATISSE can safely remove the initialization. Curiously, on 3DCONV, the Polybench/GPU *does* unnecessarily copy the contents of the buffer to the GPU.

- 2MM and 3MM: A series of kernels, each with a sequential inner loop, the last of which depends on the results of the others. The MATISSE implementation is equivalent to the Polybench/GPU versions.

- ATAX, BICG and MVT: Each benchmark features two kernels performing matrix-vector multiplications, or transpose matrix-vector multiplications. The MATISSE versions are implemented as the equivalent loops. In this version, the *cooperative schedule* optimization described in Section 5.6 are **not** being considered – in other words, only the *direct schedule* is used. MATISSE performs *scalar replacement* on this benchmark.

- CORR: The MATISSE and Polybench/GPU versions are very similar, with two notably differences: MATISSE performs *scalar replacement*, and Polybench/GPU implements CPU writes to individual elements of a GPU buffer by calling the `clEnqueueWriteBuffer` function on a single position, whereas MATISSE inefficiently copies the whole buffer to the CPU to write the single position.

- COVAR, GEMM, GESUMMV, SYR2K and SYRK: The MATISSE and Polybench/GPU versions are very similar, but MATISSE performs *scalar replacement*.

- FDTD: The MATISSE and Polybench/GPU versions are very similar.

- GESUMMV: The MATISSE and Polybench/GPU versions are very similar, but MATISSE allocates a buffer on the GPU without copying data and then zero-

initializes it on the GPU, whereas in Polybench/GPU this buffer is copied from the CPU to the GPU.

- GRAMSCHM: The benchmark consists of 3 kernels, two of which are very similar in MATISSE and Polybench/GPU (and MATISSE was unable to fully perform scalar replacement). However, one of the kernels performs a dot product and computes the square root of a single scalar that is written to a matrix position. Polybench/GPU implements this operation as a kernel where the first work-item performs all the work and all other work-items perform no work whatsoever (256 work-items are launched for this kernel in the Polybench/GPU version). In the MATISSE version, however, this was implemented a simple loop with a sum reduction and relied on MATISSE's built-in handling of reductions, with the result that all work-items perform some work, but the final square root is performed on the host CPU. Unfortunately, MATISSE does not support writing to a single GPU matrix position from the host, so the entire matrix is unnecessarily copied to the host for the square root, and back to the device afterwards.

An additional difference is that MATISSE generates the `reqd_work_group_size` attribute in all Polybench/GPU kernels, whereas the original version does not include this attribute in any version.

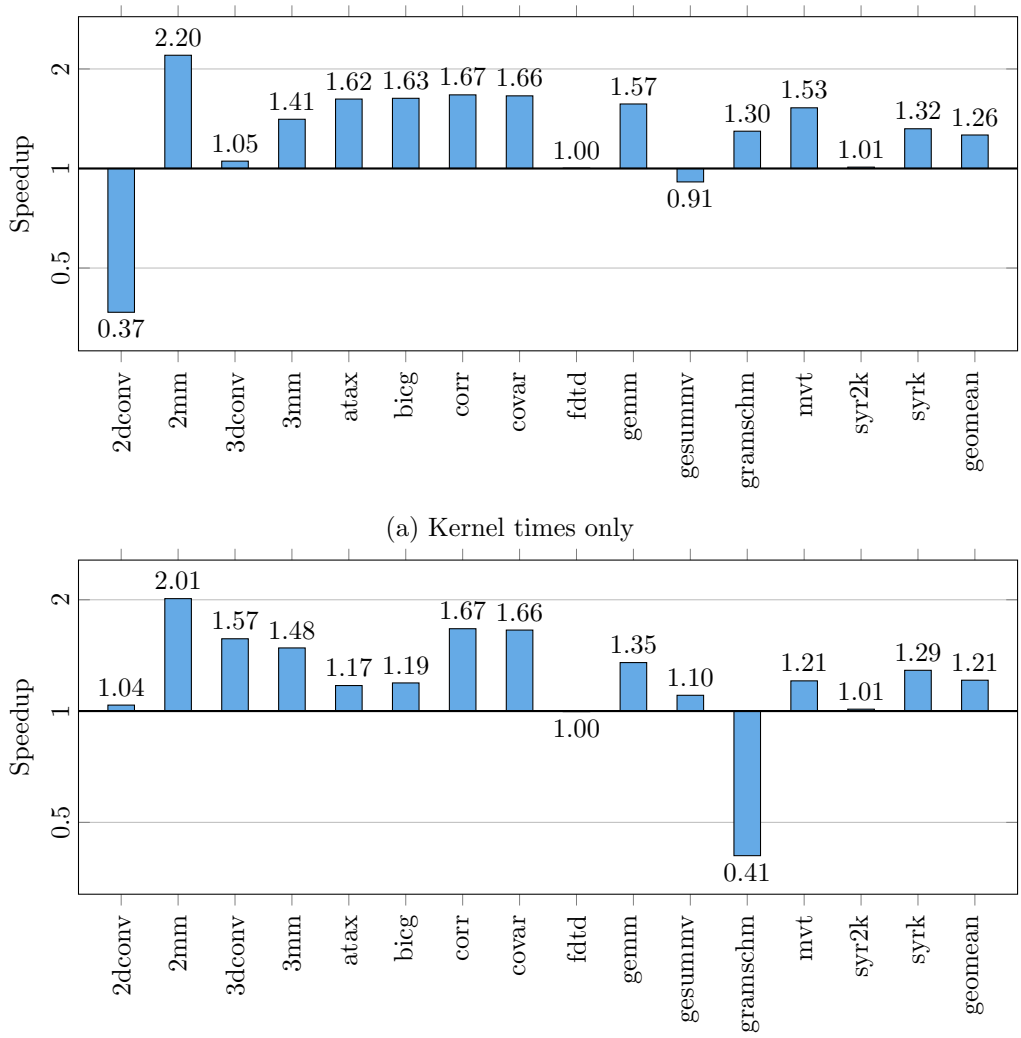
All execution times (kernel times both with and without data transfer times) had a 95% confidence margin of error of less than 3% of the average execution time, and often less than 1% of the average execution time.

Figure 6.12 shows the speedup of the MATISSE versions over the Polybench/GPU counterparts, without any target specialization, running on a discrete GPU on system 1. Both the results for kernel times and the results with the total of kernel and data transfer times are presented.

The MATISSE versions were up to 2.01× faster than the hand-coded originals considering both kernel and data transfer times, and 1.21× faster than the original when only kernel times are considered. The generated kernels were 26% faster than the hand-coded ones, on geometric mean.

The 2DCONV benchmark has notorious slowdowns in kernel times due to the additional generated kernel (for zero-initialization). However, this benchmark is bottlenecked by data transfers, so it has roughly the same overall performance as the Polybench/GPU version. The GESUMMV benchmark kernels take longer to execute in MATISSE because a Polybench/GPU data transfer is implemented as a kernel in MATISSE. When total times are considered, this difference is eliminated and, in fact, MATISSE has a small advantage. The main disadvantage for MATISSE is the GRAMSCHM benchmark, for a reason previously noted: MATISSE generates unnecessary data transfers of an entire buffer in a loop, whereas Polybench/GPU performs a single element copy. The 3DCONV benchmark performs better in MATISSE as the compiler can eliminate an unnecessary buffer initialization/copy. However, in most benchmarks, the speedups of MATISSE can be attributed to a single optimization: scalar replacement.

On this device, target specialization has very little impact on performance, as the only used target property is the detection that this specific GPU supports OpenCL 1.2, so the `clEnqueueFillBuffer` can and should be used. This has an impact on the kernel times of 2DCONV (in fact, it suffices to make the MATISSE version outperform the Polybench/GPU version by approximately 1.69×), but otherwise has no impact on total performance, as the benchmark continues to be bottlenecked by data transfers.



(a) Kernel times only

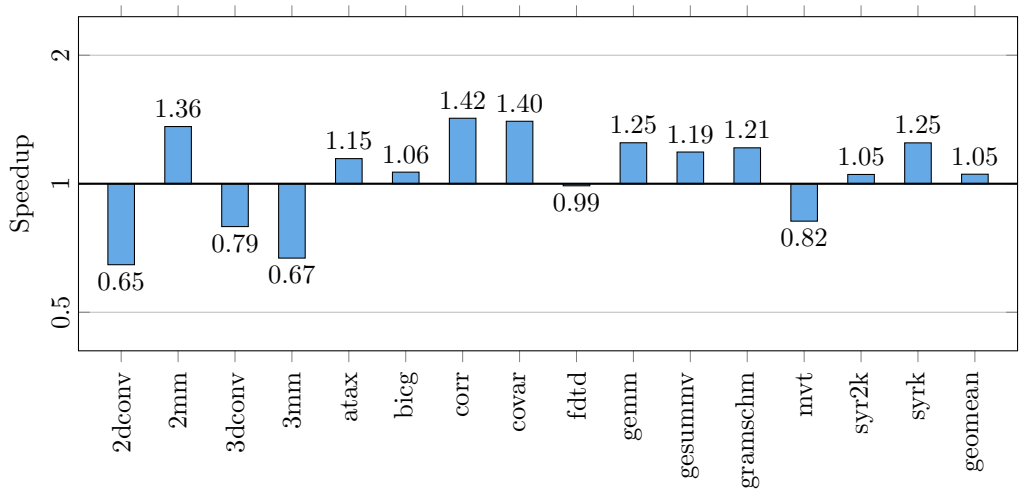
(b) Kernel and data transfer times

Figure 6.12: Speedup of the MATISSE-generated parallel versions over the Polybench/GPU OpenCL versions, running on a discrete GPU (system 1), without using target-aware optimizations. The Y axis uses a logarithmic scale.

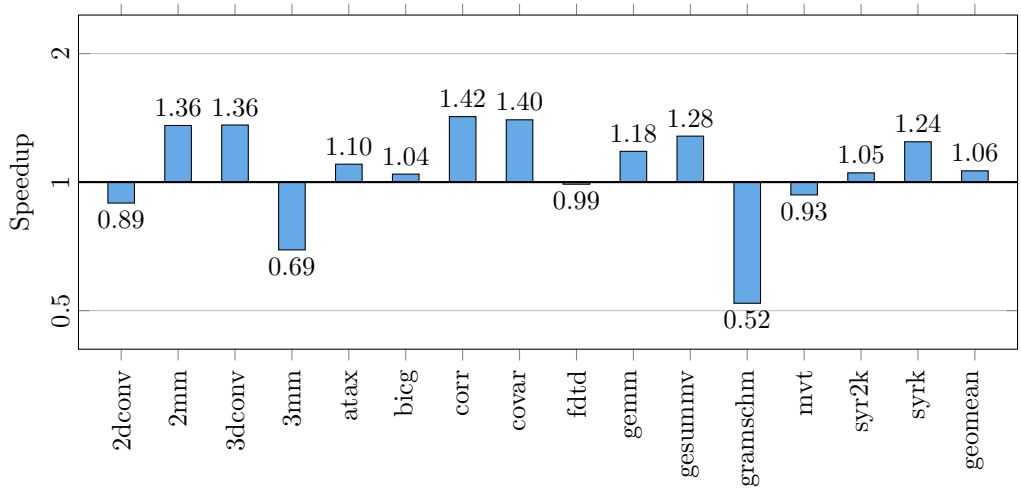
When both kernel and data transfer times are considered, the geometric mean of the difference in performance between the specialized and non-specialized versions on this GPU is less than 1%.

Figure 6.13 shows the speedup of the MATISSE versions over the Polybench/GPU counterparts, without any target specialization, running on an integrated GPU on system 1. Both the results for kernel times and the results with the total of kernel and data transfer times are presented.

On this GPU, MATISSE's kernel time advantages seem to no longer exist (geometric mean speedup of approximately 5%). The fact that the GESUMMV benchmark is now faster on MATISSE than on the hand-coded version is interesting, as MATISSE still generates code with an additional kernel. Possibly, on this GPU and benchmark, the impact of scalar replacement is sufficient to compensate for the additional kernel call. The main difference are the 3MM and MVT benchmarks, in which the MATISSE-



(a) Kernel times only



(b) Kernel and data transfer times

Figure 6.13: Speedup of the MATISSE-generated parallel versions over the Polybench/GPU OpenCL versions, running on an integrated GPU (system 1), without using target-aware optimizations. The Y axis uses a logarithmic scale.

generated code is faster than the PolyBench/GPU equivalent on the Discrete GPU, but slower on the Integrated GPU. In these benchmarks, MATISSE generates kernels that Polybench/GPU implements as copies.

Let us consider the impact of using target specialization on the Integrated GPU. Target specialization is more meaningful on the Integrated GPU due to its support for SVM. Note that schedules other than `direct` have not been used in this experiment. The results are presented in Figure 6.14.

Most Polybench/GPU benchmarks are not impacted by the proposed target-aware specialization. The coalescence heuristics have been designed for kernels with a single non-1 local size, so on Polybench/GPU very few benchmarks end up using SVM. The speedup of 2DCONV is caused by the use of the `clEnqueueFillBuffer` function instead of a dedicated kernel. The BICG and GRAMSCHM benchmarks, however, use SVM and are faster for it, though SVM does not eliminate entirely the slowdown

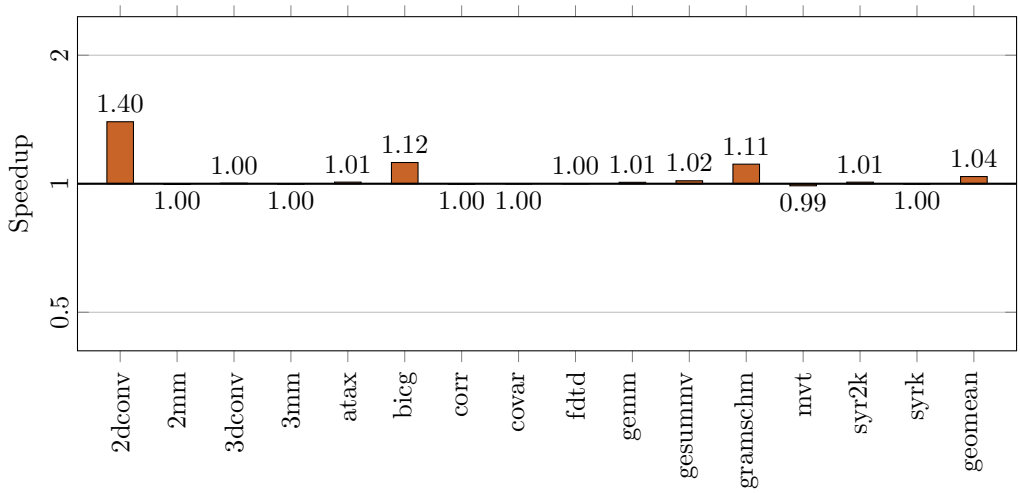


Figure 6.14: Impact of using target-aware specialization for MATISSE-generated code (speedup of specialized over non-specialized), on an integrated GPU (system 1), when both kernel and data transfer times are considered. The Y axis uses a logarithmic scale.

in GRAMSCHM. The ATAX benchmark uses SVM, but it does not seem to have a significant impact.

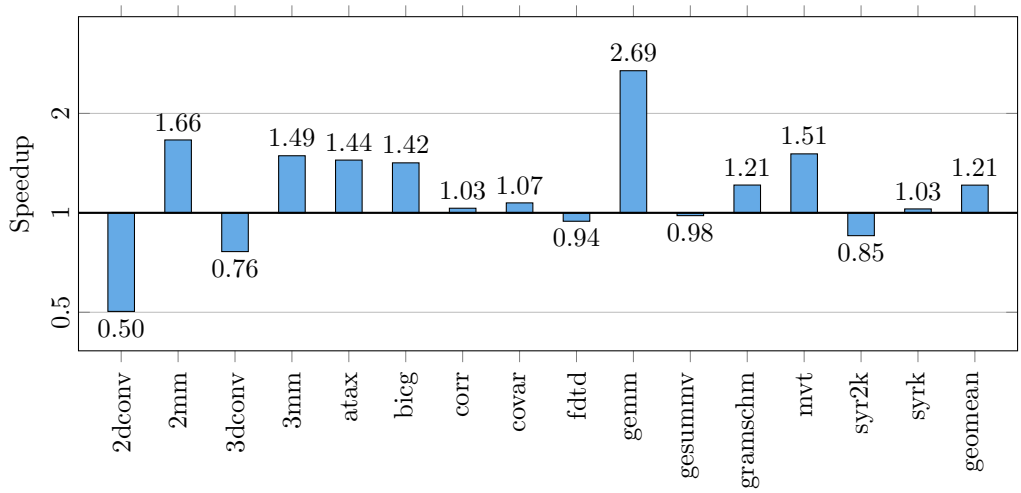
Finally, let us consider evaluated the performance of the MATISSE-generated OpenCl code on an AMD CPU. The results are presented in Figure 6.15.

In general, the AMD CPU platform seems to be less impacted by data transfers, as the kernels themselves take longer to execute. This difference explains why GRAMSCHM is no longer the worst-case scenario for MATISSE – the GRAMSCHM kernels now take almost 11× longer to execute than the data transfers. Other than that, most of the conclusions for the discrete GPU apply to the AMD CPU.

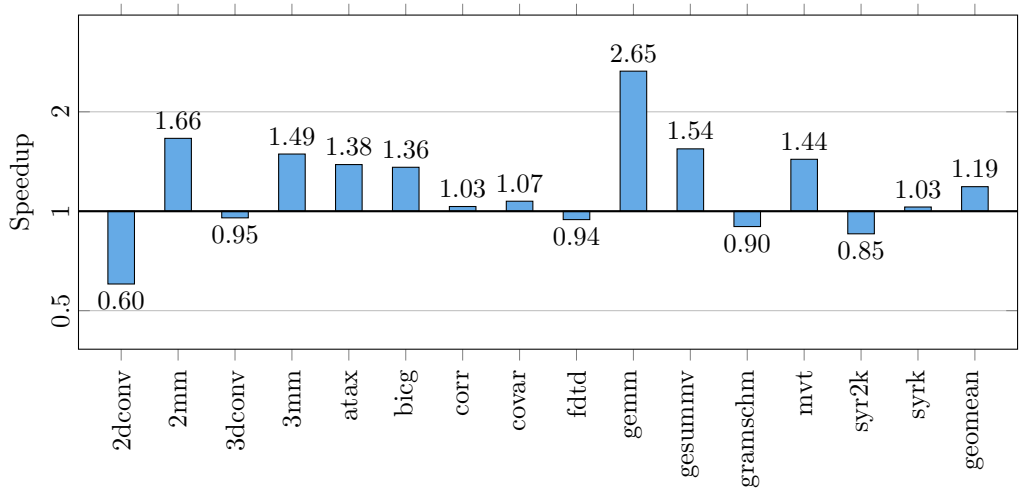
As for target specialization, AMD's OpenCL implementation for CPUs does not support SVM and it was forced the use of the `direct` schedule on all benchmarks, so the only specialized property is the support for `clEnqueueFillBuffer`. The end-result is similar to that of target specialization for discrete GPUs, this time with a geometric mean speedup of around 1.5%. Notably, however, since the generated kernels are much slower on the CPU than on GPUs, the impact of the fill buffer function is larger on the 2DCONV benchmark. Due to this difference alone, the target-aware version is almost 20% faster than the non-target-aware version, when both kernel and data transfer times are considered.

In order to determine the impact of using MATISSE on code readability, an analysis of each of the benchmarks is presented in terms of lines of code of original Polybench/GPU C and OpenCL, lines of code of MATISSE OpenCL, number of directives, and how many of those directives are `%!no_index_overlap`. For the purposes of the number of lines of code of the original Polybench/GPU, the lines of code that were added for profiling purposes were excluded, as those were not in the original version. Comments and empty lines were also excluded. These results are presented in Table 6.8.

Part of the cause for the higher number of lines in the C version is the presence of validation code, including an implementation version of the same algorithms as sequential C code. But even without these, the C code is substantially larger than MATLAB, as C with OpenCL is substantially lower-level than MATLAB with MATISSE directives. For instance, the MATLAB version omits the data transfer and OpenCL initialization



(a) Kernel times only



(b) Kernel and data transfer times

Figure 6.15: Speedup of the MATISSE-generated parallel versions over the Polybench/GPU OpenCL versions, running on an AMD CPU (system 1), without using target-aware optimizations. The Y axis uses a logarithmic scale.

code entirely. Keep in mind that the MATLAB versions were written in a loop-oriented code style. A vectorized approach would be even more compact.

In general, the results show that MATISSE is competitive with hand-coded C/OpenCL code, at least when the MATLAB code is written to be similar to the low-level equivalent. However, for improved performance, MATISSE should be modified to account for cases where only small segments of a matrix need to be copied, as that would fix the significant slowdowns present in the GRAMSCHM benchmark.

6.8 Impact of Cooperative Schedule

The `schedule(cooperative)` was developed to optimize three similar benchmarks: ATAX, BICG and Sub-band Coding, which all perform what is effectively a matrix-vector multiplication. However, to ensure that the optimization was applicable to other

Table 6.8: Lines of code (LoC) of the hand-coded C and MATISSE versions of the Polybench/GPU benchmarks. NIO refers to the number of `%!no_index_overlap` directives.

Name	Hand-Coded #LoC		MATLAB #LoC	
	C	OpenCL	Total	Directives (NIO)
2DCONV	207	21	28	1 (0)
2MM	278	32	18	1 (0)
3DCONV	219	30	28	2 (0)
3MM	325	45	27	1 (0)
ATAX	241	28	16	1 (0)
BICG	261	32	18	1 (0)
CORR	325	66	50	7 (1)
COVAR	273	46	39	9 (1)
FDTD	279	41	26	3 (0)
GEMM	235	20	12	2 (0)
GESUMMV	229	20	11	1 (0)
GRAMSCHM	265	45	26	3 (0)
MVT	248	30	15	1 (0)
SYR2K	232	20	14	2 (0)
SYRK	220	20	12	2 (0)

use-cases as well, a different artificial benchmark was developed (i.e., *Sum Vertical*). This benchmark also exhibits the traits that the optimization is designed to apply to, despite not being a matrix-vector multiplication.

This section attempts to answer the following questions:

1. Which variants of `schedule(cooperative)` work best;
2. What speedup can be expected when the `schedule(cooperative)` is used selectively, in comparison to not using it at all;
3. What slowdown can be expected when the `schedule(cooperative)` is used improperly, in comparison to not using it at all;
4. Whether the profitability of `schedule(cooperative)` varies with input sizes.

To answer the first three questions, the ATAX, BICG and Sum Vertical benchmarks were executed with single-precision matrices of shape 4096×4096 (and vectors of size 4096, when applicable). For the Sub-band Coding benchmark, a single-precision matrix of size 256×131072 (and a secondary matrix of size 32×256 that is only used in sequential code) was used.

The variants of the schedule that were measured are:

- Level of cooperation: Work-group cooperation (i.e., `cooperative`), sub-group cooperation (i.e., `subgroup_cooperative`, required OpenCL 2.0) or emulated sub-group cooperation (i.e., `subgroup_cooperative` with the *sub-groups as warps fallback*).

- Different forms of performing the reductions of the cooperative execution loops: the kernel can aggregate all partial results in a local buffer and then have the leader work-item aggregate the results in a simple `for` loop (the *simple* variant), the kernel can aggregate the results in a hierarchical manner, where an work-item combines two partial results, and on the next iteration only half as many work-items aggregate the result, until a single work-item (the leader work-item) has the final value (the *interleaved* variant), or finally the kernel can use OpenCL's built-in reduction functions (the *native* variant, only available in OpenCL 2.0).

Therefore, on devices without OpenCL 2.0, 4 variants are tested (*simple* and *interleaved* reduction variants, multiplied by two levels of cooperation), and on devices with OpenCL 2.0, 8 variants are tested (the three reduction approaches, multiplied by the three levels of cooperation, minus the invalid combination of native reductions with *sub-groups as warps fallback*). All measured times refer to kernel times, with data transfers and host-side computations excluded, as the cooperative schedule requires few to no changes to the host side.

Figure 6.16 shows the results for a discrete AMD GPU (system 1, with OpenCL 1.2). The results show that the interleaved version was consistently faster than the simple version. In geometric mean, the interleaved version was around 29% faster than the simple version (for the work-group variants). The difference for the sub-groups as warps fallback is not as significant, as warps are smaller than work-groups, so the difference in processed work between the simple and interleaved versions is lessened. The warp fallback versions were found to be somewhat faster than the work-group versions, especially among the non-interleaved versions. Regardless, on these benchmarks, even the less efficient versions of the cooperative schedule resulted in significant speedups over the direct schedule (6.0 to 8.5×).

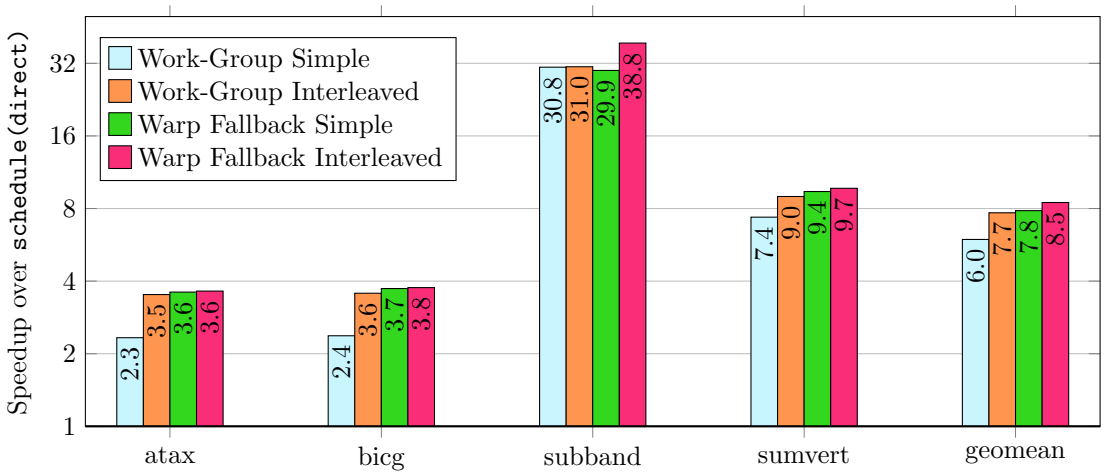


Figure 6.16: Comparison of multiple versions of the cooperative schedule, on a discrete AMD GPU running on system 1, in terms of speedups over the direct schedule.

Figure 6.17 shows the results for an integrated AMD GPU (system 1, with OpenCL 2.0). The results show that the performance of the native implementations of reductions (e.g., `work_group_reduce_add`) is extremely disappointing, as even the simple implementation was able to outperform it nearly always. The interleaved versions are still the best approaches for the work-group and sub-group variants. The results also show that the sub-group variants are slower than expected, particularly since significant

speedups were achieved with the sub-group as warps fallback. This suggests that either AMD's sub-group implementation has margin for further optimization, and/or some hard-coded information that was provided to the warp fallback variant (e.g., the warp size, or the knowledge that the global size is always a multiple of the local size[2]) can be used to generate more efficient code. Once again, even the less efficient versions of the cooperative schedule are faster than the direct schedule for these benchmarks.

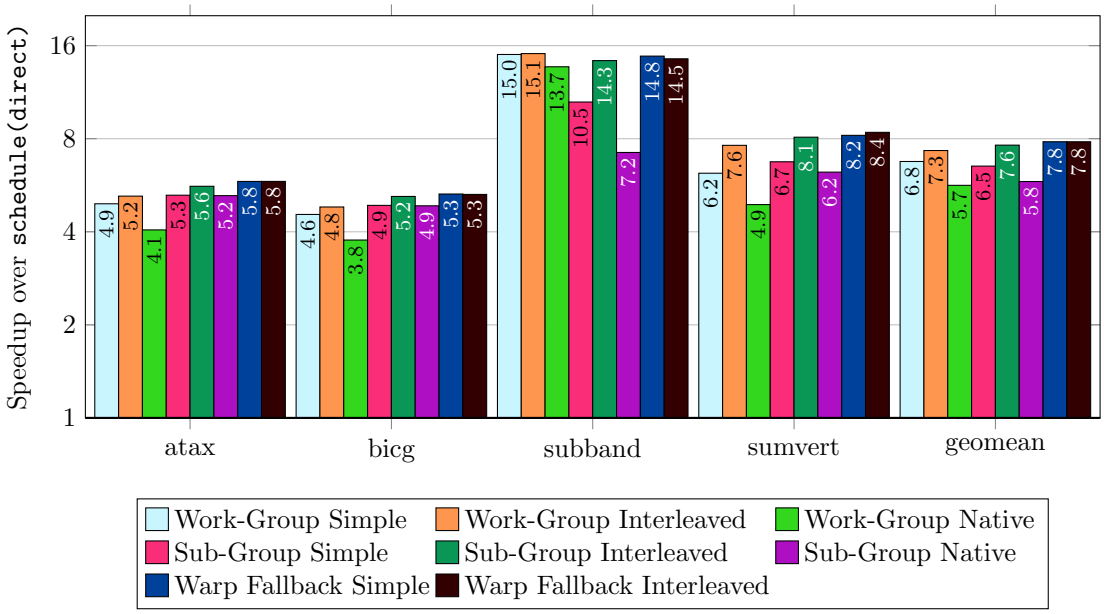


Figure 6.17: Comparison of multiple versions of the cooperative schedule, on an integrated AMD GPU running on system 1, in terms of speedups over the direct schedule.

On an AMD CPU OpenCL platform (system 1, with OpenCL 1.2), the results show that the cooperative schedules consistently perform worse than the `schedule(direct)` versions, so it was not determined which variant is best – the optimization should be disabled altogether.

Figure 6.17 shows the results for a discrete NVIDIA GPU (system 3, with OpenCL 1.2). It was found that the work-group cooperative schedule with simple reductions caused slowdowns, and the interleaved reduction, while better, was insufficient to consistently result in speedups. The sub-groups as warps fallback, on the other hand, did produce significant speedups (geometric mean of 2.06× faster). The reduction type to use (simple or interleaved) with the warp approach seems to matter very little, as the performance difference is below 2%.

It was also attempted to determine the impact of using the cooperative schedule in circumstances in which doing so is not appropriate. For instance, the ATAX benchmark consists of two kernels, but only one benefits from the cooperative schedule, as the other already has coalesced memory accesses. For this experiment, MATISSE was forced to use the cooperative schedule (using interleaved reductions) for both kernels, and the impact of doing so was measured. Figure 6.19 shows the results. As seen, the *forced* variants (i.e., where the cooperative schedule is used regardless of whether it makes sense) consistently perform worse than the proper version, and nearly always perform

[2]This is true for MATISSE-generated code, but no longer true in the general case in OpenCL since version 2.0.

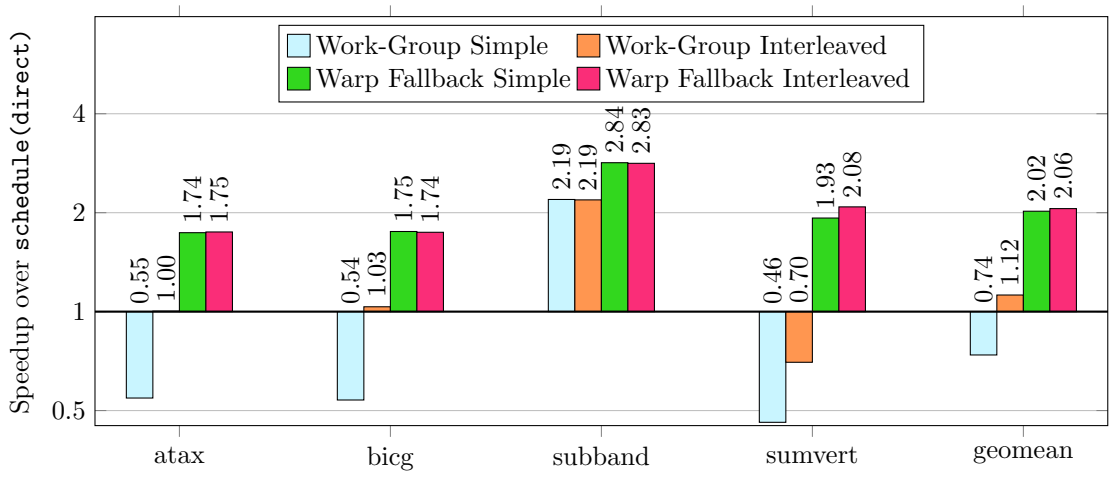


Figure 6.18: Comparison of multiple versions of the cooperative schedule, on a discrete NVIDIA GPU running on system 3, in terms of speedups over the direct schedule.

worse than if no optimization had been done at all. These results suggest that the penalty for using the cooperative schedule when it is unprofitable to do so is generally greater than the benefit of using the cooperative schedule when it is profitable. On the AMD GPUs, the three proper versions perform roughly the same, so the difference in performance between the forced versions implies that the penalty for improperly using the cooperative schedule is lessened with the sub-group variants. This makes sense, as each sub-group is smaller than a work-group, so the redundant/leader-only workload is reduced.

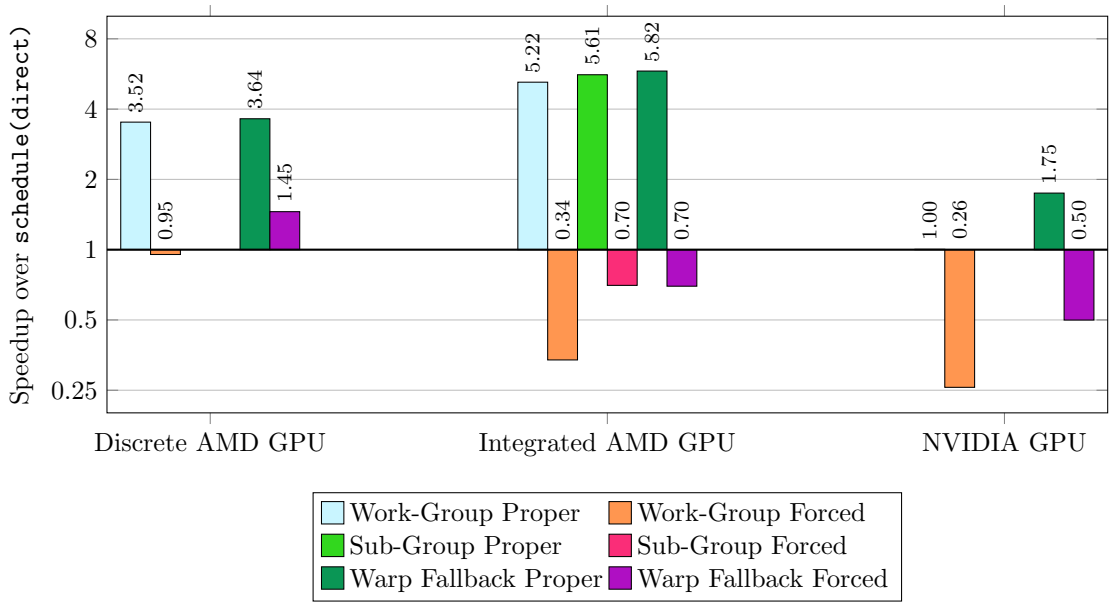


Figure 6.19: Comparison of multiple versions of the ATAX benchmark, with interleaved reductions, on multiple GPUs from systems 1 and 3, in terms of speedups over the direct schedule. The proper versions use the direct and cooperative schedules for the appropriate kernels, whereas the forced versions use the cooperative schedules for both ATAX kernels.

Finally this section explores whether the profitability of the cooperative schedule varies with input sizes, using the ATAX benchmark (see Figure 6.20). The profitability of the cooperative schedule does indeed depend on the input size, as well as the specific variant used. The results show that, for matrices of size 1024×1024, the cooperative schedule is already profitable, provided that the proper variant is used.

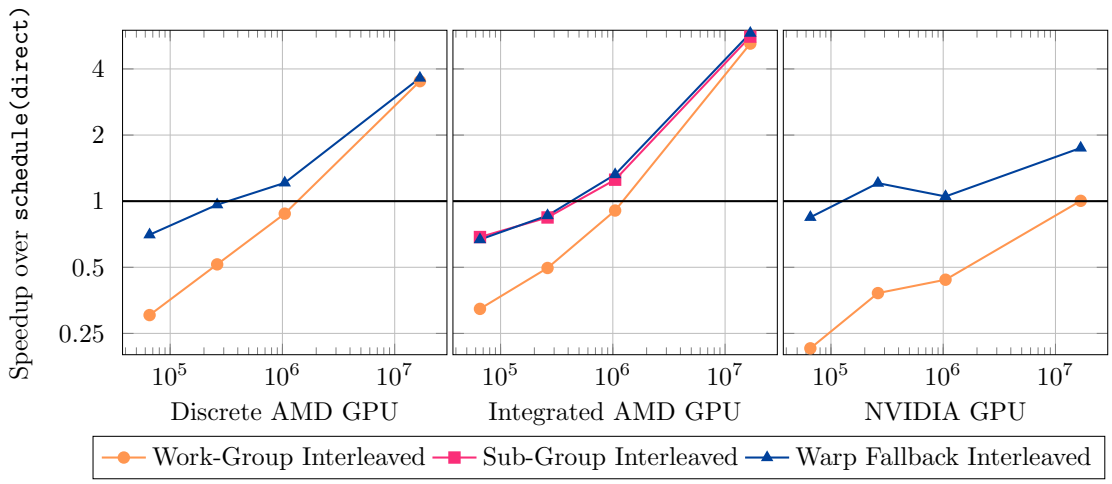


Figure 6.20: Comparison of multiple versions and input sizes of the ATAX benchmark, with interleaved reductions, on multiple GPUs from systems 1 and 3, in terms of speedups over the direct schedule.

MATISSE was configured to automatically use the cooperative schedule on the tested GPUs, specifically with the sub-groups as warps variant with interleaved reductions, when no schedule is explicitly specified (i.e., the `schedule(auto)` mode). MATISSE does not check the input sizes of the kernels (as these are often dynamic properties and the choice of schedule uses pure static analysis), but it does check whether the kernels are suitable for the cooperative schedule (i.e., it does not use the aforementioned *Forced* mode).

On CPUs, MATISSE does not use the cooperative schedule unless it is explicitly instructed to do so by the user, because there are performance improvements on any of the variants.

6.9 Alternative Schedules on AMD's CPU Platform

Although CPUs do not benefit from the cooperative schedule, it is conceivable that they could still benefit from other schedules. This section explores this topic, specifically on AMD's CPU platform (system 1), as Intel's auto-vectorization has implications for scheduling. This section is relative to kernel times only, as the schedule does not impact data transfer times or CPU times. All execution times are the average of 10 benchmark executions. The Complex Product benchmark uses matrices of input size $1024 \times 1024 \times 2$, Dilate uses 4096×4096 and Sub-band Coding 128×65536.

It was found that, in general, the sequential variants of schedules outperform the global rotation variants, and generally by a significant margin (up to 19.8× faster). In one case (the FDTD benchmark), the global rotation fixed work-groups schedule exceeded the specified timeout of 90 seconds (the real execution time is around 3 minutes),

whereas the other versions all had an average execution time under 30s. For the purposes of this section, this case is treated as if it had taken 90 seconds to execute. The sequential variant of the coarse schedule (with a coarsening factor of 4) is 22% faster (geometric mean) than the global rotation variant. The sequential variant of the fixed work-groups (with 16 work-groups) schedule is 2× faster than the global rotation variant. These results indicate that the sequential variants are a better default for CPUs than the global rotation variants. The exception being some examples with memory access patterns with poor locality (notably the artificial *No Interchange* benchmarks).

Next, it was attempted to determine which schedule category (direct, coarse or global rotation) is more suitable for the CPU. Figure 6.21 shows the speedups associated with using alternative schedules over the `schedule(direct)` version. In general, better results were obtained with the fixed work-groups versions (geometric mean of 1.41×) than with the coarse versions (geometric mean of 1.22×), though there were some benchmarks where the thread coarsened version outperformed the fixed work-groups version (notably the `No Interchange` and `Transpose` benchmarks).

Overall, between `schedule(direct)`, `schedule(coarse_sequential)`, and `schedule(fixed_work_groups_sequential)`, fixed work-groups tends to perform better on AMD's CPU platform. Based on these results, MATISSE's target-aware settings for AMD's CPU platform were modified to use the fixed work-groups by default.

6.10 Summary

This section presents the set of benchmarks, respective input sizes, and experiments that were used to evaluate the MATISSE prototype and techniques. The benchmarks were taken from a variety of sources, including previous work in MATISSE, the San Diego Vision Benchmark Suite and the Polybench/GPU benchmark suite. For some techniques, artificial benchmarks were also used.

The results include tests to measure the impact of specific subsystems of MATISSE, including temporary matrix elimination, parallelization, SVM and the execution schedules. Additionally, this section includes comparisons with the previous MATISSE OpenCL backend, sequential code, and manually-coded OpenCL.

Overall, these results suggest that the MATISSE techniques can help achieve significant speedups on a variety of cases, though in some cases there is margin for further improvement.

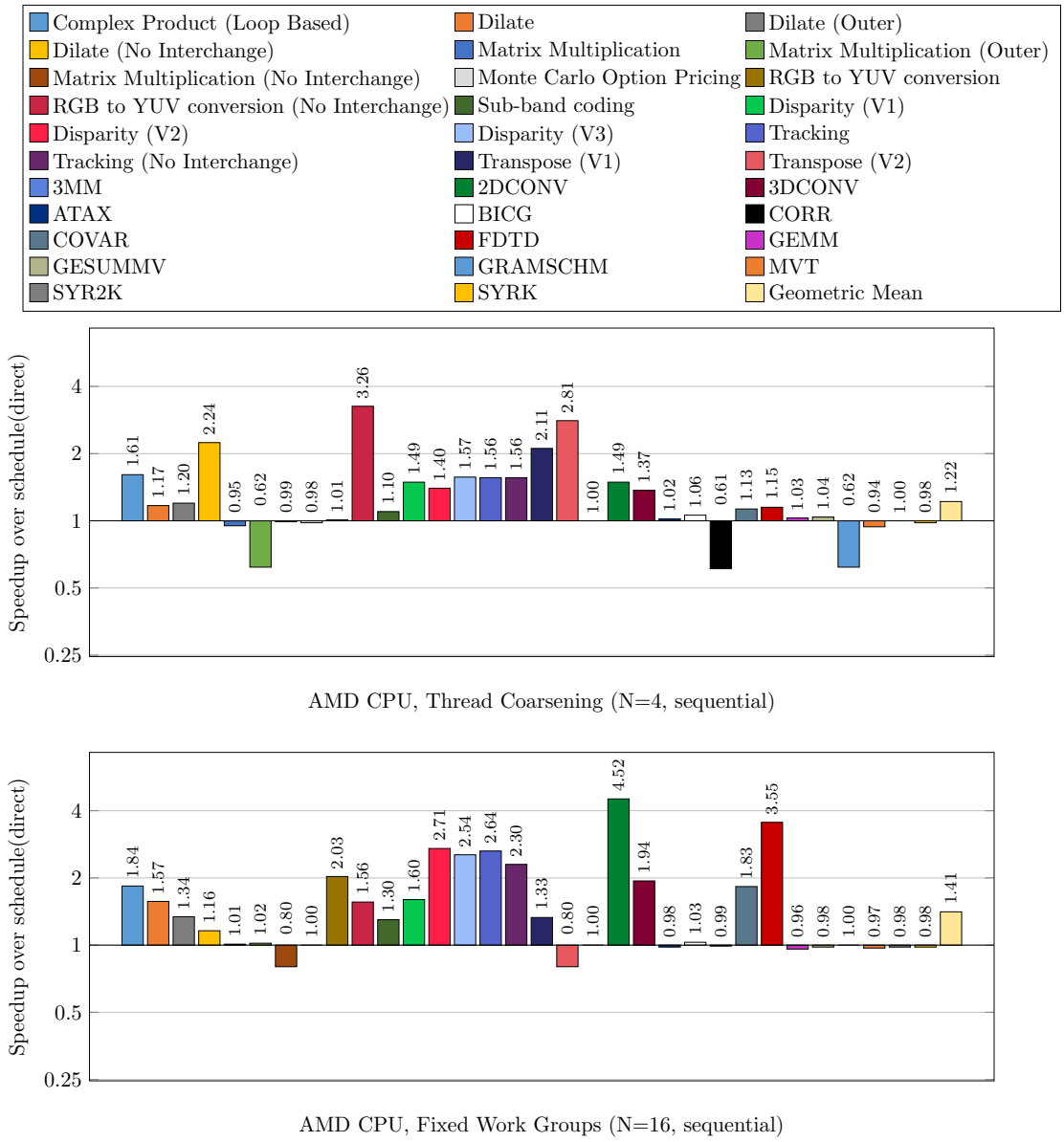


Figure 6.21: Speedup of using alternative schedules, relative to `schedule(direct)`. Only kernel times are considered.

7

Conclusion

Contents

This chapter summarizes the contents of this thesis, its main contributions, and presents possible future work.

7.1 Final Remarks

This thesis proposed compiler techniques to support the mapping of matrix-oriented computations to heterogeneous systems, mainly consisting of a CPU and a GPU. The proposed techniques simplify the development of efficient parallel programs, by designing a simple directive system for MATLAB and techniques to use those directives to parallelize the source code. The work done involved the research and development of target-aware optimizations to generate higher-quality OpenCL code for CPUs and GPUs. The viability and efficiency of those compiler optimizations were demonstrated by implementing them with a compiler prototype based on the MATISSE compiler framework [BPN+13], and evaluated on a set of various representative benchmarks, such as *Disparity* and *Tracking* from the San Diego Vision Benchmark Suite [VAJ+09], and Polybench/GPU [GGXS+12]. The optimizations are complemented by heuristics based on static program properties and information about the target device and target toolchain, in order to determine when to apply them. All of these techniques were integrated in a working compiler prototype that can be used to research and evaluate future optimizations and heuristics.

Among the researched compiler optimizations targeting CPUs and GPUs are the use of Shared Virtual Memory (SVM) and associated heuristics, as well as the use of alternative mappings of parallel loop iterations to work-items (i.e., *schedules*). Using SVM on a set of representative benchmarks allowed MATISSE to achieve geometric mean speedups on the generated code from 9% to 126%, depending on the target device, over no SVM usage, and from 9% to 80% over a naive use of SVM. Moreover, using the cooperative schedule optimization, MATISSE can generate more efficient code for GPUs, up to 38.8× faster in some cases. On AMD's CPU platform, MATISSE achieved a geometric mean speedup of 1.41 by using an alternative schedule by default (fixed work groups). This thesis also discusses how the Z3 SMT solver [DMB08] can be used to remove unnecessary bounds checking, identify loop carried dependences, and assist in determining when two or more matrices have the same size and shape.

The proposed techniques allow MATISSE to generate OpenCL code, from MATLAB that resembles a C coding style, that outperforms manually coded OpenCL code, with a geometric mean speedup of 1.19× in the Polybench/GPU benchmark suite when both kernel and data transfers are considered, MATISSE-generated code could outperform manually-coded OpenCL in 10 out of 15 benchmarks.

Compared to the previous MATISSE OpenCL backend, the new backend is able to generate code with significantly fewer data transfers, being able to achieve speedups of up to 2.5× compared to the previous backend even without target-aware optimizations being considered, while requiring significantly fewer and simpler directives.

7.2 Future Work

There are multiple avenues of research that could be explored with MATISSE, in order to further improve the efficiency of the generated code, and reduce the difficulty of producing efficient parallel code.

One of the most important features that MATISSE is missing is profitability analysis, in which a compiler automatically determines whether offloading certain sections of code to a co-processor (e.g., a GPU) would lead to speedups. Implementing profitability analysis techniques could lead to a further simplification of the directive system, as some of the directives (e.g., the `%!parallel` directives without parameters) could become unnecessary. As the decision to offload code to the GPU may depend on the size of the inputs, generating multiple versions might be necessary.

Moreover, the parallelization work in this thesis is based on parallel loops, as even element-wise vector computations are converted into loops. However, there are other mechanisms of parallelization that fit well in MATLAB and could be used. Many MATLAB parallelization approaches are based on matrix operations for which loop generation is unsuitable. For instance, although efficient code generation for matrix multiplication algorithms is difficult, high-quality BLAS (Basic Linear Algebra Subprograms) implementations already exist, so these can and should be leveraged for high-performance computing. The vectorized approach can be less flexible than our loop-based approach, but these two solutions are most likely not mutually exclusive. A hybrid approach that combines loop-based parallelization with specialized parallel implementations of certain built-in functions could be explored.

Another possible extension of this work include adding support for FPGAs through OpenCL High-Level Synthesis, as FPGAs have their own challenges and are still difficult to develop for. High-performance FPGA support would most likely require additional target-aware optimizations and heuristics, as indicated by our preliminary analysis [PRC17].

OpenMP generation is interesting not only because of its suitability for CPU parallelism, but also due to its support for offloading directives. This would enable a comparison between direct OpenCL generation and GPU OpenMP code generation. Furthermore, some programs benefit from running some kernels on the CPU and others on the GPU. OpenMP support could allow users to parallelize some of the CPU kernels, further improving performance.

Initial experiments [RNC18] suggest that support for half-precision data types with vectorization could improve the performance of the generated OpenCL on certain GPUs, but vectorization and half-precision types have not yet been integrated in MATISSE. Unfortunately, MATLAB code is generally written using single/double-precision floating point types, so adding support for half-precision could be of limited use without a means to automatically reduce the precision of certain variables or expressions and of analyzing the impact of reducing precision in the overall functionality.

Finally, an interesting topic of research would be automatizing the Foreign Function Interface (FFI) management, to facilitate calls to functions in external third-party C/OpenCL libraries. Integration of third-party libraries poses challenges, as the format of data structures of MATISSE-generated code and custom C libraries can and often does differ, so FFI management requires mechanisms to specify interfaces. Moreover, the differences in semantics between C and MATLAB introduce memory lifetime-related issues that are complex to address. Finally, the interoperability code itself can add overhead.

References

[ABDFP15] Giovanni Agosta, Alessandro Barenghi, Alessandro Di Federico, and Gerardo Pelosi. OpenCL performance portability for general-purpose computation on graphics processor units: an exploration on cryptographic primitives. *Concurrency and Computation: Practice and Experience*, 27(14):3633–3660, 2015.

[Adv12] Advanced Micro Devices, Inc. *Southern Islands Series Instruction Set Architecture*, December 2012. https://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf.

[Adv15] Advanced Micro Devices, Inc. *AMD OpenCL™ Optimization Guide*, August 2015. http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf.

[Adv17] Advanced Micro Devices, Inc. The AMD OpenCL™ Zone. http://developer.amd.com/tools-and-sdks/opencl-zone/, 2017. Accessed: November 9th, 2017.

[Adv18] Advanced Micro Devices, Inc. GN Architecture — AMD. https://www.amd.com/en/technologies/gcn, 2018. Accessed: September 24th, 2018.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.

[ARM17a] ARM Limited. *ARM® Mali™ GPU OpenCL Developer Guide*, February 2017. http://infocenter.arm.com/help/topic/com.arm.doc.100614_0303_00_en/arm_mali_gpu_opencl_developer_guide_100614_0303_00_en.pdf.

[ARM17b] ARM Limited. Technologies — NEON - Arm Developer. https://developer.arm.com/technologies/neon, 2017. Accessed: November 13th, 2017.

[ARM19] ARM. Graphics and Multimedia Processors — Mali GPUs – Arm Developer. https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus, 2019. Accessed: April 27th, 2019.

[BC17] João Bispo and João M. P. Cardoso. A MATLAB subset to C compiler targeting embedded systems. *Software: Practice and Experience*, 47(2):249–272, 2017.

[BDR+09] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[BHS13] André R. Brodtkorb, Trond R. Hagen, and Martin L. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4 – 13, 2013.

[BPN+13] João Bispo, Pedro Pinto, Ricardo Nobre, Tiago Carvalho, João M.P. Cardoso, and Pedro C. Diniz. The MATISSE MATLAB compiler. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pages 602–608, July 2013.

[BRC15a] João Bispo, Luís Reis, and João M. P. Cardoso. C and OpenCL Generation from MATLAB. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1315–1320, New York, NY, USA, 2015. ACM.

[BRC15b] João Bispo, Luís Reis, and João M. P. Cardoso. Techniques for Efficient MATLAB-to-C Compilation. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 7–12, New York, NY, USA, 2015. ACM.

[CBHV10] Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In Rajiv Gupta, editor, *Compiler Construction*, pages 46–65, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[CBP+16] Tiago Carvalho, João Bispo, Pedro Pinto, Luís Reis, Ricardo Nobre, and João M. P. Cardoso. The lara-based compiler toolsuite. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'16)*, Santa Barbara, California, USA, 2016.

[CDCP13] João M. P. Cardoso, Pedro C. Diniz, José G. F. Coutinho, and Zlatko Petrov, editors. *Compilation and Synthesis for Embedded Reconfigurable Systems: An Aspect-Oriented Approach*. Springer, 2013.

[Cha13] Arun Chauhan. HLLC / ParaM. http://www.cs.indiana.edu/~achauhan/Software/hllc.html, April 2013. Accessed: February 28th, 2018.

[DGHS17] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. Inside Volta: The World's Most Advanced Data Center GPU — Parallel Forall. https://devblogs.nvidia.com/parallelforall/inside-volta/, May 2017. Accessed: November 11th, 2017.

[DH12a] Jesse Doherty and Laurie Hendren. Mcsaf: A static analysis framework for matlab. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 132–155, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[DH12b] Anton Willy Dubrau and Laurie Jane Hendren. Taming matlab. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 503–522, New York, NY, USA, 2012. ACM.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[DRP99] Luiz De Rose and David Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. Program. Lang. Syst.*, 21(2):286–323, March 1999.

[DWL+12] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[Edd17] Steve Eddins. Deep Learning with MATLAB R2017b - Deep Learning - MATLAB & Simulink. https://blogs.mathworks.com/deep-learning/2017/10/06/deep-learning-with-matlab-r2017b/, 2017. Accessed: July 28th, 2018.

[FBH16] Vincent Foley-Bourgon and Laurie Hendren. Efficiently Implementing the Copy Semantics of MATLAB's Arrays in JavaScript. *SIGPLAN Not.*, 52(2):72–83, November 2016.

[FSV14] Jianbin Fang, Henk Sips, and Ana Varbanescu. Aristotle: A performance Impact Indicator for the OpenCL Kernels Using Local Memory. *Sci. Program.*, 22:239–257, 01 2014.

[GGXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.

[GH14] Rahul Garg and Laurie Hendren. Velociraptor: An Embedded Compiler Toolkit for Numerical Programs Targeting CPUs and GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 317–330, New York, NY, USA, 2014. ACM.

[GP-12] GP-you Group. *GPUmat User Guide*. GP-you Group, January 2012. Version 0.28. Available at http://svn.code.sf.net/p/gpumat/code/trunk/doc/GPUmat_User_Guide.pdf. Accessed: February 28th, 2018.

[GP-15] GP-you Group. GPUmat download — SourceForge. http://sourceforge.net/projects/gpumat/, 2015. Accessed: February 28th, 2018.

[Har13] Mark Harris. Unified Memory in CUDA 6 — Parallel Forall. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/, November 2013. Accessed: November 8th, 2017.

[Har15] Mark Harris. New Features in CUDA 7.5 — Parallel Forall. https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/, July 2015. Accessed: November 11th, 2017.

[Har16] Mark Harris. Mixed-Precision Programming with CUDA 8 — Parallel Forall. https://devblogs.nvidia.com/parallelforall/mixed-precision-programming-cuda-8/, December 2016. Accessed: November 11th, 2017.

[Har18] Peter Harris. Graphics and Multimedia Development — The Bifrost Shader Core – Arm Developer. https://developer.arm.com/graphics/developer-guides/the-bifrost-shader-core, March 2018. Accessed: September 24th, 2018.

[Hc14] Ltd. Hardkernel co. ODROID-XU+E. http://www.hardkernel.com/main/products/prdt_info.php?g_code=G137463363079, 2014. Accessed: April 10th, 2017.

[IBM18] IBM. The X10 Parallel Programming Language. http://x10-lang.org/, 2018. Accessed: November 19th, 2018.

[IEC18] IEC. IEC – SI Zone ¿ Prefixes for binary multiples. https://www.iec.ch/si/binary.htm, 2018. Accessed: November, 6th 2018.

[Int09] Intel Corporation. Intel® Xeon® Processor X5550. https://ark.intel.com/products/37106/Intel-Xeon-Processor-X5550-8M-Cache-2_66-GHz-6_40-GTs-Intel-QPI, 2009. Accessed: September 26th, 2018.

[Int12] Intel Corporation. Multi-core Introduction. https://software.intel.com/en-us/articles/multi-core-introduction, March 2012. Accessed: October 1st, 2018.

[Int14] Intel Corporation. OpenCL Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor. https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi- coprocessor , January 2014. Accessed: November 12th, 2017.

[Int15] Intel Corporation. Memory Access Overview — Intel® Software. https://software.intel.com/en-us/node/540444, February 2015. Accessed: November 8th, 2018.

[Int17a] Intel Corporation. Improve Performance Using Vectorization and Intel® Xeon® Scalable Processors. https://software.intel.com/en-us/articles/improve-performance-using-vectorization-and-intel-xeon-scalable-processors, October 2017. Accessed: September 24th, 2018.

[Int17b] Intel Corporation. Intel FPGA SDK for OpenCL - Overview. https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, November 2017. Accessed: November 9th, 2017.

[Int17c] Intel Corporation. Intel® Hyper-Threading Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html, September 2017. Accessed: November 13th, 2017.

[Int17d] Intel Corporation. Intel® Turbo Boost Technology 2.0. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html, September 2017. Accessed: November 13th, 2017.

[Int17e] Intel Corporation. Work-Group Size Recommendations Summary — Intel® Software. https://software.intel.com/en-us/node/540442 Version 3.3, 2017. Accessed: November 11th, 2017.

[Int19] Intel Corporation. Unleash Your Data Center - For Intel Xeon CPU with FPGAs. https://www.intel.com/content/www/us/en/data-center/products/programmable/overview.html, 2019. Accessed: April 27th, 2019.

[JB07] Pramod G Joisha and Prithviraj Banerjee. A translator system for the MATLAB language. *Software: Practice and Experience*, 37(5):535–578, 2007.

[KA01] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*, pages 302–304. Morgan Kaufmann Publishers Inc., 2001.

[Kar14] Rama Karedla. Intel Xeon E5-2600 v3 (Haswell) Architecture & Features. http://repnop.org/pd/slides/PD_Haswell_Architecture.pdf, 2014. Accessed: February 20th, 2018.

[KBR14] John Kessenich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language. http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf, 2014. Accessed: July 7th, 2014.

[KH14] Vineet Kumar and Laurie Hendren. MIX10: Compiling MATLAB to X10 for High Performance. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 617–636, New York, NY, USA, 2014. ACM.

[Khr15] Khronos OpenCL Working Group. *The OpenCL Specification*. The Khronos Group Inc., November 2015. http://www.khronos.org/registry/cl/specs/opencl-2.1.pdf Version: 2.1, Document Revision: 23.

[Khr16] Khronos OpenCL Working Group. *The OpenCL C Specification*. The Khronos Group Inc., April 2016. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0-openclc.pdf Version: 2.0, Document Revision: 33.

[Kon12] Patrick Konsor. Performance Benefits of Half Precision Floats — Intel® Software. https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats, August 2012. Accessed: September 11th, 2017.

[LH14] Xu Li and Laurie Hendren. Mc2FOR: A tool for automatically translating MATLAB to FORTRAN 95. pages 234–243. IEEE, 2014.

[Li17] Kelvin Li. *OpenMP Accelerator Support for GPUs*, September 2017. Accessed: November 9th, 2017.

[Lom11] Chris Lomont. Introduction to Intel® Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions, June 2011. Accessed: November 13th, 2017.

[LPD+16] Ioannis Latifis, Karthick Parashar, Grigoris Dimitroulakos, Hans Cappelle, Christakis Lezos, Konstantinos Masselos, and Francky Catthoor. Matlab to c compilation targeting application specific instruction set processors. In *2016 Design, Automation and Test in Europe Conference Exhibition (DATE)*, pages 1453–1456, March 2016.

[LPD+17] Ioannis Latifis, Karthick Parashar, Grigoris Dimitroulakos, Hans Cappelle, Christakis Lezos, Konstantinos Masselos, and Francky Catthoor. A MATLAB Vectorizing Compiler Targeting Application-Specific Instruction Set Processors. *ACM Trans. Des. Autom. Electron. Syst.*, 22(2):32:1–32:28, January 2017.

[Lui17] Justin Luitjens. CUDA Pro Tip: Increase Performance with Vectorized Memory Access — Parallel Forall. https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, May 2017. Accessed: November 11th, 2017.

[Mat13a] MathWorks. MATLAB - the language of technical computing. http://www.mathworks.com/products/matlab/, 2013. Accessed: November 20th, 2013.

[Mat13b] MathWorks. MATLAB GPU computing support for NVIDIA CUDA-enabled GPUs. http://www.mathworks.com/discovery/matlab-gpu.html, 2013. Accessed: February 26th, 2018.

[Mat14a] MathWorks. Matrices and Arrays - MATLAB & Simulink. http://www.mathworks.com/help/matlab/learn_matlab/matrices-and-arrays.html, 2014. Accessed: June 26th, 2014.

[Mat14b] MathWorks. System Requirements - Release R2017b. http://www.mathworks.com/support/sysreq/current_release/index.html, 2014. Accessed: November, 8th 2017.

[Mat14c] MathWorks. Using GPU ARRAYFUN for monte-carlo simulations - MATLAB & simulink example. http://www.mathworks.com/help/distcomp/examples/using-gpu-arrayfun-for-monte-carlo-simulations.html, 2014. Accessed: Feb 13th, 2017.

[Mat16] MathWorks Support Team. Should I use MATLAB Compiler SDK, or MATLAB Coder to integrate my MATLAB applications with C/C++? https://www.mathworks.com/matlabcentral/answers/223937-should-i-use-matlab-compiler-sdk-or-matlab-coder-to-integrate-my-matlab-applications-with-c-c#answer_182772, 2016. Accessed: November, 15th 2018.

[Mat17a] MathWorks. Preallocation - MATLAB & Simulink. https://www.mathworks.com/help/matlab/matlab_prog/preallocating-arrays.html, 2017. Accessed: November 8th, 2017.

[Mat17b] MathWorks. Vectorization - MATLAB & Simulink. https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html, 2017. Accessed: November 8th, 2017.

[Mat18a] MathWorks. Compilation Directive %#codegen - MATLAB & Simulink. https://www.mathworks.com/help/simulink/ug/adding-the-compilation-directive-codegen.html, 2018. Accessed: July 27th, 2018.

[Mat18b] MathWorks. Decide when to use parfor - MATLAB & Simulink. https://www.mathworks.com/help/distcomp/decide-when-to-use-parfor.html, 2018. Accessed: February 27th, 2018.

[Mat18c] MathWorks. GPU Coder – Supported Functions. https://www.mathworks.com/content/dam/mathworks/tag-team/Objects/g/GPU_Coder_Supported_Functions.pdf, 2018. Accessed: July 27th, 2018.

[Mat18d] MathWorks. GPU Coder - MATLAB & Simulink. https://www.mathworks.com/products/gpu-coder.html, 2018. Accessed: July 27th, 2018.

[Mat18e] MathWorks. MATLAB Coder - MATLAB. http://www.mathworks.com/products/matlab-coder/, 2018. Accessed: July 25th, 2018.

[MDO14] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 455–466, New York, NY, USA, 2014. ACM.

[Mes15] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf Version: 3.1, June 2015. Accessed: September 27th, 2018.

[MGY+15] Saoni Mukherjee, Xiang Gong, Leiming Yu, Carter McCardwell, Yash Ukidave, Tuan Dao, Fanny Nina Paravecino, and David Kaeli. Exploring the Features of OpenCL 2.0. In *Proceedings of the 3rd International Workshop on OpenCL*, IWOCL '15, pages 5:1–5:5, New York, NY, USA, 2015. ACM.

[Mic14] Microsoft. HLSL (Windows). http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561%28v=vs.85%29.aspx, 2014. Accessed: July 7th, 2014.

[MS09] Jiayuan Meng and Kevin Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 256–265, New York, NY, USA, 2009. ACM.

[ND10] J. Nickolls and W.J. Dally. The GPU Computing Era. *Micro, IEEE*, 30(2):56–69, March 2010.

[NVI12] NVIDIA Corporation. The Cg Toolkit. https://developer.nvidia.com/cg-toolkit, 2012. Accessed: July 7th, 2014.

[NVI14a] NVIDIA Corporation. CUDA FAQ. https://developer.nvidia.com/cuda-faq, 2014. Accessed: June 20th, 2014.

[NVI14b] NVIDIA Corporation. Parallel programming and computing platform — CUDA — NVIDIA — NVIDIA. http://www.nvidia.com/object/cuda_home_new.html, 2014. Accessed: June 20th, 2014.

[NVI17a] NVIDIA Corporation. Best Practices Guide :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, September 2017. v9.0.176, Accessed: November 10th, 2017.

[NVI17b] NVIDIA Corporation. GPU vs CPU? What is GPU Computing? — NVIDIA. http://www.nvidia.com/object/what-is-gpu-computing.html, 2017. Accessed: November 13th, 2017.

[NVI17c] NVIDIA Corporation. OpenCL — NVIDIA Developer. https://developer.nvidia.com/opencl, 2017. Accessed: November 9th, 2017.

[NVI17d] NVIDIA Corporation. Programming Guide :: CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2017. v9.0.176, Accessed: November 8th, 2017.

[NVI18a] NVIDIA. CUDA Parallel Computing - GPU Computing on the CUDA Architecture. http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html, 2018. Accessed: October 1st, 2018.

[NVI18b] NVIDIA Corporation. PTX ISA :: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html, September 2018. v10.0.130, Accessed: September 25th, 2018.

[Oct14a] Octave community. GNU octave. http://www.gnu.org/software/octave/, 2014. Accessed: June 28th, 2014.

[Oct14b] Octave community. How is Octave different from Matlab? - FAQ - Octave. http://wiki.octave.org/FAQ#Porting_programs_from_Matlab_to_Octave, 2014. Accessed: June 28th, 2014.

[Ope15] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, November 2015. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf Version: 4.5.

[Ope17] OpenACC. *The OpenACC® Application Program Interface*, November 2017. https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf Version: 2.6.

[PAG11] Ashwin Prasad, Jayvant Anantpur, and R. Govindarajan. Automatic Compilation of MATLAB Programs for Synergistic Execution on Heterogeneous Processors. *ACM SIGPLAN Not.*, 46(6):152–163, June 2011.

[PG12] Ashwin Prasad and R. Govindarajan. Compiler optimizations to execute MATLAB programs on memory constrained GPUs. In *First Asia-Pacific Programming Languages and Compilers Workshop (APPLC 2012)*, 2012.

[PHW+13] Simon J. Pennycook, Simon D. Hammond, Steven A. Wright, J. A. Herdman, Iain Miller, and Stephen A. Jarvis. An investigation of the performance portability of OpenCL. *Journal of Parallel and Distributed Computing*, 73(11):1439 – 1450, 2013. Novel architectures for high-performance computing.

[PRC17] Nuno Paulino, Luís Reis, and João M. P. Cardoso. On Coding Techniques for Targeting FPGAs via OpenCL. In *Proceedings of the conference Parallel Computing 2017*, 2017.

[RBC16] Luís Reis, João Bispo, and João M. P. Cardoso. SSA-based MATLAB-to-C Compilation and Optimization. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 55–62, New York, NY, USA, 2016. ACM.

[Rei14] Luís Reis. Optimization and Generation of OpenCL Code for Embedded Computing. Master's thesis, MIEIC, Faculty of Engineering of the University of Porto, July 2014.

[RNC18] Luís Reis, Ricardo Nobre, and João M. P. Cardoso. Impact of Vectorization Over 16-bit Data-Types on GPUs. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '18, pages 32–38, New York, NY, USA, 2018. ACM.

[RWZ88] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM, 1988.

[Sab18] Sable Research Group. The McLab Project. http://www.sable.mcgill.ca/mclab/, 2018. Accessed: November, 13th 2018.

[SCS16] Johannes Spazier, Steffen Christgau, and Bettina Schnor. Automatic Generation of Parallel C Code for Stencil Applications Written in MATLAB. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 47–54, New York, NY, USA, 2016. ACM.

[SFSV13] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. An application-centric evaluation of OpenCL on multi-core CPUs. *Parallel Computing*, 39(12):834 – 850, 2013.

[Shu12] Loren Shure. Understanding Array Preallocation - Loren on the Art of MATLAB. http://blogs.mathworks.com/loren/2012/11/29/understanding-array-preallocation/, 2012. Accessed: March 23rd, 2016.

[Shu16a] Loren Shure. MathWorks - MATLAB and SimuLink for Technical Computing - B. http://www.mathworks.com/, 2016. Accessed: July 4th, 2014.

[Shu16b] Loren Shure. Run Code Faster With the New MATLAB Execution Engine - MATLAB & Simulink. https://blogs.mathworks.com/loren/2016/02/12/run-code-faster-with-the-new-matlab-execution-engine, 2016. Accessed: October 24th, 2018.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, pages 194–210, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Sof14] Intel Software. Opencl™ 2.0 shared virtual memory overview. https://software.intel.com/en-us/articles/opencl-20-shared-virtual-memory-overview, 2014. Accessed: June 15th, 2017.

[SRC11] Chun-Yu Shei, Pushkar Ratnalikar, and Arun Chauhan. Automating GPU Computing in MATLAB. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 245–254, New York, NY, USA, 2011. ACM.

[SSCP14] Diogo Sampaio, Rafael Martins de Souza, Sylvain Collange, and Fernando Magno Quintão Pereira. Divergence analysis. *ACM Trans. Program. Lang. Syst.*, 35(4):13:1–13:36, January 2014.

[Sut04] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. http://www.gotw.ca/publications/concurrency-ddj.htm, December 2004. Accessed: September 13th, 2017.

[Sym16] Symja. Symja - Java Computer Algebra Library. https://bitbucket.org/axelclk/symja_android_library/wiki/Home, 2016. Accessed: March 23rd, 2016.

[SYRC11] Chun-Yu Shei, Adarsh Yoga, Madhav Ramesh, and Arun Chauhan. MATLAB Parallelization through Scalarization. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 44–53, Feb 2011.

[The08] The Khronos Group Inc. The Khronos Group Releases OpenCL 1.0 Specification. http://www.khronos.org/news/press/the_khronos_group_releases_opencl_1.0_specification, December 2008. Accessed February 7th, 2014.

[The09] The Khronos Group Inc. clGetEventProfilingInfo – OpenCL 1.0 Reference Pages. https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clGetEventProfilingInfo.html, 2009. Accessed: October 29th, 2018.

[The13] The Khronos Group Inc. Specifying Attribute For Unrolling Loops – OpenCL 2.0 Reference Pages. https://www.khronos.org/registry/OpenCL/sdk/2.0/docs/man/xhtml/attributes-loopUnroll.html, 2013. Accessed: October 29th, 2018.

[The17] The Khronos Group Inc. SPIR Overview - The Khronos Group Inc. https://www.khronos.org/spir/, 2017. Accessed: November 9th, 2017.

[The18] The SMT-LIB Initiative. SMT-LIB The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/index.shtml, 2018. Accessed: November, 8th 2018.

[VAJ+09] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS: The San Diego Vision Benchmark Suite. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 55–64, Washington, DC, USA, 2009. IEEE Computer Society.

[Val18] Valgrind™. Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html, 2018. Accessed: November, 6th 2018.

[WAG09] Jian-Zhong Wang, Bastiaan Aarts, and Vinod Grover. Loop unroll pragma extension. https://www.khronos.org/registry/OpenCL/extensions/nv/cl_nv_pragma_unroll.txt, 2009. Accessed: October 29th, 2018.

[Xil17] Xilinx Inc. SDAccel Development Environment. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html, 2017. Accessed: November 9th, 2017.

Appendix A

Compiler Usage Manual

Contents

This appendix describes how to use the MATISSE compiler. Currently, the compiler is designed around the execution of benchmark and validation test files, and compiling full *applications* or *libraries* is more complex. Regardless, this section shows how to do so as well.

A.1 Basic Usage

The behavior of MATISSE is controlled by *setup files*, files that describe which files should be compiled, the types of each variable, which LARA aspects should be used, etc. The preferred mechanism to create these setup files is to use MATISSE's visual interface, shown in Figure A.1. Users can edit each property and then save the setup file. All folder paths are relative to the location of the setup file.

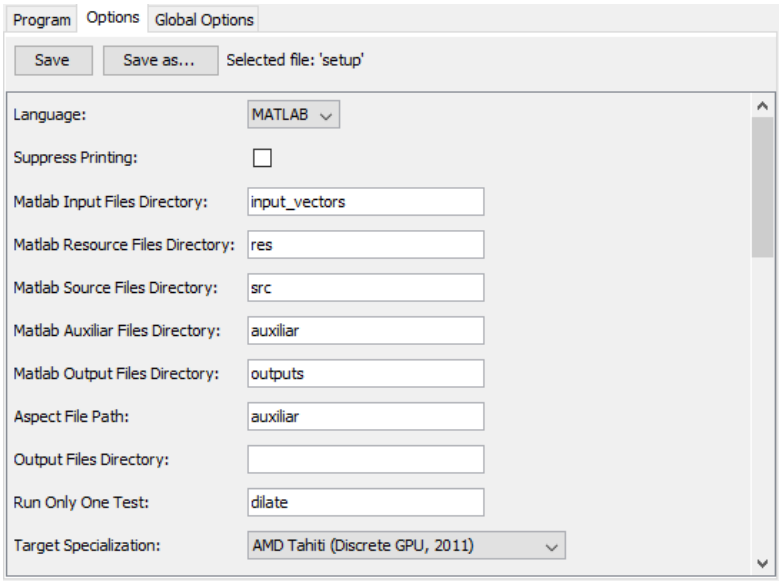


Figure A.1: MATISSE's Graphical User Interface for editing setup files.

Among the most important properties to set are:

Source Files Directory The location of the *main* source code files to compile. Each file represents a different benchmark or test.

Input Files Directory The location of the input files (in `*.mat` or `*.m` files). MATISSE searches for input files in the `<input files directory>/<test name>/` directory, and generates a separate source code project and binary for each test file. These files are used as the program inputs, as well as to determine the types of the inputs of the *main* MATLAB function.

Aspect File Path Path of the LARA aspect file to execute. If empty, no user aspect file is executed. MATISSE assumes that each LARA aspect file contains an aspect with the same name as the file, and that is the one to be executed.

Auxiliar Files Directory The *source files directory* stores the location of MATLAB files, but only those that should be directly executed as a benchmark/test. MATLAB files that are used by other functions, but are not meant to be called directly, are placed in this directory, in `<auxiliar files directory>/<test name>`.

Run Only One Test If only a single file in the source files directory should be compiled, that file should be specified here.

Target Specialization The OpenCL device to target.

Output Files Directory Directory containing the correct results of each test, if validation is desired.

Disable Parallelism Indicates whether MATISSE can generate parallel code, as opposed to sequential code only.

Compiler The compiler to use to generate the final binary from the source code files. Usually `gcc`.

Enable Z3 If true, MATISSE uses Z3 for its scalar solver. See Section 5.2.

Assume Matrix Indices In Range True if all matrix accesses can be assumed to be in range. Equivalent to adding `%!assume_indices_in_range` directory to all functions.

Assume Matrix Sizes Match True if all matrices in matrix operations can be assumed to have the same sizes (e.g., in `A + B`, A and B have the same size). Equivalent to adding `%!assume_matrix_sizes_match` directory to all functions.

Once the setup file has been generated, MATISSE can compile the MATLAB files, either in graphical mode, or in command line mode. It is possible to run MATISSE in command-line mode, using the command `java -jar matisse.jar <setupfile>`.

A.2 Building Applications and Libraries

MATISSE does not feature any dedicated *application* or *library* building mode, but it is still possible to create one nevertheless. For applications, the developer can create a *main* MATLAB function without arguments/outputs and an empty input file. For libraries with a single externally visible function, the following is the recommended approach to use:

- Annotate the externally visible function with the `%!export` directive (see Subsection 4.1.2);
- Create an input file containing an example of each of the arguments (e.g., matrices with the proper number of dimensions);
- Create an aspect file with the types of arguments if more specific type/shape information is wanted (e.g., if the matrices *always* have a fixed shape);

- Delete `main_test.c`, `MATISSE_init.c` and associated headers from the generated C files (as they are not relevant to this use case);
- Use the remaining C files to build the library.

Note that if OpenCL code generation is enabled, programs using the compiled library should call the `MATISSE_cl_initialize` before any other MATISSE-generated functions.

For libraries with multiple externally visible functions:

- Annotated every externally visible function with the `%!export` directive.
- Create an input file containing an example of each of the arguments that the exported function uses.
- Create a *glue* function taking all of the example inputs, invoking every exported function, and returning all of the outputs. An example of this function can be seen in Figure A.2. Note that it does not need to be annotated with the `%!export` directive. Returning the outputs of each function call is important, as otherwise MATISSE may eliminate these functions
- Delete `main_test.c`, `MATISSE_init.c`, the glue function and associated headers from the generated C files.
- Use the remaining C files to build the library.

```
1 function [f_out1, f_out2, g_out] = glue(f_in, g_in1, g_in2)
2         [f_out1, f_out2] = f(f_in);
3         [g_out] = g(g_in1, g_in2);
4 end
```

Figure A.2: MATISSE function used to build a library that exports functions `f` and `g`.

A.3 Custom Phase Orders

The MATISSE compiler is designed to support custom phase orders of the SSA IR passes executed after type inference, enabling the user to tune the compiler.

Some SSA IR passes, such as conversion to CSSA and the transformation of some SSA IR instructions into simpler operations, are necessary for the proper functioning of the compiler. These passes are executed at the end of the user-specified phase order, regardless of user input.

The recommended mechanism for using custom phase orders is:

1. Compile a MATLAB program with the default phase order.
2. Copy the `post-type-passes.recipe` file from the MATISSE output folder to the folder containing the setup file. This file contains the default phase order.
3. Modify the contents of the new file to contain the intended phase order;

4. Modify the *setup file* to specify the new file as the *Custom Recipe File*.

Figure A.3 presents an example of a file specifying a custom phase order. In this example, the following passes are executed: constant branch elimination, dead code elimination, and SSA validation. The `validator-name` parameter of the validation pass indicates a label that MATISSE should print should the validation fail. This is useful when this pass is executed multiple times, to determine which execution failed.

```
1 !typed-ssa v2
2
3 ConstantBranchEliminationPass
4 DeadCodeEliminationPass
5 SsaValidatorPass: validator-name="after-pass"
6 # some passes are always applied after this and can't be disabled.
```

Figure A.3: Example of a custom phase order.

Appendix B

MATISSE SSA IR Instructions

This appendix lists the main MATISSE SSA IR instructions and their semantics. Table B.1 includes the instructions used for sequential code, as well as the ones used for parallelism.

A description of the MATISSE SSA IR itself is presented in Subsection 4.2.3.

Table B.1: List of MATISSE SSA IR instructions and their semantics

Name	Description
General	
`line <NUM>`	Original line of the next instructions
`% comment`	Program comment (can be safely ignored)
`$out = phi <SOURCED_INPUTS>`	SSA-standard ϕ node. Inputs in format `#blockId:$varName`.
`$out = $in`	Assign a variable to another variable.
`$out = <NUM>`	Assign a number literal to a variable.
`$out = str '<STR>'`	Assign a string literal to a variable.
`$out = !undefined`	Define a variable with an undefined value.
`$out = init`	Initialize variable in OpenCL kernel. The SSA IR is unaware of how this is done.
`<OUTS> = parallel_copy <INPUTS>`	Used only for out-of-SSA translation, as described by Boissinot et al. [BDR+09].
Control Flow	
`branch $c, #if, #else, #end`	Conditional program execution (condition c)
`while #body, #end`	A repeated (`while 1`) loop
`for $s, $i, $e, #body, #end`	Range for loop (`s:i:e`)
`iter`	Obtain the current `for` loop iteration
`break`	Breaks the current loop
`continue`	Continue to the next loop iteration
Function Calls	
`$out = arg <ID>`	Obtain value of function argument
`<OUTS> = untyped_call f <IN>`	Call function `f` (before type inference)
`<OUTS> = call f [R] <IN>`	Call function `f` (returns a value of type `R`)
Matrix Accesses	
`$out = get $m, <IDXS>`	Equivalent to MATLAB's matrix get.
`$out = simple_get $m, <IDXS>`	An in-range matrix access with scalar indices.
`$out = get_or_first $m, $i`	If `$m` is a scalar, then do `$out = $m` Otherwise, do a `simple_get $m, $i`.
`$m2 = set $m1, <IDXS>, $v`	Equivalent to MATLAB's matrix set.
`$m2 = simple_set $m1, <IDXS>, $v`	An in-range matrix set with scalar indices.
`$m2 = multi_set $m1, <VALS>`	Sets multiple (consecutive, starting at 1) positions.
`$out = range_get $m, <IDXS>`	Similar to `get`, but indices can be ranges instead of SSA variables.
`$m2 = range_set $m1, <IDXS>, $v`	Similar to `set`, but indices can be ranges instead of SSA variables.
`$m2 = set_all $m1, $v`	Sets all elements of a matrix to a given scalar value.

Name	Description
`$out = relative_get $m, $dims <IDXS>`	Accesses a matrix using the dimensions of another matrix. Used to implement statements such as `A(:, :) = matrix;`.
Matrix Sizing	
`$out = end $m, <IDX>, <NUM_IDXS>`	Gets the value of `end` in MATLAB, for the `<IDX>`'th index of an access to `$m` with `<NUM_IDXS>` indices.
`$out = combine_size <INS>`	Obtain the size matrix for the element-wise combination of the given matrices.
`$out = vertical_flatten $in`	Equivalent to `out = in(:);`.
`$out = access_size $m, $i`	Equivalent to `out = size(B(I));`.
Validation	
`val_boolean $in`	Raise an error if `X` can not be used as a boolean.
`val_at_least_one_empty_matrix <INS>`	Raise an error if none of the inputs are empty.
`val_same_size <INS>`	Raise an error if the inputs do not all have the same size.
`val_true $in`	Raise an error if `$in` is not true.
Host/Device Data Management	
`$out = allocate_on_gpu $in`	Allocate a buffer on the GPU, given the size of `$in`.
`$out = copy_to_gpu $in`	Copy a matrix to the GPU, creating a new buffer.
`$buffer = allocate_global_reduction_buffer DataType, $numElements`	Allocate a buffer to be used to compute a reduction.
`$out = complete_reduction Type $buffer, $numGroups, $initial`	Completes a reduction, given a reduction type, reduction buffer, number of work-groups and initial host (CPU) value.
Work-item/Work-group Counting	
`$o = compute_group_size $n, $b`	Computes how many work-groups are necessary, for a kernel with n tasks with a work-group size of b.
`$o = use_group_size $i`	Cast a number to a work-group size. Used for the fixed work-groups schedules (see Section 5.5).
`$o = compute_global_size $b, $n`	Computes the global size of the kernel, with work-group size b and n work-groups.
Parallel Execution	
`parallel_block [s] #c, #e`	Identifies a block `#c` where MATISSE should search for parallelizable code, given settings s. Afterwards, executing proceeds to block `#e`
`set_gpu_range $i, $b, $e, $v`	Fill buffer i (positions b to e) with the given value v. Has an output when using Shared Virtual Memory.

Name	Description
`<OUTS> = invoke_parallel s <INS>`	Invoke a parallel section s.
`<OUTS> = invoke_kernel k S <INS>`	Invoke kernel k, with S work-items and the given arguments. Outputs are only used for Shared Virtual Memory.

Appendix C

MATISSE Execution Schedules

This appendix lists the full set of *execution schedules* supported by MATISSE in Table C.1. A more in-depth discussion of the purpose and impact of schedules is presented in Sections 5.5 and 5.6.

Table C.1: Description of available MATISSE execution schedules.

Name	Description
	Main
`auto`	MATISSE automatically determines which schedule should be used, depending on the target device, and loop to parallelize. This is the default schedule.
`direct`	Each work-item processes exactly 1 task, in order. The total number of work-items is the number of tasks.
`coarse`	MATISSE automatically chooses between the `sequential` and `global_rotation` coarse schedules, depending on the target device.
`fixed_work_groups`	MATISSE automatically chooses between the `sequential` and `global_rotation` fixed work group schedules, depending on the target device.
	Cooperative schedules
`cooperative`	Work-items in each work-group cooperatively perform computations.
`subgroup_ cooperative`	Work-items in each sub-group cooperatively perform computations.
	Fixed Work Groups
(Common)	The total number of work-groups is manually specified, and tasks are evenly distributed across the work-items. The exact number of work-items depends on the local size.
`fixed_work_groups_ sequential`	Each work-item processes a chunk of consecutive tasks.
`fixed_work_groups_ global_rotation`	Consecutive work-items process consecutive tasks, to promote memory coalescing.
	Coarse Schedules
(Common)	Each work-item processes a multiple tasks.
`coarse_ sequential`	Each work-item processes a chunk of consecutive tasks.
`coarse_ global_rotation`	Consecutive work-items process consecutive tasks, to promote memory coalescing.

Appendix D

SSA IR Pass Execution

Contents

This appendix lists the full set of SSA passes used by MATISSE and the order they are executed in.

D.1 List of Passes

This section lists the full set of SSA passes in Table D.1 Functions marked with (*) are only executed when MATISSE is in parallel code generation mode.

Table D.1: Description of available MATISSE execution schedules.

Name	Description
Can Be Executed In Any Stage (Before or After Type Inference)	
`ssa-validator`	Verifies that certain IR properties are respected.
`assume-in-range`	Adds the indices in range directive to all functions. Applied when MATISSE is in unchecked mode.
`assume-size-match`	Adds the matrix sizes match directive to all functions. Applied when MATISSE is in unchecked mode.
`redun-assign-elim`	Eliminates assignments (i.e., $A = B$), by replacing all references to A with references to B. In typed mode, the transformation is only applied if both variables have the same type.
`dead-code-elim`	Eliminates code that is dead or unreachable.
`empty-branch-elim`	Eliminates branches where both the then and else blocks are empty.
`block-reorder`	Ensures that blocks are in a canonical order and removes unreachable blocks.
`assume-builder`	Converts `%!assume_*` directives into SSA IR function properties.
`reorder-phi`	Ensure that `phi` instructions come before other instructions on any given block.
`ssa-printer`	Debug pass that prints the SSA IR code. Unused by default.
Can Only Be Executed After Type Inference	
`logical-access`	Converts `A(L)` matrix accesses into `A(find(A, L))`, when L is a logical value.
`scalar-val-bool`	Removes `val_boolean X` instructions when X is a scalar (as scalars are always valid to use as booleans in MATLAB).
`redun-output-elim`	In each typed function instantiation, rename unused return variables from `Name$ret` to a temporary variable name. Dead code elimination can then remove those variables entirely if they are unused in the body of the function.
`remove-type-str`	Removes the type parameter of calls to e.g., `zeros` since, after type inference, they are not necessary anymore.
`const-branch-elim`	Removes `if(true)`/`if(false)` branches, and replaces them with the body of the then/else cases, respectively.
`redun-cast-elim`	Eliminates redundant casts (i.e., casts to the same type).

Name	Description
`table-simplify`	Replaces simple `horzcat`/`vertcat` calls (implicitly used by matrix construction expressions `[...]`) by replacing them with the corresponding allocation and matrix sets.
`conv-matx-access`	On functions with the `assume indices in range` property, replaces accesses with scalar numeric indices with the `simple_*` equivalent.
`horzcat-elim`	Similar to `table-simplify`, but deals only with `horzcat` calls, and is capable of combining multiple rows (e.g., `[[1, 2], [3, 4]]`).
`conv-range-access`	Converts accesses with multiple indices into `range_get`/`range_set` instructions, for use in subsequent passes.
`conv-set-all`	Replaces `set $m2, $idx $scalar_value` instructions, where `$idx = 1:<numel of $m2>`, into `set_all` instructions, for use in subsequent passes.
`full-range-elim`	Replaces `range_*` instructions with the equivalent validation, allocation, and construction loop nest.
`set-all-elim`	Replaces `set_all` instructions with the equivalent loop nest.
`colon-elim`	Replaces `colon(a, b)` or `colon(a, 1, b)` calls with the equivalent allocations and loops.
`multi-get-elim`	Removes get instructions where the indices are matrices, replacing them by the equivalent allocations and loops.
`simple-m-set-elim`	Removes set instructions where exactly indices are matrices, replacing them with the equivalent loops.
`trivial-loop-elim`	Eliminates loops with 0 or 1 iterations.
`basic-access-simp`	Replaces `get`/`set` instructions with the `simple_*` equivalent, if valid. See Section 5.2.
`matrix-prealloc`	Preallocate matrices that are declared and dynamically grown inside loops. See Section 5.3.
`vert-flatten-elim`	Replaces `vertical_flatten` instructions with explicit allocations and loops.
`element-wise`	Replaces element-wise operations with the equivalent loop operations. See Subsection 5.1.1.
`redun-size-check`	Remove redundant `val_same_size` and `combine_size` instructions.
`val-size-simpl`	Simplifies redundant or partially redundant `val_same_size` instructions, by reducing the number of checks even in instructions that can not be fully eliminated.
`dot-reduct-elim`	Replaces calls to `dot` with the equivalent loop construct.
`cumul-reduct-elim`	Replaces calls to `sum` and `mean` with the equivalent loop construct.
`minmax3-rdct-elim`	Replaces calls to `min` and `max` that have 3 arguments with the equivalent allocations and loop nests.

Name	Description
`alloc-val-elim`	Replaces valued allocation functions such as `zeros` where the result is entirely overwritten before being read with calls that do not set the allocated data to any particular value.
`alloc-to-set-all`	Replaces valued allocation functions with calls to `set_all` instructions, for subsequent passes to optimize.
`get-index-simpl`	Removes redundant indices (of constant value 1) from `get`, `simple_get` and `relative_get` instructions. All `relative_get` instructions with a single non-redundant index are converted to `simple_get` instructions.
`get-or-first-simp`	Finds cases where the index of a `get_or_first` instruction is an iteration variable of a loop. If that loop is of the form `1:1:N` where $N \leq$ *size* of the matrix that was obtained, then the `get_or_first` can be converted into the `simple_get` equivalent.
`redun-trans-elim`	Eliminates calls to `transpose` and `ctranspose` that apply to 1D matrices, when the output of those functions is only used for single-index accesses.
`shape-propagation`	Identifies cases where a matrix is unnecessarily copied due to shape mismatch (generally caused by the `redun-trans-elim` pass), and fixes the shape.
`loop-acc-extract`	Detects matrix positions that are repeatedly modified in a loop, and rewrites the loop so that a scalar accumulator variable is used instead.
`triv-ac-size-elim`	Replaces `A = access_size B, I` instructions with a call to `A = size(B, I)` function, when B is a 1D matrix and I is a scalar.
`loop-start-normal`	Normalizes loop ranges into `1:X:Y` form.
`loop-icm`	Loop Invariant Code Motion. It was developed purely to assist the `loop-interchange` pass, so certain valid cases are disabled.
`loop-interchange`	Reorders loop nests to promote efficient memory accesses.
`loop-fusion`	Combines loop nests when possible. See Subsection 5.1.2.
`dup-read-elim`	Eliminates duplicated reads to the same matrix position.
`get-set-simplify`	Identifies `simple_get` instructions that access a matrix position that has been modified with `simple_set`, and replaces the get with an assignment to the set value.
`useless-mat-elim`	In cases where a matrix constructed inside a loop is never used after the loop, remove the `phi` nodes and reference the starting matrix instead, so that `dead-code-elim` can remove more uses of it.
`alloc-simplify`	In `X = zeros(A); Y = zeros(size(X));`, converts the second call into `Y = zeros(A);` (also works with other allocation functions), to make `dead-code-elim` more effective.

Name	Description
`alloc-size-simpl`	Removes calls to the allocation function (without data initialization) with a size given by a size matrix, with calls to another allocation function that copies the original matrix shape, to make `dead-code-elim` more effective.
`loop-mat-cp-elim`	When a matrix is allocated (with unitialized values) with the shape of another matrix (the source), and afterwards there is a `for` loop that sets all elements of the new matrix, replace references to the new matrix with references to the source matrix (i.e., reuse the source matrix instead of allocating a new one).
`access-size-elim`	Replaces `access_size` instructions with simpler operations.
`bounds-check-mot`	Identifies checked get instructions and converts them to `simple_get` instructions, by adding explicit bounds check instructions before the loop is executed.
`unnec-val-elim`	Eliminates `val_true $in` instructions where `$in` is known to be true.
`fxd-access-prop`	Replaces `$out = simple_get $x, <IDXS>`, where `$x` is the output of `zeros` or `ones`, and replaces it with an `$x = 0` or `$x = 1` instruction, respectively.
`redun-alloc-elim`	Removes allocations that are only used to compute size (e.g., `numel(zeros(4, 4))`, and replace the size function calls with the corresponding computation.
`loop-dep-elim`	In loops where a matrix is modified, but the original version of the SSA matrix is still referenced, replace references to the original SSA matrix with references to the in-loop SSA version, if doing so is valid.
`set-output-prop`	In `x = f(...); A(...) = x;`, sets the type of `x` to match `A`.
`symja-const-bld`	Uses Symja [Sym16] to discover the constant value of variables constructed with arithmetic expressions.
`end-elim`	Replace `end` instructions with simpler instructions.
`comb-size-elim`	Replace `combine_size` instructions with simpler instructions.
`loop-iter-simpl`	Reverts the changes of `loop-start-normal`.
`multi-set-const`	Converts chains of `simple_set` instructions into a combined `multi_set` instruction.
`opt-reporter`	Reports a missed optimization opportunity related to `sum` calls.
`no-dupl-exports`	Validate that there are no multiple function instances with the same explicit ABI name.

Name	Description
`ref-arg-dupl`	In functions with `%!by_ref` arguments, it is possible for the initial and the final variables to have colliding lifetimes. This pass solves this problem by duplicating reference arguments. If the lifetimes did not intercept, then the assignment will be eliminated by the final variable allocator. If they do, this fixes it by allowing the final variable allocator to safely assign `var$1` and `var$ret` to the same group.
`conv-to-cssa`	Converts the IR to CSSA form, as described by Boissinot et al. [BDR+09].
`par-block-ext` (*)	Extracts parallel block sections.
`invk-par-impl` (*)	Performs the parallelization itself, on parallel blocks.
`s-redc-buf-opt` (*)	Optimize cases where a buffer is copied to the CPU, then back to the GPU, by reusing the GPU buffer. See Section 5.7.
`const-buf-opt` (*)	Combine buffers that contain to the same data if they are not modified. See Section 5.7.
`undef-cp-elim` (*)	Eliminate copies of uninitialized data, by replacing `copy_to_gpu` instructions with the `allocate_on_gpu` equivalent. See Section 5.7.
`redun-cp-size` (*)	Eliminate copies that are only used to compute the size of a matrix. GPU code does not modify the shape of matrices, so using the outdated CPU matrix is correct. See Section 5.7.
`del-reduc-cp` (*)	Detects a pattern of matrix copies in a loop and replaces a `phi` instruction with a new copy. Counter-intuitively, adding this new copy actually reduces the total quantity of data transfers, as it allows `dead-code-elim` to eliminate the copies before the loop and at the end of each iteration. See Section 5.7.
`gpu-svm-elim` (*)	Converts SVM variables into explicit buffers, in certain cases. See Section 5.8.
`copy-ovwr-elim` (*)	Finds `copy_to_gpu` instructions followed by `set_gpu_range` instructions that cover the entire buffer and replaces the copy with a simple allocation. See Section 5.7.
`lp-mut-buf-cp` (*)	Finds matrices copied to the device within a loop, and back to the host at the end of each iteration, and rewrites the loop so that fewer copies are used. See Section 5.7.
`lp-ro-buf-cp` (*)	Extracts `copy_to_gpu` from loop bodies, for read-only buffers. See Section 5.7.
`svm-srg-elim` (*)	On certain devices, replaces `set_gpu_range` instructions with the equivalent OpenCL kernel call. See Section 5.8.

D.2 Default Phase Order

This section describes the default phase order used by MATISSE, in Table D.2.

Table D.2: Default Phase Order.

Pass Name	
Before Type Inference	
1. `ssa-validator`	
2. `assume-in-range` (depending on compiler setup options)	
3. `assume-size-match` (depending on compiler setup options)	
4. `redun-assign-elim`	5. `dead-code-elim`
6. `empty-branch-elim`	7. `block-reorder`
8. `assume-builder`	9. `ssa-validator`
After Type Inference	
10. `logical-access`	11. `scalar-val-bool`
12. `redun-output-elim`	13. `remove-type-str`
14. `const-branch-elim`	15. `redun-cast-elim`
16. `redun-assign-elim`	17. `table-simplify`
18. `dead-code-elim`	19. `conv-matx-access`
20. `horzcat-elim`	21. `conv-range-access`
22. `conv-set-all`	23. `dead-code-elim`
24. `full-range-elim`	25. `set-all-elim`
26. `colon-elim`	27. `multi-get-elim`
28. `simple-m-set-elim`	29. `conv-matx-access`
30. `trivial-loop-elim`	31. `basic-access-simp`
32. `matrix-prealloc`	33. `vert-flatten-elim`
34. `element-wise`	35. `redun-size-check`
36. `val-size-simpl`	37. `dot-reduct-elim`
38. `cumul-reduct-elim`	39. `minmax3-rdct-elim`
40. `dead-code-elim`	41. `alloc-val-elim`
42. `alloc-to-set-all`	43. `set-all-elim`
44. `ssa-validator`	45. `horzcat-elim`
46. `trivial-loop-elim`	47. `get-index-simpl`
48. `get-or-first-simp`	49. `redun-assign-elim`
50. `redun-trans-elim`	51. `shape-propagation`
52. `loop-acc-extract`	53. `ssa-validator`
54. `block-reorder`	55. `triv-ac-size-elim`
56. `loop-start-normal`	57. `loop-icm`
58. `loop-interchange`	59. `loop-fusion`
60. `ssa-validator`	61. `dup-read-elim`
62. `get-set-simplify`	63. `useless-mat-elim`
64. `alloc-simplify`	65. `alloc-size-simpl`
66. `loop-mat-cp-elim`	67. `access-size-elim`
68. `bounds-check-mot`	69. `unnec-val-elim`
70. `fxd-access-prop`	71. `dead-code-elim`
72. `redun-alloc-elim`	73. `dead-code-elim`
74. `loop-dep-elim`	75. `dead-code-elim`

Pass Name	
76. redun-size-check	77. redun-assign-elim
78. loop-icm	79. loop-interchange
80. set-output-prop	81. symja-const-bld
82. ssa-validator	
Additional Passes for Parallel Code Generation Mode	
At this point, MATISSE executes most of the *mandatory final passes*, but only when in parallel mode. The reason for this is to ensure that certain instructions are eliminated from the IR, so that the parallelization code does not need to consider them. It is likely that some of these passes are unnecessary and could be removed.	
83. ssa-validator	84. const-branch-elim
85. multi-get-elim	86. logical-access
87. vert-flatten-elim	88. full-range-elim
89. set-all-elim	90. end-elim
91. access-size-elim	92. comb-size-elim
93. dead-code-elim	94. multi-set-const
95. dead-code-elim	96. opt-reporter
97. reorder-phi	98. block-reorder
99. no-dupl-exports	
At this point, MATISSE is ready to parallelize the code	
100. par-block-ext	101. invk-par-impl
102. s-redc-buf-opt	103. const-buf-opt
104. undef-cp-elim	105. redun-cp-size
106. del-reduc-cp	107. redun-cp-size
108. dead-code-elim	109. gpu-svm-elim
110. copy-ovwr-elim	
SVM elimination may insert new data transfer instructions, so MATISSE re-applies several data transfer optimization passes.	
111. s-redc-buf-opt	112. const-buf-opt
113. undef-cp-elim	114. redun-cp-size
115. del-reduc-cp	116. redun-cp-size
117. lp-mut-buf-cp	118. lp-ro-buf-cp
119. svm-srg-elim	120. dead-code-elim
121. ssa-validator	
Mandatory Final SSA Passes	
122. ssa-validator	123. const-branch-elim
124. multi-get-elim	125. logical-access
126. vert-flatten-elim	127. full-range-elim
128. set-all-elim	129. end-elim
130. access-size-elim	131. comb-size-elim
132. loop-iter-simpl	133. dead-code-elim
134. multi-set-const	135. dead-code-elim
136. opt-reporter	137. reorder-phi
138. block-reorder	139. no-dupl-exports
140. ssa-validator	141. ref-arg-dupl
142. conv-to-cssa	