Extending the Capabilities of Tiramisu

by

Malek Ben Romdhane

B.S., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by . . . .
Saman P. Amarasinghe
Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Extending the Capabilities of Tiramisu

by

Malek Ben Romdhane

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

High performance computing requires not only writing highly efficient code, but also targeting multiple architectures (e.g. CPU, GPU, MPI). However, not only does bundling algorithm and optimization often obfuscate the code, but different architectures require different optimizations and programming tools. Tiramisu [3], an optimization framework, tries to solve this issue by separating algorithm, optimizations, and architecture details, and by targeting multiple architectures in a unified syntax.

In this work, we highlight the implementation of a Julia interpreter that compiles a subset of the language to Tiramisu. We show that by adding simple Tiramisu optimization commands to Julia code, we can achieve up to 14× speedup.

We also present an implementation of a CUDA backend for Tiramisu in order to target GPUs. We showcase a flexible Tiramisu CUDA API, as well as how common GPU usage patterns can be expressed in Tiramisu. We demonstrate that Tiramisu matches or outperforms the performance of the Halide GPU backend.

Thesis Supervisor: Saman P. Amarasinghe
Title: Professor

Acknowledgments

First and foremost, I would like to thank my supervisor, Saman Amarasinghe, for granting me the opportunity to work on amazing projects. I would also like to thank him for all the guidance he provided, and for making sure that the quality of my work is as good as it can be.

I would also like to thank Riyadh Baghdadi, who regardless of how busy he is, is always willing to lend a helping hand. I could not have finished my work, nor written my thesis, without his help and guidance.

I would like to thank the entire COMMIT group, a truly pleasant and helpful group who made my year in the lab very enjoyable.

Lastly, I would especially like to thank my family, whose support was essential in helping me through both my undergraduate studies and master's studies.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **11**

1.1 Background 11

1.2 Contributions 12

2 Tiramisu **15**

2.1 Tiramisu Overview 16

2.2 The Four-Layer IR 18

2.2.1 The Polyhedral Model 18

2.2.2 High Level Scheduling Commands 21

2.3 An Example Tiramisu Program 24

2.4 Advantages of Tiramisu 26

3 Julia to Tiramisu **27**

3.1 Overview 27

3.2 Related Work 28

3.3 Metaprogramming in Julia 28

3.3.1 Surface Syntax AST 30

3.3.2 Lowered Form 30

3.4 Design Choices 31

3.4.1 ParallelAccelerator 31

3.4.2 Scope of the Julia to Tiramisu Transpiler 31

3.5 The Transpilation Pipeline 32

3.5.1 The Pretag Pass 32

3.5.2 The Recovery Pass 32

3.5.3 The Dead Code Elimination Pass 33

3.5.4 The Code Generation Pass 33

3.6 Evaluation 36

4 Tiramisu to CUDA **39**

4.1 Related Work 39

4.2 Introduction to CUDA 40

4.2.1 The CUDA-enabled GPU Architecture 40

4.2.2 The CUDA API 42

4.3 Scheduling for GPUs in Tiramisu 43

4.3.1 CUDA Computations in Tiramisu 43

Tiramisu CUDA Example 45

4.3.2 Common Shared Memory Patterns in Tiramisu 47

Tiramisu Shared Memory 1D Filter Example 53

4.4 CUDA Generation Architecture 54

4.5 Evaluation 58

5 Conclusion **63**

A Shared Memory Pattern Performance Experiment **65**

List of Figures

2-1 Tiramisu overview 17

2-2 Tiramisu tiled box blur code 25

3-1 The Julia to Tiramisu pipeline 29

3-2 Execution time of benchmarks through the Julia and Tiramisu pipeline, normalized for Tiramisu 37

4-1 Tiramisu code representing code that performs addition and subtraction of two matrices on GPU 46

4-2 Thread utilization diagram for both possible GPU implementations of the blur filter 48

4-3 An example of Tiramisu introducing if guards when fusing for loops of different sizes 50

4-4 Simplified Tiramisu code representing the implementation of the 1D filter on GPU 53

4-5 ISL AST and resulting CUDA AST pseudocode showcasing the boundary condition extraction 56

4-6 End to end execution time of GPU benchmarks for Halide and Tiramisu, normalized for Tiramisu 59

4-7 Data copy execution time of GPU benchmarks for Halide and Tiramisu, normalized for Tiramisu 60

4-8 Kernel execution time of GPU benchmarks for Halide and Tiramisu, normalized for Tiramisu 60

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 Background

Over the last few years, available computational power has risen significantly, not only due to increasingly powerful CPUs, but also thanks to the proliferation of distributed computation clusters, and specialized hardware like GPUs and FPGAs. This has opened the door for many computationally expensive applications, that either have to deal with large amounts of data, or a very large search space. Examples of such applications include machine learning, and especially neural networks[22], scientific computation and simulation[14], and image processing[15].

Thus it is important, even necessary, to write high performance code that targets and makes use of all the available computation power. This means not only targeting peak performance possible by each harware piece within the system, but being able to use all computation hardware available in hetergenous systems (i.e. containing multiple types of processors). For example, this means being able to leverage the full power of a computing cluster by distributing the computation on all CPU and GPU cores in every node in the cluster.

There is an inherent difficulty with writing optimal code; mixing algorithm and optimizations obfuscates code. One reason for this is that the intuitive implementation of an algorithm might not be optimized for the hardware it is supposed to be run on. For example, when targetting CPUs, the intuitive implementation might not be cache optimal, resulting in significant slowdown. Another issue is that targetting multiple types of hardware often means using a different set of tools, and even a mixture of different programming languages, complicating the process of writing and maintaining the code, on top of the necessity to have communication and coordination between the different cores concurrently running the code in the heterogeneous system.

Tiramisu is an optimization framework that aims to solve these problems by offering a unified way to target multiple architectures that separates algorithm from optimization and architecture details. It can be used as an optimization framework and as a backend for compilers.

Using Tiramisu as a backend for compilers is not trivial. Tiramisu uses a producer consumer pattern to represent algorithms, and relies on the polyhedral model[7] to specify iteration spaces. This comes in contrast with the imperative pattern of many programming languages, which relies on loop nests and instructions to specify an algorithm. Thus, using Tiramisu as a backend for a compiler of an imperative style language requires a transformation between the patterns that conserves correctness and allows for optimizations. In chapter 3, we show how such transformation could be implemented. We chose Julia, an imperative style high performance high level language, with a focus on numerical analysis.

In order to achieve peak performance, Tiramisu needs to target multiple architectures. Adding support for a new architecture means not only having the ability to use the architecture, but also providing an API that is expressive and consistent with the rest of Tiramisu. In chapter 4, we detail the implementation of a CUDA target for Tiramisu, designed to satisfy these goals. This implementation allows Tiramisu users to achieve significant speedups for highly parallelizable algorithms.

1.2 Contributions

We implemented Julia to Tiramisu conversion by extracting program execution and data mapping information from the abstract syntax trees the Julia compiler exposes to the user. We relied in our implementation on a fork of ParallelAccelerator, a Julia optimization framework[1]. Additionally, we exposed TIRAMISU optimization primitives in Julia code using Julia macros. Through this implementation, we demonstrated the use of TIRAMISU as a backend optimization framework for a DSL compiler. We also demonstrated that optimization through TIRAMISU can achieve speedups of up to 14×.

Additionally, we implemented of a GPU backend for TIRAMISU specifically targeting CUDA, a general purpose programming framework for NVidia GPUs. We exposed a CUDA API for TIRAMISU that is coherent with the rest of TIRAMISU's API, all while being very expressive. We demonstrated the implementation of common GPU programs in TIRAMISU, including shared memory usage paradigms. We demonstrated that this GPU implementation generates efficient code with performance matching or outperforming Halide generated code, another popular optimization framework[15].

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Tiramisu

Tiramisu is an optimization framework designed to solve four big challenges:

- Targeting multiple architectures at the same time with the same syntax. This means being able to generate optimized programs that leverage the full power of the available hardware from CPU to GPU to FPGAs to distributed systems all at once. This not only means support for multiple architectures, but also tools to achieve correct communication and coordination between them.

- Removing data dependence. Since the memory representation of data is often intertwined with how the data is used in a program, any decisions to change the memory layout results in huge changes to the code. For example, this can happen if a programmer wishes to add support for a new architecture that requires a different layout, or if the programmer decides to change the size or number of dimensions of a buffer, or transform a large buffer into a smaller circular buffer.

- Generating optimized, efficient code. Nowadays optimization frameworks do a great deal of optimization to the code using complex automated techniques, relying on cost models, heuristics, and machine learning. These techniques, however good they are, are still limited because it is sometimes difficult for the frameworks to reason about the correctness of the optimization at a global level, which is required in order to find the most optimal mapping for all hardware used. Thus, it is sometimes only possible to generate extremely optimized code by hand, with the user being able to reason about the correctness of the optimizations.

- Keeping a reasonable legible representation of optimized code. Oftentimes optimizations affect the representation of the written code, which becomes riddled with architecture specific idioms, obfuscating the meaning of the original algorithm and making the implementation less portable. A better solution would be to allow the algorithm to be expressed in a simple unified way, all while having a representation flexible enough to allow the expression of optimizations seperately from algorithm specification.

In order to solve these problems, Tiramisu relies on a representation of the code based on the polyhedral model, and introduces a novel four-layer IR that separates algorithm from data mapping, optimization transformation, and architecture targeting.

2.1 Tiramisu Overview

Tiramisu is an optimization framework that uses a high-level architecture independent representation of an algorithm and a set of scheduling commands and data mappings, and generates optimized code accordingly. Tiramisu code can be written by hand, or targeted by a DSL compiler. The user can specify architecture independent scheduling commands and data mappings, as well as target architecture specific features like multi-core parallelism, vectorization, distributed computing, GPUs, and FPGAs in a unified syntax.

Tiramisu by design focuses on expressing parallel algorithms. In particular, it is designed to optimize algorithms that use dense arrays and that rely on loop nests. Such algorithms are very common in dense linear algebra and tensor algebra, stencil computations, image processing, and neural networks.

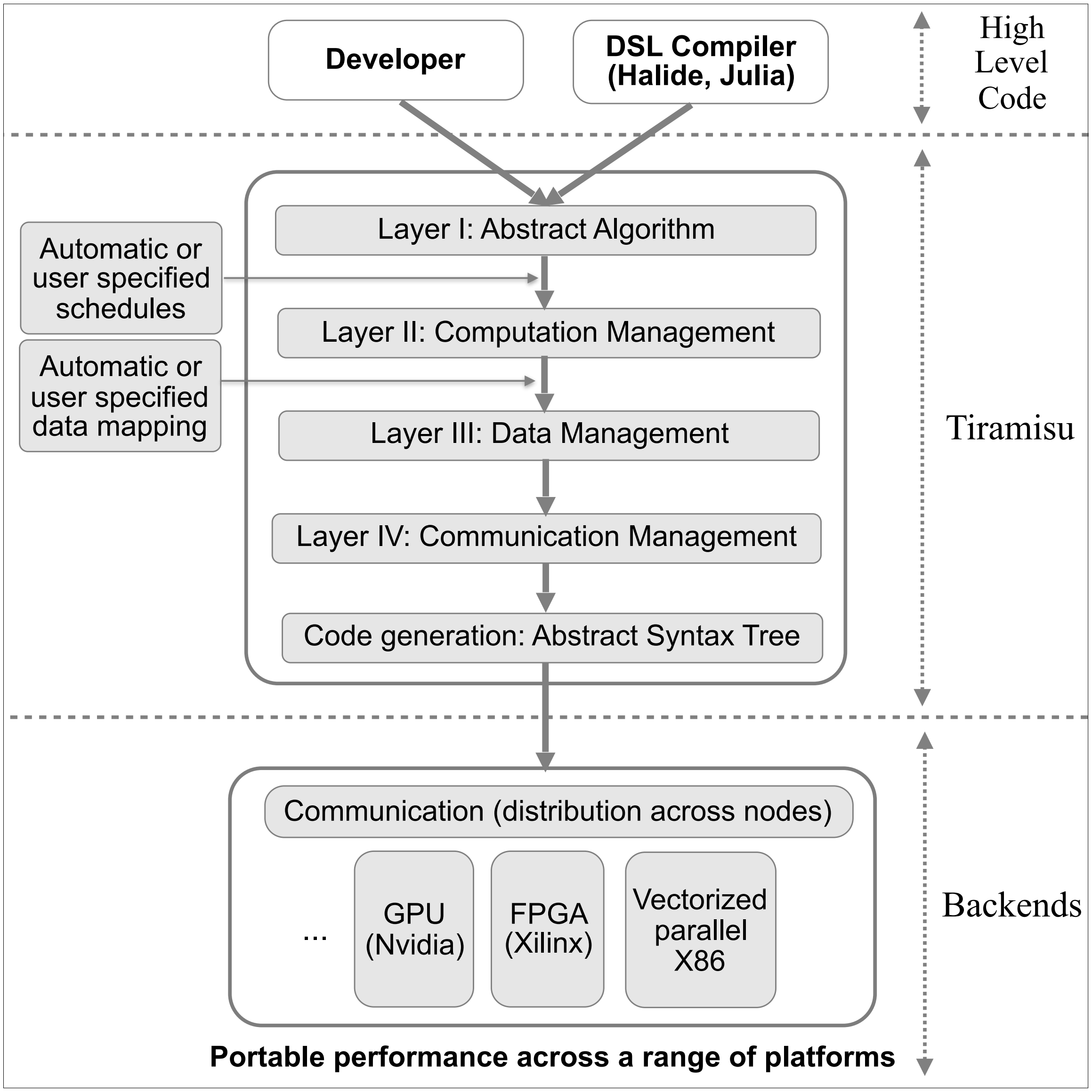
Developer
DSL Compiler (Halide, Julia)
High Level Code
Layer I: Abstract Algorithm
Automatic or user specified schedules
Layer II: Computation Management
Automatic or user specified data mapping
Layer III: Data Management
Layer IV: Communication Management
Code generation: Abstract Syntax Tree
Tiramisu
Communication (distribution across nodes)
...
GPU (Nvidia)
FPGA (Xilinx)
Vectorized parallel X86
Backends
Portable performance across a range of platforms

Figure 2-1: Tiramisu overview

2.2 The Four-Layer IR

In order to achieve the desired level of separation between algorithm, scheduling, and data mapping, Tiramisu programs are specified over 4 layers:

I. The *Abstract Algorithm* layer, this is where the algorithm is specified as a set of computations. Computation here roughly translates to a single instruction, as well as the iteration space this instruction exists in. This does not include any scheduling information (e.g. the order of execution of computations, loop fusion, loop tiling). Additionally, dependences are given as producer consumer relationships, avoiding memory-based dependences.

II. The *Computation Management* layer, this is where the computations are scheduled and optimized. This layer specifies the order of execution of the computations, any optimizations that could be applied to the computation's iteration space (e.g. tiling), as well as targetting the computation towards a specific architecture (e.g. GPU, distributed). This layer is still data layout oblivious, which simplifies a lot of the optimizations, and allows data layout changes to happen without having to rewrite all the optimizations.

III. The *Data Management* layer, this is where the data layout is specified. This includes buffers, mapping computations to memory, as well as the possibility to manually specify buffer allocations and frees.

IV. The *Communication Management* layer, this is where synchronization and communication operations are specified (within, and between different architectures). This is also where manual buffer allocations and frees are scheduled.

2.2.1 The Polyhedral Model

In order to achieve this level of separation, Tiramisu relies on the polyhedral model, a mathematical framework that can be used to represent loop iteration spaces, as well as apply many transformations, which can be used to optimize programs, and to specify data mappings. There are two important structures within the polyhedral model; **sets**, and **transformation maps**.

A set within the polyhedral model consists of integer tuples of the same dimensionality (called lattice points). It is described by specifying the dimensionality of the tuple, parametrizing each dimension within the tuple, and imposing conditions on those parameters using quasi-affine expressions. Quasi-affine expressions are expressions on integer literals and integer variables that include addition, subtraction, comparison operators, and conjunction [$\wedge$, and] and disjunction [$\vee$, or] operators. They also include multiplication as long as one of the operands is an integer literal, and division and the modulo operator as long as the second operand is an integer literal. Quasi-affine expressions can also be used to implement other common mathematical operators like the floor, ceiling, and absolute value functions. These sets are very valuable tools that can be used to represent iteration spaces for computations; for example the following polyhedral set

$$\{(i, j) : 0 \leq i < N \wedge i < j < N\}$$

can be used to represent the following loop nest

```
for (int i = 0; i < N; i++)
  for (int j = i + 1; j < N; j++)
    computation;
```

In Tiramisu, like this example, each dimension within the tuple is used to represent a loop level in a loop nest.

Transformations maps are mappings that use quasi-affine expressions to specify relations between polyhedral sets. They can be used to perform code transformations, to specify data mappings, and to order computations. An example of an optimization is loop skewing; consider the following nested for loops:

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < M; j++)
    a[i][j] = a[i - 1][j] + a[i][j - 1];
```

The inner for loop cannot be parallelized, as `a[i][j]` depends on `a[i][j - 1]`, which is produced in the same inner loop. However, by applying the following transformation map

$$\{(i, j) \to (i + j, j)\}$$

the code becomes

```
for (int i = 2; i < N + M - 2; i++)
  for (int j = max(i - N, 0) + 1; j < min(M, i) - max(i - N, 0); j++)
    a[i - j][j] = a[i - j - 1][j] + a[i - j][j - 1];
```

`a[i - j][j]` now depends on `a[i - j - 1][j]`, which is produced in the previous i iteration, and `a[i - j][j - 1]`, i.e. `a[(i - 1) - (j - 1)][j - 1]`, which is also produced in the previous i iteration. Thus the inner loop can now be parallelized. In fact, this example was generated automatically using Tiramisu, using the original computation, and a skewing directive.

Another use for transformation maps is data mappings. Say we want to compute the 200th Fibonacci number, using a computation whose iteration space is

$$\{(i) : 2 \leq i < 200\}$$

and whose expression is

$$\texttt{fib}(i) \leftarrow \texttt{fib}(i - 1) + \texttt{fib}(i - 2)$$

The easiest data mapping to ensure correctness is to create a 200 element buffer, initialize the first two values to 0 and 1, and stpre each $\texttt{fib}(i)$ as the i^{th} element in the buffer. Assuming the buffer's name is `f`, this is easily done using the following trivial map from computation to buffer (called access map)

$$\{\texttt{fib}(i) \to \texttt{f}[i]\}$$

However, if we only care about the 200th Fibonacci number, there is no need to store all 200 values; we only care about the last two. In Tiramisu, there is no need to modify the computation, it suffices to change the data mapping. We reduce our buffer to a circular buffer containing two elements, and we use the following access map to ensure correctness

$$\{\texttt{fib}(i) \to \texttt{f}[i \bmod 2]\}$$

What makes this concept powerful is that transformation maps naturally compose. This powerful mathematical framework is what allows for the four layer separation, and optimization composability that Tiramisu provides. Transformation maps are used throughout Tiramisu, including in the backend to order the execution of computations.

2.2.2 High Level Scheduling Commands

There are two types of computations, scheduled and unscheduled. A scheduled computation results in a C-style instruction in the code generated by Tiramisu (e.g. a buffer write). Unscheduled computations on the other hand do not result in an instruction. They rather function as wrappers, either around an expression (i.e. the computation is inlined), or around a buffer. Computations that are wrappers around buffers are just used for their access function, and not have an associated expression to compute. Other than inlining, layer II does not deal with unscheduled computations, and only deals with scheduled computations.

The most important command at layer II is the `after` command (and its derivative `before` and `between` commands). This specifies how computations are executed. For example say there are computations `a` and `b` specified by the iteration spaces

$$\{\texttt{a}(i, j) : 0 \le i < N \wedge 0 \le j < N\}$$

and

$$\{\texttt{b}(i, j) : 0 \le i < N \wedge 0 \le j < M\}$$

The command `b.after(a, root)` results in the following program structure

```
1 for (int i = 0; i < N; i++)
2    for (int j = 0; j < N; j++)
3       a(i, j);
4 for (int i = 0; i < N; i++)
5    for (int j = 0; j < M; j++)
6       b(i, j);
```

The command `b.after(a, i)` results in the following program structure

```
1 for (int i = 0; i < N; i++) {
2    for (int j = 0; j < N; j++)
3       a(i, j);
4    for (int j = 0; j < M; j++)
5       b(i, j);
6 }
```

It even takes care of differing loop bounds. That means that the command `b.after(a, j)` results in the following program structure

```
 1 for (int i = 0; i < N; i++) {
 2    for (int j = 0; j < min(N, M); j++) {
 3       a(i, j);
 4       b(i, j);
 5    }
 6    for (int j = min(N, M); j < N; j++)
 7       a(i, j);
 8    for (int j = min(N, M); j < M; j++)
 9       b(i, j);
10 }
```

If $N > M$, then the loop at line 5 will run, and if $M > N$, then the loop at line 7 will run.

The other layer II commands can be split into transformation commands, tagging commands, or a combination. Transformation commands change the iteration space using transformation maps. They include

- `interchange(i, j)`, which changes the order of the loop nests. An example is

$$\{(i, j) \to (j, i)\}$$

- `split(i, s, i0, i1)`, which tiles a loop in one dimension using an integer tile size s into two variables i_0 and i_1. An example is

$$\{(i) \to (i_0, i_1) : i = i_0 \times s + i_1 \wedge 0 \leq i_1 < s\}$$

- `tile(i, j, s1, s2, i0, j0, i1, j1)`, which tiles a loop in two adjacent dimensions using an integer tile size $s_1 \times s_2$ into variables i_0, j_0, i_1, and j_1. It's equivalent to applying `split(i, s0, i0, i1)`, `split(j, s1, j0, j1)`, and `interchange(i1, j0)`.

Tagging commands specify how a loop dimension should be executed. They include

- `tag_parallel(i)`, which says dimension i should be parallelized.
- `tag_distributed(i)`, which says dimension i should be distributed.
- `tag_gpu(i0, i1, i2, j0, j1, j2)`, `tag_gpu(i0, j0, i1, j1)`, and `tag_gpu(i0, i1)`, which says these dimensions should be executed on the GPU. These commands will be explained in more detail in the GPU chapter.

Some commands combine both transformation and tagging. They include

- `vectorize(i, s)`, which applies `split(i, s, i0, i1)` and tags the inner loop to be vectorized.
- `unroll(i, s)`, which applies `split(i, s, i0, i1)` and tags the inner loop to be unrolled.

In layer III, the user declares buffers and specifies data mappings. The `set_access` command is used to specify data mappings. In the Fibonacci number example specified in 2.2.1, the data mapping instruction would be

$$\texttt{fib.set\_access}\left(\{\texttt{fib}(i) \rightarrow \texttt{f}[i \bmod 2]\}\right)$$

In layer IV, the user specifies communication directives. These include data copies, synchronization, and architecture dependent buffer allocations. These directives are specified as computations, and are scheduled using layer II commands.

2.3 An Example Tiramisu Program

Let us consider the example of a box blur, applied first in the j dimension, and then in the i dimension. This means applying

$$\texttt{blur\_j}(i,j) = \frac{1}{3}\sum_{k=-1}^{1} \texttt{in}(i, j+k)$$

then

$$\texttt{blur}(i,j) = \frac{1}{3}\sum_{k=-1}^{1} \texttt{blur\_j}(i+k, j)$$

There is large data reuse within these computations, in the first computation among columns, and in the second computation among rows. This means that cache will influence performance. Assuming a row major buffer, the second computation will not be cache efficient if the image to be blurred is large. This is because the accesses will have large strides, and by the time the computation reached the end of a row, all data from the beginning of a row will be purged from the cache. One solution to that would be to tile this computation into smaller blocks that fit within the cache, to maximize reuse. The Tiramisu specification for this implementation is highlighted in figure 2-2, showing the split between the layers.

Layer I:

$\{\texttt{in}(i,j) : 0 \le i < N \wedge 0 \le j < M\}$, **unscheduled**
$\{\texttt{blur\_j}(i,j) : 0 \le i < N \wedge 1 \le j < M-1\}$:
    $(\texttt{in}(i,j-1) + \texttt{in}(i,j) + \texttt{in}(i,j+1))/3$
$\{\texttt{blur}(i,j) : 1 \le i < N-1 \wedge 1 \le j < M-1\}$:
    $(\texttt{blur\_j}(i-1,j) + \texttt{blur\_j}(i,j) + \texttt{blur\_j}(i+1,j))/3$

Layer II:

$\texttt{blur.tile}(i, j, 16, 16, i_0, j_0, i_1, j_1)$
$\texttt{blur.after}(\texttt{blur\_j}, \texttt{root})$

Layer III:

$\texttt{buffer\_in} : N \times M$, **input**
$\texttt{buffer\_blur\_j} : N \times M$, **temporary**
$\texttt{buffer\_blur} : N \times M$, **output**
$\texttt{in.set\_access}\,(\texttt{in}(i,j) \to \texttt{buffer\_in}[i,j]))$
$\texttt{blur\_j.set\_access}\,(\texttt{blur\_j}(i,j) \to \texttt{buffer\_blur\_j}[i,j])$
$\texttt{blur.set\_access}\,(\texttt{blur}(i,j) \to \texttt{buffer\_blur}[i,j])$

Original Tiramisu Box Blur Code

Listing 2.1: Pseudocode of Generated Code

```
allocate buffer_blur_j[N][M];
for (int i = 0; i < N; i++)
  for (int j = 1; j < M - 1; j++)
    buffer_blur_j[i][j] = (
      buffer_in[i][j-1] + buffer_in[i][j] + buffer_in[i][j+1])/3;
for (int i0 = 0; i0 < floor((N - 1)/16); i0++)
  for (int j0 = 0; j0 < floor((M - 1)/16); j0++)
    for (int i1 = 1 - min(i0 * 16, 1);
         i1 < min(N - 1 - i0 * 16, 16) + min(i0 * 16, 1);
         i1++)
      for (int j1 = 1 - min(j0 * 16, 1);
           j1 < min(M - 1 - j0 * 16, 16) + min(j0 * 16, 1);
           j1++)
        buffer_blur[i0 * 16 + i1][j0 * 16 + j1] = (
          buffer_blur_j[i0 * 16 + i1 - 1][j0 * 16 + j1] +
          buffer_blur_j[i0 * 16 + i1][j0 * 16 + j1] +
          buffer_blur_j[i0 * 16 + i1 + 1][j0 * 16 + j1])/3;
free(buffer_blur_j);
```

Figure 2-2: Tiramisu tiled box blur code

In this example, we create a temporary buffer to store the result of `blur_j`. This means it will automatically be allocated, and then freed.

2.4 Advantages of Tiramisu

One of the big advantages of Tiramisu is that it uses the polyhedral model. This makes the transformations that can be done to loops easily composable, simplifies the implementation of many optimizations, and allows for checks on the correctness of optimization. This makes Tiramisu a great choice for an optimization framework.

To elaborate, Tiramisu offers the following advantages:

- Optimizations expect a certain input format to work on. Traditionally, when composing optimizations, you need to rewrite the resulting optimization by hand, which can get very difficult if using multiple optimizations. By expressing loops as quasi-affine expressions, and optimizations as quasi-affine relations, it becomes easy to compose them, and the product of the composition does not need to be rewritten.

- By using the polyhedral model and quasi-affine transformations,Tiramisu allows the user to apply optimizations that are traditionally hard to apply in an interval based model of the iteration domain. Loop skewing is an example of such transformations.

- The ability to check the correctness of optimizations. For example, consider the case of loop fusion. Two loops cannot be fused if one loop produces results that are consumed by the other, and the fusion results in the consumption of a value before it is produced. The polyhedral model allows to easily check the validity of these transformations.

In the next chapter, we show how we can use these advantages in order to optimize the performance of Julia, a high level DSL with a focus of numerical analysis.

Chapter 3

Julia to Tiramisu

Julia is a dynamic general-purpose programming language designed with a focus on numerical computation and performance in mind. This makes Julia a great candidate to optimize using TIRAMISU. It also results in a few differences between Julia and more traditional numerical computation languages (e.g. MATLAB, python) that allows it to run faster and approach C performance[9]. Unlike standard implementations of other traditional numerical computation languages, Julia is not executed through an interpreter. It rather uses JIT compilation to create and run native code using LLVM. Additionally, even though it is a dynamically typed language, type information is more visible to the users than other dynamic languages. To generate well-optimized native code, all type information should be either deduced or explicitly annotated.

Both of these facts simplify the integration of TIRAMISU as an optimization backend; type information is necessary to generate TIRAMISU code, and the generation can be done by modifying the JIT compilation to run through TIRAMISU.

3.1 Overview

The Julia to TIRAMISU **transpiler** (source-to-source compiler) relies on Julia's metaprogramming features, which are highlighted in section 3.3. It is based off the ParallelAccelerator package, and is focused on numerical analysis applications. Section 3.4

explains these choices in more detail. It goes through a multi-stage pipeline that takes in a Julia function and returns a wrapper around an equivalent compiled Tiramisu function, as is explained in section 3.5. Additionally, figure 3-1 shows an overview of the pipeline. Using Tiramisu to optimize Julia code can result in significant speedups, as is shown in section 3.6.

3.2 Related Work

Polly is an LLVM optimization framework that relies on the polyhedral model[12]. Julia uses Polly for optimization when during the LLVM compilation stage. However, because it is at the LLVM level, a lot of information necessary to perform optimizations well is not available. For example, control flow structures are expressed using low level building blocks, which makes them difficult to recover and optimize. Additionally, the linearization of buffer accesses makes it difficult to recover memory access patterns. On top of that, Polly performs polyhedral optimization automatically, which might not always result in the best performance.

The Julia to Tiramisu transpiler is not the first example of use of a polyhedral framework for the optimization of a DSL. Tiramisu is already used to optimize Halide[15][3], a DSL for highly optimized image processing. PENCIL[2] is another polyhedral optimization framework designed to be targeted by DSLs, which was used to optimize both the VOBLA[8] linear algebra DSL, and the SpearDE[5] signal processing DSL. However, PENCIL uses Pluto[10], an automatic polyhedral optimization algorithm.

3.3 Metaprogramming in Julia

In order to transpile Julia code to Tiramisu, it's necessary to be able to manipulate it. The Julia language has a large emphasis on metaprogramming. In fact, Julia exposes different levels of the compilation pipeline to the user in order to enable highly flexible metaprogramming. Of these, two are used in the transformation of

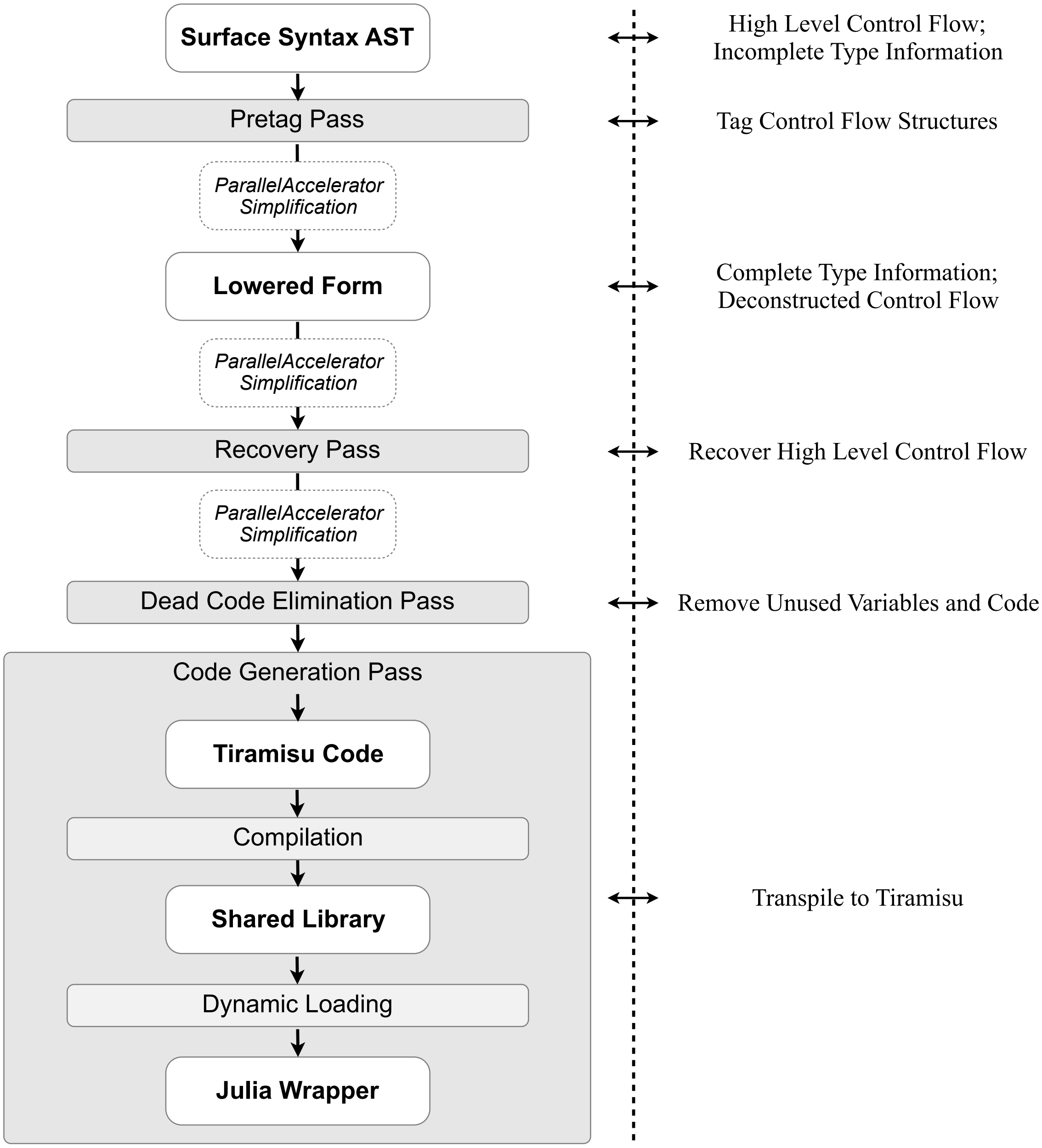
Surface Syntax AST
Pretag Pass
ParallelAccelerator Simplification
Lowered Form
ParallelAccelerator Simplification
Recovery Pass
ParallelAccelerator Simplification
Dead Code Elimination Pass
Code Generation Pass
Tiramisu Code
Compilation
Shared Library
Dynamic Loading
Julia Wrapper
High Level Control Flow; Incomplete Type Information
Tag Control Flow Structures
Complete Type Information; Deconstructed Control Flow
Recover High Level Control Flow
Remove Unused Variables and Code
Transpile to Tiramisu

Figure 3-1: The Julia to Tiramisu pipeline

Julia code to Tiramisu code; the surface syntax AST and the lowered form.

3.3.1 Surface Syntax AST

The surface syntax AST is produced by the Julia parser, and thus is a direct representation of the written code without any modifications. This is the main target for meta-programming in Julia, through the use of macros. Macros are Julia functions that are executed before compiling Julia code. They target a specific section of the Julia code. They take in a surface syntax AST, process it, and return a replacement surface syntax AST.

3.3.2 Lowered Form

The lowered form is an AST resulting from an intermediate step in the compilation process. It has fewer types of nodes than the surface syntax AST. In the lowered form, all the macros are expanded. Additionally, all control flow structures are converted into branch instructions (i.e. conditional and unconditional gotos), thus flattening all such structures. A completely type annotated AST (including deduced types) only exists in lowered form, thus making it necessary to use in order to generate the Tiramisu equivalent.

Relevant types of nodes in this AST include:

- `Expr`: This is the most generic node type. It could be a function call, a value assignment, a conditional branch, or a return statement, a meta node (a compiler comment node), among other things.

- `GotoNode`: This is an unconditional branch instruction.

- `LabelNode`: This is a branch target.

- `Slot`: This represents a local variable or argument in the code.

- `SSAValue`: This represents a static single assignment variable that is inserted by the compiler.

Using this reduced form has a big drawback. Since all control structures are lost and replaced with branches, it is difficult to recover the loop structure. Thus, to generate TIRAMISU, it is necessary to use both forms of the AST.

3.4 Design Choices

3.4.1 ParallelAccelerator

Because the lowered form of the code is destined for code generation through LLVM, there is a lot of superfluous information that makes it hard to decode and turn into TIRAMISU code. One example is linkage information about function calls (including arithmetic operations on primitive types, buffer accesses) that makes it hard to determine what function is being called.

ParallelAccelerator[1] is a Julia package developed by Intel that also aims to accelerate Julia code. It analyzes both types of the Julia AST, tries to detect opportunities to accelerate the code (parallelization, vectorization, etc...), generates optimized C code, compiles it, and loads it back into Julia to replace the original function. It performs simplifications of the superfluous information in the AST. Additionally, it is designed to optimize only specified function in the code, thus enabling users to mix optimized code with unoptimized/unoptimizable code. For that reason, and in the spirit of code reuse, the Julia to Tiramisu transpiler is based on a fork of ParallelAccelerator.

3.4.2 Scope of the Julia to Tiramisu Transpiler

TIRAMISU, by virtue of its focus on loop nests is a great tool for tensor algebra, which strongly aligns with Julia's focus on numerical analysis. More specifically, the subset covered by the Julia to Tiramisu transpiler focuses mainly on for loops, simple arithmetic operations (addition, subtraction, multiplication, division), and buffer and scalar reads and writes. This is not limiting because the transpiler can target the specific part of the code that needs to be optimized, and the rest of the code can go through the regular Julia pipeline.

Additionally, it extends Julia by exposing macros that target Tiramisu scheduling commands that are usually useful in numerical analysis, like the after command for fusing loops, and the loop parallelization command.

3.5 The Transpilation Pipeline

ParallelAccelerator (and the Tiramisu transpiler) works through passes. Each pass has a specific role, and a pass type which specified the type of AST that it takes; **macro** for the surface syntax AST, and **typed** for the typed lowered form. This sections lists the passes implemented by the Tiramisu transpiler.

3.5.1 The Pretag Pass

The pretag pass is a **macro** pass. Because the lowered form replaces all of the control flow structures with branches and labels, it can be difficult to recover the original control flow structure. For that reason, a pretag macro pass is added to label all control flow structures, and add the data as *meta* nodes in the AST. Since these nodes are not generated by Julia, they do not get modified or destroyed as the AST is lowered, except to keep it in sync with the rest of the AST; for example, if a certain variable is referenced in a for loop, and that variable is transformed to a `Slot` in the lowered AST, references to that variable in the annotation introduced by the pretag pass are updated to reference the `Slot` node in the lowered form.

After the pretag pass, the AST goes through other ParallelAccelerator passes for AST simplification.

3.5.2 The Recovery Pass

The recovery pass is a **typed** pass. The point of the recovery pass is to recover the original structure of the code. In other words, it transforms the AST from branches and labels back to higher level control flow structures like for loops, all while keeping the type information provided in the lowered form.

The recovery pass relies mainly on the tags created in the pretag pass. However, control flow structures also generate a lot of instructions in the lowered form that represent their low level executions. For example, in for loops, this includes initialization, loop bound tests, and branches. Thus a simple pattern recognition algorithm is used to remove all of the superfluous low level instructions. Since each control flow structure can introduce multiple instructions in several chunks, it is the algorithmic equivalent to matching several substrings in a specific order.

Afterwards, the AST goes through the other ParallelAccelerator typed passes that simplify code even further.

3.5.3 The Dead Code Elimination Pass

The dead code elimination pass is a **typed** pass. It has the simple role of removing unused variables and all expression relating to unused variables. Even though this can be done at the Tiramisu level, doing it at this level speeds up the compilation time. This is because there is no need to generate Tiramisu code that handles this code, and because the cost of generation, compilation, and optimization is superior to removing unused variables at this level.

3.5.4 The Code Generation Pass

The code generation pass is a **typed** pass, and is the last pass. It has the role of generating Tiramisu code, compiling it, and loading it into Julia.

Initially, each node in the AST is parsed exactly once. This is similar to how a C compiler would parse the code. This is possible because there is no forward dependecies in the AST. This pass functions recursively (i.e. it uses the same procedure to analyze nodes and their children etc. . .). It checks the type of the node, and performs different actions based on the type:

- If the node is an `Expr` node, it analyzes its expression.

- If it is a function call, the transpiler first checks if function call is supported. This includes function calls that are generated in the recovery pass (control flow delimiters such as `:for_loop_start` and `:for_loop_end`). In that case, for loops get their variables parsed, and these variables are added to a loop variable stack that keeps track of the current location of the code in the loop structure, alongside with the loop variable conditions. Currently supported function calls also include addition, subtraction, multiplication, division, as well as buffer reads and writes, which are all simplified by pre-existing ParallelAccelerator code. The arguments of the function call are then recursively parsed. If the result is the equivalent of a Tiramisu computation, it creates a computation object, based on the expression in the code, and on the deduced location of the instruction in the loop structure.

- If it is an assignment, the pass analyzes the left hand side of the assignment. If it is a scalar, then the node is a scalar assignment. In that case, a computation object is created, based on the right hand side expression and the loop structure context. If it is a buffer, then it must be a buffer allocation, as buffer write accesses are done through function calls. In that case, a buffer object is created based on the size specified in the right hand side allocation expression.

- If it is a meta node, then it is a scheduling command generated in the pretag pass. Tiramisu scheduling commands are passed as meta nodes (e.g. `:begin_parallel`, `:end_parallel`, `:begin_fuse`, `:end_fuse`, with a corresponding fusion and parallelization level). All the necessary information about scheduling (from the meta tags, and the order in which instructions are written) is stored in a temporary ScheduleState object that keeps track of all the necessary scheduling information and generates the necessary scheduling commands when the Tiramisu code is eventually generated.

  - If it is a return node, then the buffer referred to in the node is tagged to be returned.

- If it's a variable use, then it must be a scalar access either of a loop variable, or a user or compiler defined scalar. The pass asserts that the variable has already been defined, and produces an object representing the variable. It creates different objects for loop variables and other scalars. The reason for this separation is that Tiramisu treats them differently.

- If it's a primitive value, the pass simply parses that value.

The pass then generates the TIRAMISU code based on the analysis. Because producer/consumer relations are not clear in an imperative style language, it relies on unscheduled computations to ease the translation. Unscheduled computations are computations that do not have an expression, and are not generated in the actual code, but have a defined buffer access relation. They simply function as buffer read access wrappers.

The pass initially generates unscheduled computations that represent the size of the input buffers, as TIRAMISU expects that information to either be hard coded or passed in explicitly, similarly to how C code would be written. It then creates unscheduled computations that represent buffer accesses. Afterwards, it generates the actual scheduled computations, and the buffer objects. It then creates all the buffer mappings from the generated computations (both scheduled and unscheduled) to the created buffers. Finally, it generates all the scheduling commands.

In order to load the object into Julia, the pass compiles the generated tiramisu code into an object file, makes a shared library file out of it, and dynamically links it. It then replaces the original function with a wrapper that:

- Adds a size buffer for every buffer function argument. Each size buffer contains the dimensions of the corresponding argument buffer.

- Allocates the output buffer if it is originally allocated inside the function, as TIRAMISU needs the output buffer to be passed in.

- Calls the loaded function that has been generated by Tiramisu.
- Returns the output buffer if it is returned in the original function.

3.6 Evaluation

Julia v0.5 was used to perform the evaluation. In order to evaluate each benchmark, the same function code was used in both the regular Julia pipeline and the Tiramisu pipeline, and using available optimization tags. The loop fusion tag was only used for the Tiramisu pipeline, as no equivalent exists for Julia.

In order to accurately compare the execution time, the body of the function was timed for the Julia pipeline, and the execution of the external Tiramisu function was timed for the Tiramisu pipeline. Additionally, a macro was applied to the Julia pipeline function in order to remove out of bound access checks.

The benchmarks used are

- **bicg**, which stands for biconjugate gradient. It attempts to solve a system of linear equations of the form $Ax = b$, where A is a matrix, and x and b are vectors.
- **doitgen**, which performs the Multiresolution analysis kernel (MADNESS). It performs multiple scalar products and assignments.
- **mttkrp**[19], which stands for Matricized tensor times Khatri-Rao product. It consists of a multiplication of an order-3 tensor and two matrices, resulting in a matrix.
- **covariance**, which performs the covariance computation of a square matrix.
- **gesummv**, which performs a scalar, vector and matrix multiplication.

Every benchmark other than mttkrp is included in the PolyBench benchmark suite[17], and follows the same implementation.

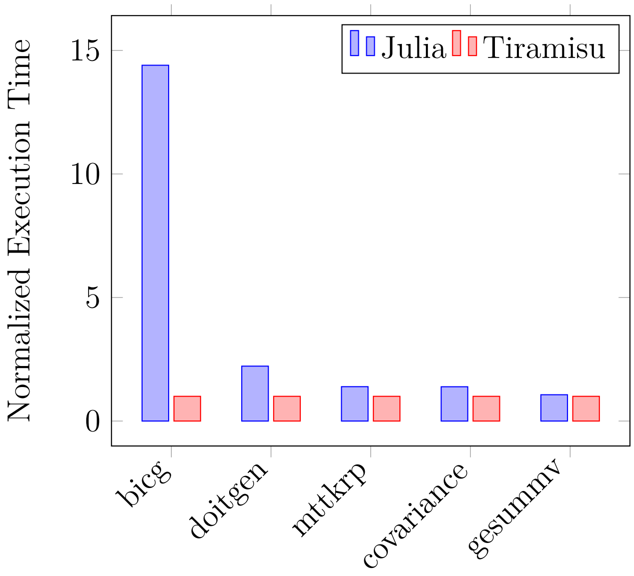

Figure 3-2: Execution time of benchmarks through the Julia and Tiramisu pipeline, normalized for Tiramisu

As is evident from figure 3-2, the execution time of the Tiramisu pipeline is either comparable to or faster than the execution time of the Julia pipeline, reaching up to a 14× speedup for bicg. The faster execution time is mainly due to the loop fusion tag and loop interchanges, which only exist in Tiramisu.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Tiramisu to CUDA

In addition to implementing the Julia to Tiramisu transpiler, work on Tiramisu involved expanding the capabilities of the framework itself, by implementing a CUDA[16] backend. CUDA is an API developed by NVidia to allow for general purpose computing on GPUs (Graphics Processing Units). This chapter goes over a general introduction to CUDA, the CUDA interface exposed to Tiramisu users, the implementation details of the Tiramisu CUDA backend, and the backend's evaluation results.

4.1 Related Work

There are multiple optimization framework that target GPUs. The PENCIL[2] polyhedral optimization framework is designed to target hardware accelerator like GPUs, and is built upon an OpenCL backend. Other examples of polyhedral optimization framework that target GPUs include GPU extensions to the Polyhedral Compiler Collection (PoCC)[4], as well as an automatic polyhedral C-to-CUDA GPU optimization framework[6].

There also exists non-polyhedral high level approaches to GPU-based optimization. Halide[15] is a DSL for image processing that exposes multiple scheduling commands ans targets GPUs using both the CUDA API[16] and OpenCL[21]. NOVA[11] and Lift[20] are functional data-parallel languages that can produce high performance GPU code generation. However, they do not expose scheduling commands, and perform scheduling automatically.

4.2 Introduction to CUDA

4.2.1 The CUDA-enabled GPU Architecture

A GPU is a hardware accelerator originally designed for applications with graphical focus, such as user interfaces, image and video manipulation, and 3D applications. This usually means manipulating a large number of points in a graphical context, and such computations are highly parallelizable. As such, GPUs are designed to be able to process many computations at the same time. For example, an NVidia Tesla V100 can run up to 5120 threads of execution in parallel.

These threads of execution however do not each run independently. They are executed within the context of warps, which are thread groupings. At a given moment, all threads within a warp are executing the same instruction (on potentially different data). In case some threads within a warp are not supposed to be executing an instruction, then they remain idle until other threads in that warp finish executing that instruction. This is called thread divergence. An example of thread divergence is highlighted in listing 4.1.

Listing 4.1: Pseudocode representing thread divergence

```
1 x = rand(); // generates a number between 0 and 1
2 if (x < 0.5)
3   heads = heads + 1;
4 else
5   tails = tails + 1;
```

Assume threads within the same warp get values of `x` on different sides of 0.5. In that case, the threads executing line 3 will perform their computation, and the other threads will have to idly wait. Afterwards, threads executing line 5 will perform their computation, with the other threads waiting.

In CUDA-enabled GPUs, threads do not use the same memory that CPU cores on the same computer use. They rather use GPU specific memory. Since the CPU is referred to as the host and the GPU is referred to as the device, their memories are referred to respectively as host and device memory. In fact, there are many types of device memory spaces:

- **Global memory** is the largest memory space available. It is visible to all threads within a device. It is also visible to CPUs, if they wish to copy data between main memory and device memory. It is the slowest of all memories to access, but if multiple threads access consecutive global memory locations at the same time, their accesses are coalesced, resulting in higher bandwidth.

- **Shared memory** is a memory space that is allocated per block, which is a grouping of threads specified by the user. It is usually significantly faster than global memory. However, it is only accessible by the device, and more specifically only within the block in which it is declared. Thus, threads are responsible for populating shared memory. It is useful when there is a high level of data reuse across a block of threads, in order to avoid going through the slower global memory. For example, if a certain section of global memory is used by a block of threads, each thread in the block could load a part of that section into shared memory, and all the threads can access the data from the faster shared memory. Such use of shared memory is very common, and can result in significant speedups.

- **Constant memory** is a memory space that is provided by the host before the computation is performed on the device. As the name suggests, it cannot be modified by the device. It is smaller in size than global and shared memory. However, it is faster than both global and shared memory if many threads access the same location at the same time. An example use case would be performing a convolution using a relatively small convolution kernel.

- **Registers** are the fastest memory space available. They are only accessible to one thread of execution at a time, and function similarly to CPU registers. However, their total number is significantly higher that in CPUs. For example, on the NVidia V100, the total memory available for GPUs is 20480KB, for up to 4KB per thread, with all threads running. When running a computation, the device allocates a portion of the registers to each thread, as instructed by the running program. They are usually used to store thread-local variables.

4.2.2 The CUDA API

The CUDA API is what allows users to interact with GPUs to perform general purpose computations. It consists of tools to program the device, and tools for the host to interact with the device, such as launching computations, and copying data between the host and device. It provides a low level instruction set for CUDA enabled GPUs, called PTX, and compilers from C, C++, and Fortran style CUDA code that compiles to PTX.

In order for these programs to be run on CUDA devices, they need to be provided in kernels. Kernels are specialized functions that incorporate in their design the parallelized nature of GPU computation. When declaring a kernel, code is simply provided for a single thread of execution, and relies on a user defined structure to identify threads. Threads are grouped into blocks. Threads, and blocks, can be organized into a 1, 2, or 3 dimensional structure. This is because some problems lend themselves towards higher dimensions (e.g. matrix addition is a two dimensional problem). Each thread has a set of unique identifiers for each dimension within its block, and each block has another set of unique identifiers within the set of all blocks, which is called a grid. These identifier sets allow a thread to determine its role in the computation. Thus if a kernel is one dimensional, then each thread has a thread x id within a block, and a block x id within the grid. If it's two dimensional, it also has a thread y id and a block y id. If it's three dimensional, it also has a thread z id and a block z id.

In order to launch a kernel, the user needs to pass all the arguments the kernel needs, as well as the number of threads per each block dimension, and the number of blocks along side each dimension of the grid. Users need to be careful when choosing the size of a block, as a single warp can only execute threads within the same block simultaneously. Thus, if the block size is not a multiple of the warp size, some threads within the warp will be idle, resulting in under utilization of the GPU. Users cannot also include all of the threads in a single block, as the number of threads per block is limited.

Within a kernel, the CUDA API offers multiple instructions in addition to regular programming primitives. These include declaring shared memory buffers, which results in one buffer allocation per block. That buffer is then accessible by all threads within that block. If a block has more threads than the warp size, it needs more than one warp to execute its threads. These warps can go out of sync and do not necessarily execute the same instruction simultaneously. This can result in errors if different threads access the same memory location, which is the main use case for shared memory. For that reason, the CUDA API also offers a synchronization barrier instruction, which is only passed when all threads in a single block reach it. Additionally, because the CUDA compiler can introduce optimizations that can modify the memory access order, synchronization might be necessary even within warps, as it also acts as memory barrier that insures that all writes are reflected in memory. Omitting it might result in race conditions.

Outside of kernel, the CUDA API enables the user to allocate global and constant memory, transfer data between device and host, and control a host of other parameters.

4.3 Scheduling for GPUs in Tiramisu

4.3.1 CUDA Computations in Tiramisu

There is a direct correlation between threads and blocks on one hand, and tiled, parallelized for loops on the other. Consider for example a single for loop with a unit step. Tiling it means creating two nested for loops, with the outer one iterating over each tile, and the inner one going through the iterations within the tile. This makes it easy to map to GPU, with a 1 dimensional grid, blocks the size of the tile, and the same number of blocks as there are tiles. Figure 4-5 highlights such an example.

The same thing happens when tiling two loops into rectangular tiles, which results in 4 loops, with the outer two iterating over the tiles (blocks), and the inner two going through the iterations within each tile (threads).

Thus, to map a Tiramisu computation to GPU, it suffices to tag 2, 4, or 6 consecutive dimensions in a computation as GPU dimensions. The outer most tagged loop gets replaced with a kernel call. The spans of the first half of the tagged loops are used to determine the number of blocks, and the spans of the second half are used to determine the size of the blocks. Any loops surrounding the tagged loops are executed on the host, and any loops contained within the tagged loops are executed as for loops on the device. If there are multiple computations within a loop nest, once the first computation is tagged to be executed on GPU, all of the computations will be executed within the same kernel. This means that entire loops, not single computations, are transformed to kernels.

When trying to tile a loop, its loop span might not be a multiple of its tile size. Thus, it is necessary to introduce a check at the inner loop level to make sure that no excess iterations are performed at the last iteration of the outer loop. Tiramisu introduces these checks automatically, and if the tiled loop is mapped to GPU, these checks are generated as if guards within the kernel.

The user does not need to specify any arguments that need to be passed to the kernel, as Tiramisu detects that information on its own based on variable and buffer access patterns.

In order to create GPU buffers, it suffices to tag buffer object as global, shared, constant, or register. Constant buffers are automatically allocated and do not need to be freed. Global buffers are automatically allocated and freed, but can also be managed manually. Shared memory is allocated manually, as it needs to be declared within the kernel. It also does not need to be freed. Registers are special buffers containing only one element (effectively a scalar). They are also allocated manually since they need to be declared within the kernel. They do not need to be freed.

Tiramisu also exposes expressions that allow the user to perform block synchronization, as well as copy data from host to constant memory, or between host and global memory.

More specifically, Tiramisu uses the following expressions and scheduling commands to map algorithms to GPU:

- `computation.tag__gpu`(i_0, i_1), `computation.tag__gpu`(i_0, j_0, i_1, j_1), and `computation.tag__gpu`$(i_0, j_0, k_0, i_1, j_1, k_1)$. This layer II scheduling command maps the specified computation to be executed on GPU, mapping the given 2, 4, or 6 consecutive dimensions to block identifiers and thread identifiers.

- `synchronize`(). A layer IV computation with this expression results in a synchronization barrier at the level where the computation is scheduled.

- `memcpy`(`buffer__src`, `buffer__dest`). A layer IV computation with this expression performs a memory copy between device and host from `buffer_src` to `buffer_dest` at the level where the computation is scheduled.

Tiramisu CUDA Example

Figure 4-1 shows an example of a Tiramisu GPU computation. This simple example takes two matrices, A, and B, copies them to device, computes their sum and difference, and copies the result back to host.

Layer I:

$\{\texttt{A}(i,j) : 0 \le i < N \wedge 0 \le j < M\}$, **unscheduled**
$\{\texttt{B}(i,j) : 0 \le i < N \wedge 0 \le j < M\}$, **unscheduled**
$\{\texttt{sum}(i,j) : 0 \le i < N \wedge 0 \le j < M\} : \texttt{A}(i,j) + \texttt{B}(i,j)$
$\{\texttt{diff}(i,j) : 0 \le i < N \wedge 0 \le j < M\} : \texttt{A}(i,j) - \texttt{B}(i,j)$

Layer II:

$\texttt{sum.tile}(i, j, 32, 32, i_0, j_0, i_1, j_1)$
$\texttt{diff.tile}(i, j, 32, 32, i_0, j_0, i_1, j_1)$
$\texttt{sum.tag\_gpu}(i_0, j_0, i_1, j_1)$
$\texttt{diff.after}(\texttt{sum}, j_1)$

Layer III:

`buffer_A_host` : $N \times M$, **input, host**
`buffer_A_gpu` : $N \times M$, **temporary, device global**
`buffer_B_host` : $N \times M$, **input, host**
`buffer_B_gpu` : $N \times M$, **temporary, device global**
`buffer_sum_host` : $N \times M$, **output, host**
`buffer_sum_gpu` : $N \times M$, **temporary, device global**
`buffer_diff_host` : $N \times M$, **output, host**
`buffer_diff_gpu` : $N \times M$, **temporary, device global**
$\texttt{A.set\_access}\,(\texttt{A}(i,j) \rightarrow \texttt{buffer\_A\_gpu}[i,j])$
$\texttt{B.set\_access}\,(\texttt{B}(i,j) \rightarrow \texttt{buffer\_A\_gpu}[i,j])$
$\texttt{sum.set\_access}\,(\texttt{sum}(i,j) \rightarrow \texttt{buffer\_A\_gpu}[i,j])$
$\texttt{diff.set\_access}\,(\texttt{diff}(i,j) \rightarrow \texttt{buffer\_A\_gpu}[i,j])$

Layer IV:

$\{\texttt{copy\_A(0)}\} : \texttt{memcpy}(\texttt{buffer\_A\_host}, \texttt{buffer\_A\_gpu})$
$\{\texttt{copy\_B(0)}\} : \texttt{memcpy}(\texttt{buffer\_B\_host}, \texttt{buffer\_B\_gpu})$
$\{\texttt{copy\_sum(0)}\} : \texttt{memcpy}(\texttt{buffer\_sum\_gpu}, \texttt{buffer\_sum\_host})$
$\{\texttt{copy\_diff(0)}\} : \texttt{memcpy}(\texttt{buffer\_diff\_gpu}, \texttt{buffer\_diff\_host})$
$\texttt{copy\_B.after}(\texttt{copy\_A}, \texttt{root})$
$\texttt{sum.after}(\texttt{copy\_B}, \texttt{root})$
$\texttt{copy\_sum.after}(\texttt{diff}, \texttt{root})$
$\texttt{copy\_diff.after}(\texttt{copy\_sum}, \texttt{root})$

Figure 4-1: Tiramisu code representing code that performs addition and subtraction of two matrices on GPU

4.3.2 Common Shared Memory Patterns in Tiramisu

When running an algorithm on GPU that produces a buffer, it is common to have each thread in the kernel produce a specific value within the buffer in order to maximize parallelism. In case the kernel requires data reuse, it is also common to have the same threads load the needed data into shared memory before performing the computation. That way, data is only accessed once from global memory within a block, but can be used by any thread of that block using the faster shared memory.

However, in many cases, the size of the output can be smaller than the size of the input. Consider a simple filter that blurs an image in the j-direction using the following expression

$$\texttt{out}(i, j) = \frac{1}{3}\sum_{k=0}^{2} \texttt{in}(i, j + k)$$

When trying to map this computation to GPU, there is a mismatch between the number of threads within a block needed to perform the loading to shared memory and the number of threads needed to perform the computation. There are two solutions to this problem, as highlighted in figure 4-2:

1. Have some threads perform an additional load to shared memory, during which the remaining threads in the block do not perform any computation, as they wait for the loading to finish. If the blocks chosen are of size $A \times B$, this means a thread utilization of $\frac{2}{B}$ during the second loading period, and a 100% utilization otherwise.

2. Have all the threads perform a single shared memory load, and only the threads that have all the needed information in shared memory perform the actual computation. If the blocks chosen are of size $A \times B$, this means a thread utilization of $\frac{B-2}{B}$ during the computation, and a 100% utilization otherwise.

We thus have two factors at play, utilization, and number of memory accesses. The first solution has worse utilization, and less global memory accesses, and the second solution has better utilization, but more global memory accesses. The effect of having more global memory accesses is mitigated by the fact that these memory accesses are

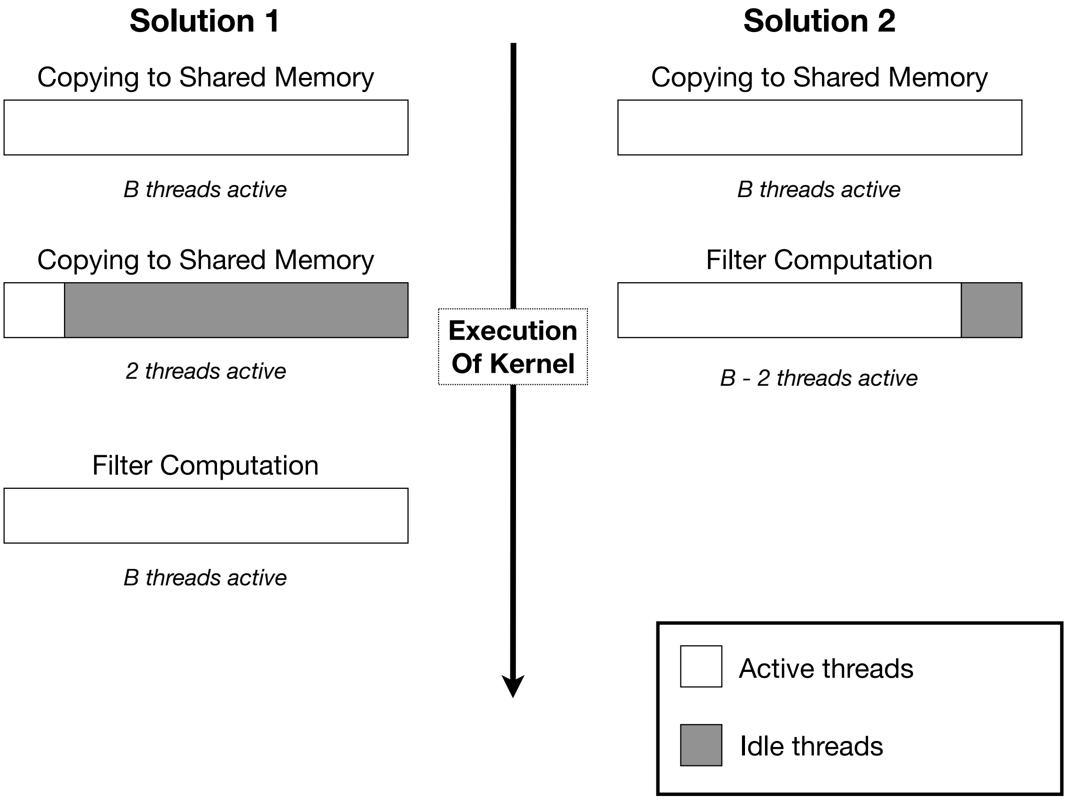

Figure 4-2: Thread utilization diagram for both possible GPU implementations of the blur filter

coalesced. When compared experimentally using the NVidia profiler `nvprof` on an NVidia K80 card using a matrix of size 6000×7500 and blocks of size 32×32, the first solution averages 5.1150ms over 60 trials, and the second solution averages 4.9134ms over 60 trials. 32×32 is a very common choice for block size. Thus both of these solutions are close in performance, with (in this case) the second solution being faster. The code for the expriment is listed in appendix A.

This section will detail how to implement the second solution in Tiramisu, as it is the tricky one to implement. There are four computations necessary to the implementation:

1. Computation `dec` for declaring/allocating the shared buffer.

2. Computation `copy` for copying the input from global memory to shared memory.

3. Computation `sync` for adding a synchronization barrier to assure that shared memory is not used by any thread until populated.

4. Computation `out` for performing the computation.

In order to explain this implementation, it is easier to analyze the one dimensional case as extending it to two dimensions is trivial, i.e.

$$\texttt{out}(j) = \frac{1}{3}\sum_{k=0}^{2}\texttt{in}(j+k) \tag{4.1}$$

The key idea behind this implementation is that if Tiramisu is trying to fuse two loops of different sizes, it will add the necessary if guards to insure correctness. Figure 4-3 highlights an example of this feature. Thus, what we should strive for in each part of the implementation is to have each part have an outer loop (corresponding to the block x id) iterating over the same interval of size corresponding to the number of blocks, and the inner loop (corresponding to the thread x id) iterating of different intervals, with the largest of which being as large as the block size.

Let the size of the output of the filter be N, and the block size be B. Assuming computation `out` if created to represent equation 4.1, it needs to be tiled in order to

Layer I:
$\{\texttt{a}(i,j) : 0 \leq i < 16 \wedge 0 \leq j < 16 \wedge 16i + j < 250\}$
$\{\texttt{b}(i,j) : 0 \leq i < 16 \wedge 0 \leq j < 16\}$
$\{\texttt{c}(i,j) : 0 \leq i < 16 \wedge 0 \leq j < 11 \wedge 11i + j < 170\}$

Layer II:
$\texttt{b.after}(\texttt{a}, j)$
$\texttt{c.after}(\texttt{b}, j)$

Tiramisu code

Listing 4.2: Generated Structure

```
for (int i = 0; i < 16; i++) {
   for (int j = 0; j < 16; j++) {
      if (i * 16 + j < 250)
         a(i, j);
      b(i, j);
      if (j < 11 && i * 11 + j < 170)
         c(i, j);
   }
}
```

Figure 4-3: An example of Tiramisu introducing if guards when fusing for loops of different sizes

map it to GPU. Each block should compute $B-2$ output values, thus the computation is tiled using a tile size of $B-2$. This results in $\lceil \frac{N}{B-2} \rceil$ outer loop iterations, i.e. blocks. The buffer mapping is trivial in this case, as buffer mappings in Tiramisu are oblivious to tiling, so it suffices to map the original computation.

The computation `copy` that copies from global memory to shared memory needs to have the same number of blocks as the `out` computation, and have a block size (i.e. a tile size) that is B. More precisely, for each block of the `out` computation, there needs to be two extra inner loop iterations (threads). Thus, the total number of iterations of the `copy` computation before tiling is

$$N + 2 \times (\text{number of blocks of } \texttt{output})$$

which when plugging in the formula for the number of blocks becomes

$$N + 2 \times \left\lceil \frac{N}{B-2} \right\rceil$$

Then, when this computation is tiled with a tile size of B, we end up with the same number of blocks as `out`, but with two extra threads per block, which is the desired effect.

However, the buffer mapping relies on the original index of the computation before tiling. And since we introduce two extra iterations per block, this means that some of these iterations on the edge of the block will be loading the same value from the input, and we wish to correct for that. Thus when trying to read the input, we need to discard any iteration that would be repeated in order to find the exact value we are trying to find from the index. This is done by subtracting two from the index for each block, i.e. when given an index i, the formula for the corrected index is

$$i - 2 \times (\text{number of previous blocks})$$

which when plugging in the block size B becomes

$$i - 2 \times \left\lfloor \frac{i}{B} \right\rfloor$$

Figuring out where the information should be stored shared memory is a simpler expression; it suffices to find where the thread is located within the block, i.e. find the thread index within the block. Since the block size is B, the index to store the information in shared memory is $i \bmod B$.

Thus, to summarize how to implement the `copy` computation:

- It needs to iterate over an interval of size $N + 2 \times \left\lceil \frac{N}{B-2} \right\rceil$.
- It needs to be tiled with a tile size of B.
- Iteration i needs to load data from $\texttt{in}\left(i - 2 \times \left\lfloor \frac{i}{B} \right\rfloor\right)$.
- Iteration i needs to store data in the shared buffer `sdata` at $\texttt{sdata}\left[i \bmod B\right]$.

If a computation needs to load more than 2 extra values, it suffices to replace every occurence of 2 in the formulae above with the new number of values.

The last computations to implement are the synchronization computation and the shared buffer declaration. The synchronization computation needs to be executed in every block by every thread in the block, or otherwise the execution of the kernel might hang and result in a deadlock. Thus we need to make sure that the span of the original iteration space of the `sync` computation before tiling is a multiple of B. Thus it needs to have $\left\lceil \frac{N}{B-2} \right\rceil \times B$ iterations. Synchronization computations do not have buffer accesses. The shared buffer declaration computation `dec` is implemented in the exact same way.

A big advantage of this technique is that it is expressed in the original loop indices, instead of the indices after tiling. Having a smaller number of indices significantly simplifies the code necessary to express a program. Another advantage of this technique is that TIRAMISU can leverage the composability of the polyhedral model to vastly simplify the generated code. All of these complicated and computationally expensive expressions would vanish to be expressed in terms of simpler expression that rely on the block and thread indices.

Additionally, this technique can be implemented in any dimension, and even composed across dimensions, as it allows the user to pick the execution model of any computation across any dimension:

- Executed in every thread of every block like `dec` and `sync`.

- Executed in every thread of every block, except maybe the last block (global boundary condition) like `copy`.

- Executed by a fixed group of thread of every block, except maybe the last block which might have even fewer threads running (global boundary condition) like `out`.

Tiramisu Shared Memory 1D Filter Example

Figure 4-4 highlights how the 1 dimensional filter example described above would be implemented in TIRAMISU. For brevity and clarity, global and host buffer declaration, and their related access functions were omitted. Data copies were also omitted. Figure 4-1 contains usage examples of the omitted detail.

Layer I:
$\{\texttt{in}(i,k) : 0 \le i < N \wedge 0 \le k < 3\}$, **unscheduled**
$\{\texttt{out}(i) : 0 \le i < N\} : (\texttt{in}(i,0) + \texttt{in}(i,1) + \texttt{in}(i,2))/3$

Layer II:
$\texttt{out.tile}(i, B-2, i_0, i_1)$

Layer III:
$\texttt{sdata} : B$, **temporary, shared**
$\texttt{in.set\_access}\,(\texttt{in}(i,k) \rightarrow \texttt{sdata}\,[i \bmod (B-2) + k])$

Layer IV:
$\{\texttt{in\_global}(i) : 0 \le i < B\}$, **unscheduled**
$\left\{\texttt{dec}(i) : 0 \le i < \left\lceil \frac{N}{B-2} \right\rceil \times B\right\} : \texttt{declare}(\texttt{buffer\_sdata})$
$\left\{\texttt{copy}(i) : 0 \le i < N + 2 \times \left\lceil \frac{N}{B-2} \right\rceil\right\} : \texttt{in\_global}\left(i - 2 \times \left\lfloor \frac{i}{B} \right\rfloor\right)$
$\left\{\texttt{sync}(i) : 0 \le i < \left\lceil \frac{N}{B-2} \right\rceil \times B\right\} : \texttt{synchronize}()$
$\texttt{copy.set\_access}\,(\texttt{copy}(i) \rightarrow \texttt{sdata}\,[i \bmod B])$
$\texttt{dec.tile}(i, B, i_0, i_1)$
$\texttt{copy.tile}(i, B, i_0, i_1)$
$\texttt{sync.tile}(i, B, i_0, i_1)$
$\texttt{dec.tag\_gpu}(i_0, i_1)$
$\texttt{copy.after}(\texttt{dec}, i_1)$
$\texttt{sync.after}(\texttt{copy}, i_1)$
$\texttt{out.after}(\texttt{sync}, i_1)$

Figure 4-4: Simplified TIRAMISU code representing the implementation of the 1D filter on GPU
Irrelevant detail has been omitted.

4.4 CUDA Generation Architecture

There are two steps involved in the compilation of CUDA target code in Tiramisu; the generation of an AST (hereafter called the *CUDA AST*) that represents the kernel, and the compilation of the AST and integration of the kernel call into the rest of the CPU code generated by Tiramisu.

Tiramisu relies on the ISL library[23] to compute, among other things, the control flow structure of the final program. For that, ISL provides an AST that consists mainly of for loops, if/else conditions, function calls (which in Tiramisu represent where a specific computation's instruction is to be included), along side all the necessary expressions to be able to specify the AST (e.g. references to loop variables, quasi-affine arithmetic operations). It does not contain all of the information necessary to produce the code, as the final expressions of each computation as well as certain loop tags, including the GPU tags, are stored elsewhere. Listing 4.2 is an example of an ISL AST.

In order to transform the ISL AST into CUDA AST, a generator object is created. The generator object has a mutable state, and is in charge of visiting the ISL AST nodes (as well as the expressions of every computation) and generating the CUDA AST and information about how to integrate the CPU code with the GPU code. It keeps track of declared scalars and buffers, current loop variables, whether or not the generator is in a kernel or not, and if so, what loop variables map to what block/thread indices, among other things.

It is initially fed the ISL AST of the whole Tiramisu code, for which the transformation is rather straightforward, except in the case of a for loop, or a computation instruction, or a scalar access. In the case of a for loop, it constructs the for loop with an empty body, and adds the iterator to the set of declared scalars as well as the loop iterator stack along side the minimum and the maximum bounds of the iterator. The bounds are important in case the iterator turns out to be a GPU thread/block index, as the bound information is necessary to specify the block/grid size.

The generator then analyzes and transforms the body of the loop. This will reveal if the loop represents a kernel's thread or block index, because this information is only known at the computation instruction level within the loop's body. The generator checks if the loop iterator has been tagged as a GPU index while analyzing the body. If it has, and there are more iterators that have been tagged as GPU iterators, then the for loop is replaced with its body in the CUDA AST. If it is the last GPU iterator, then a kernel object is created that contains the body of the for loop, and the for loop is replaced by a kernel call in the CUDA AST. If the for loop's iterator is not tagged as GPU, then that for loop is simply placed in the CUDA AST as is.

The ISL AST also contains computation instructions. A computation instruction represents where a computation will be placed in the generated call and is what gets transformed into a buffer write, a function call, or some other instruction like an allocation or a synchronization barrier. Additionally, computations contain all the GPU tagging information. Thus, the first thing that the generator does when encountering a computation instruction is to check if it is GPU tagged. If it is, then the generator changes all the tagged iterators to accesses to GPU block or thread indices, and stores that information so that any future mention of the iterators is transformed to GPU index accesses. It also extracts any boundary conditions created by ISL, including those generated by the tiling, and replaces them with if guards. An example of the boundary condition extraction is shown in figure 4-5.

The generator then creates the actual instructions associated with computation by

1. generating all the constants associated with the computation, and necessary to perform the computation.

2. generating the actual instruction. If the computation does not generate a buffer or scalar write, then the instruction represented by the computation (e.g. buffer declaration/allocation, buffer free, memory copy, or synchronization barrier instruction) is simply created as is. Otherwise, it generates the write from the expression and information about the structure provided by ISL. Using ISL is important to insure correctness in case a loop variable is modified through a

Listing 4.3: Provided ISL AST

```
for (int i = 0; i < floor(N / 16); i++)
  for (int j = 0; j < min(16, N - 16 * i); j++)
    a(i, j)
```

Listing 4.4: Generated CUDA AST structure

```
// Generated kernel
// __global__ here means it's a kernel
__global__ void kernel()
{
   if (threadIdx.x < N - 16 * blockIdx.x)
      a(blockIdx.x, threadIdx.x);
}
// Replacement call
// kernel<<<a, b>>> says there are a blocks with b threads each
kernel<<<N/16, 16>>>();
```

Figure 4-5: ISL AST and resulting CUDA AST pseudocode showcasing the boundary condition extraction
The original loop has a span of N, and is tiled with a tile size of 16 and mapped to GPU.

transformation. For example, if a loop variable is tiled into two loop variables, then any mention of it should be replaced by its expression in terms of the resulting variables. This is also the part where accesses are used to translate a computation to buffer/scalar write, and to replace any computation access with a buffer access.

3. wrapping the generated assignment in a conditional block using the user defined predicate, if one is provided. The user defined predicate is an additional condition that the user can provide to restrict when to execute the computation, other than the schedule provided to ISL.

Additionally, any buffer or scalar created in a computation within a kernel is tagged as kernel local. This means that it shouldn't be passed into the kernel from the host. A non local scalar access could represent a thread or block index, in which case the access is transformed into a GPU index access. Any other buffer or scalar that is accessed within a kernel must have been defined outside the kernel. It is thus kept track of, to be passed in through the generated kernel call. Buffer accesses only happen within the scope of a computation. Scalar accesses on the other hand can occur anywhere in the ISL AST, so these checks are also implemented when a scalar is created (i.e. an iterator) or accessed in the ISL AST.

As a result of all of these steps, a CUDA AST is created that represents all of the code, including the CPU. However, since the CPU code is implemented using a different IR (the Halide IR), a new AST is created that contains:

1. constant memory buffer declarations, and methods to be able to access these constant buffers.

2. kernel definitions for all generated kernels.

3. host wrapper functions that contain the kernel calls, and that specify the grid and block dimensions.

This CUDA AST is then used to generate CUDA C code and write it to a file on disk in preparation for compilation. The CUDA C generation relies on a visitor pattern. Every type of node within the AST has one or more print methods associated with them. In the general case, executing a print method returns the textual representation of a node in CUDA C. There are some other types of specialized print methods. The body printing method for example takes care of adding C braces and semi-colons for correctness, as well as beautifying the code by adding indentation to make the generated code easier to debug. The declaration print method on buffers and scalars specifies that the buffer or scalar is being printed in the context of a declaration, in order to add declaration specific information, like the data type or memory location of the created object. This visitor pattern is implemented using dynamic dispatch (i.e. virtual methods).

On the host side, the Halide IR generator is able to create instructions that perform the allocation and freeing of global memory, as well as copying data between the host and the device (for global and constant memory). These instructions call hand-written precompiled wrappers around the CUDA Runtime API. Additionally, whenever the Halide IR generator encounters a for loop that was found to specify a kernel call by the CUDA AST generator, the for loop is not analyzed, but rather replaced with a call to the kernel call wrapper.

Finally, at the compilation phase, three object files are created in case the Tiramisu code targets the CUDA API. One is the usual CPU code generated by the Halide backend from the Halide IR. The other two are compiled by NVCC from the generated CUDA C file mentioned above. One contains the CPU wrapper code, and the other one contains the GPU kernel code and constant memory buffer symbols. When using the generated Tiramisu function, it suffices to pass in these object files.

4.5 Evaluation

In order to evaluate the performance of the GPU backend, multiple benchmarks were implemented in both Tiramisu and Halide, which is another optimization framework. The performance comparison includes an end to end evaluation (i.e. from data on host to result on host), as well as a comparison of the data copy time and the kernel execution time. The benchmarks were run on a p3.2xlarge AWS instance, which contains an NVidia Tesla V100 GPU.

Since Halide automates a lot of its decision, Halide benchmarks were written in order to trigger the usage of shared memory when useful for a fair comparison. This was verified by looking at the generated code. Halide however does not currently support constant memory.

The benchmarks used are

- **cvtColor**, which converts an RGB image to grayscale.

- **convolution**, which applies a 5×5 convolution filter on an image. This tests usage of both shared and constant memory.

- **warpAffine**, which warps an image according to an affine map.

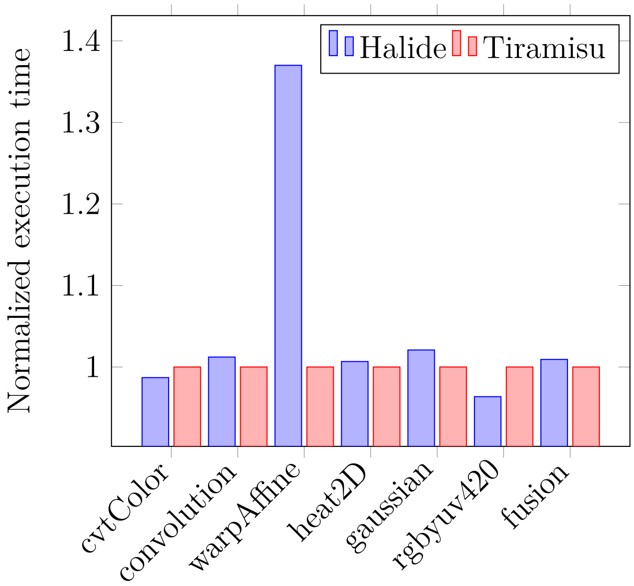

Figure 4-6: End to end execution time of GPU benchmarks for Halide and TIRAMISU, normalized for TIRAMISU

- **heat2D**, which computes one iteration of Heat2D. Heat2D models heat transfer on a (2-dimensional) matrix. This tests usage of shared memory.

- **convolution**, which applies a guassian filter in the x, then in the y direction. This tests usage of both shared and constant memory.

- **rgbyuv420**, which converts an image from the RGB color space, to the YUV color space.

- **fusion**, which tests loop fusion (and thus the ability to perform two separate computation within the same kernel).

The benchmarks used a 32 × 32 block size. All benchmarks used a 2112 × 3520 pixel image as an input, except heat2D, which uses a 10000 × 10000 matrix.

A look at figure 4-6 shows that end-to-end performance is comparable for both Halide and TIRAMISU, except for warpAffine. Figure 4-7 highlights that is due to data copy time. This is because the input buffer only uses one channel, and in Halide, the whole buffer is copied based on buffer metadata, whereas in TIRAMISU, buffers are

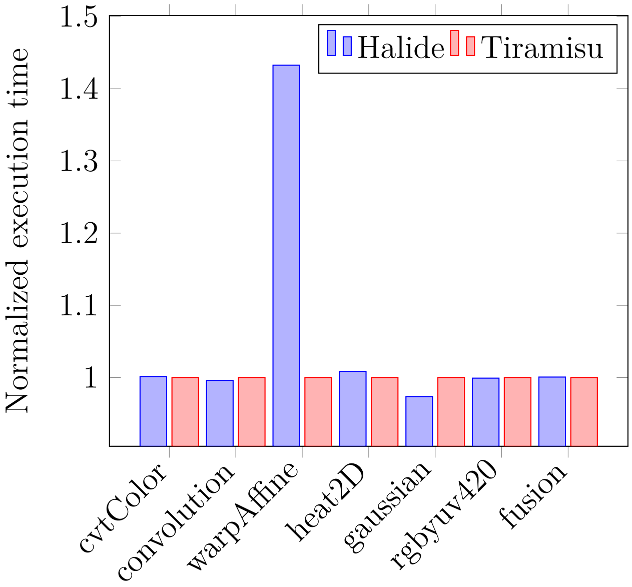

Figure 4-7: Data copy execution time of GPU benchmarks for Halide and Tiramisu, normalized for Tiramisu

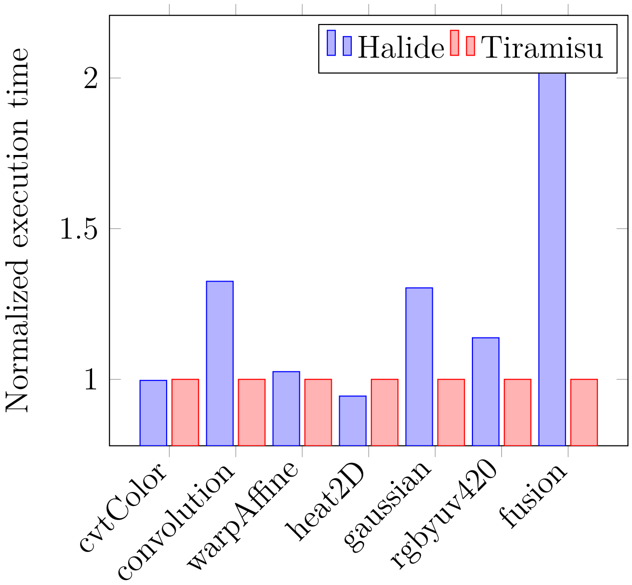

Figure 4-8: Kernel execution time of GPU benchmarks for Halide and Tiramisu, normalized for Tiramisu

copied based on the size the user specifies. This figure also shows that data copies dominate end to end performance, as is expected.

Figure 4-8 shows that kernel performance is mostly comparable, except in the case of convolution, gaussian, and fusion. Convolution and gaussian run faster Tiramisu because of the use of constant memory, which is not yet supported in Halide. Fusion runs faster in Tiramisu because it merges multiple separate computations into one kernel. Additionally, those computations feature data reuse, which is optimized so that there is a unique access to the data from global memory.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Conclusion

We have implemented a Julia to Tiramisu transpiler, a programming language with a focus on numerical analysis, in order to optimize its performance. This showcased not only the viability of Tiramisu as an optimization backend for DSL compilers, but also that using it leads to clear performance increases. We also implemented a CUDA backend for Tiramisu in order to expand the range of architectures it can target, which is important to achieve peak computational performance in a heterogenous computation system. We showcased that it achieves comparable performance to the Halide platform, a popular optimization platform developed by Google.

Moving forward, a rewrite of the Julia to Tiramisu transpiler using the Cassette[18] package would increase its robustness and maintainability. Since the transpiler relies on pattern matching of a low level IR, any major change in the way the Julia transpiler lowers its AST would break the transpiler. The Cassette.jl package, currently in development, aims to provide a consistent robust API to influence the compilation process of Julia, which would be a natural target for a Julia to Tiramisu transpiler.

As for the CUDA backend, improvements can be made both on the level of support of CUDA features, as well as the Tiramisu CUDA API. Examples of CUDA features that could be supported include texture memory, asynchronous execution of memory copies and kernels using CUDA events, as well as more fine-grained synchronization techniques (e.g. warp level memory barriers). Additionally, extensions to Tiramisu would ease the implementation of many GPU kernels. Parametric tiling for example, i.e. tiling by a tile size unknown at compile time, would allow the user to query device information dynamically in order to pick the most optimal tile size. An easier way to express for loops for which the iterator decreases logarithmically would ease the implementation of reductions. Additionally, TIRAMISU can provide abstractions for common GPU usage patterns to make its API easier to use. An example of such usage pattern is the shared memory pattern discussed above.

Appendix A

Shared Memory Pattern Performance Experiment

Listing A.1: CUDA code for the shared memory experiment

```
#include <iostream>

template <int blockSize>
__global__ void case_1(
    float * input, float * output, int N, int M)
{
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;
    const int by = blockIdx.y;
    const int tidx = tx + bx * blockSize;
    const int tidy = ty + by * blockSize;
    const bool predicate = tidy < N && tidx < M;
    __shared__ float sdata[blockSize][blockSize + 2];
    if (predicate)
    {
        sdata[ty][tx] = input[tidy * (M + 2) + tidx];
        if (tx < 2)
        {
            // Necessary if the data is not aligned
            // with the block size
            int added = min(M - blockSize * bx, blockSize);
            sdata[ty][tx + added] =
                input[tidy * (M + 2) + tidx + added];
        }
    }
    __syncthreads();
    if (predicate)
    {
        output[tidy * M + tidx] = sdata[ty][tx]
```

```
            + sdata[ty][tx + 1] + sdata[ty][tx + 2];
    }
}

template <int blockSize>
__global__ void case_2(
    float * input, float * output, int N, int M)
{
    const int tx = threadIdx.x;
    const int ty = threadIdx.y;
    const int bx = blockIdx.x;
    const int by = blockIdx.y;
    const int blockSizeRestricted = (blockSize - 2);
    const int tidx = tx + bx * blockSizeRestricted;
    const int tidy = ty + by * blockSize;
    __shared__ float sdata[blockSize][blockSize];
    if (tidy < N && tidx < M + 2)
    {
        sdata[ty][tx] = input[tidy * (M + 2) + tidx];
    }
    __syncthreads();
    if (tx < blockSizeRestricted && tidy < N && tidx < M)
    {
        output[tidy * M + tidx] = sdata[ty][tx]
            + sdata[ty][tx + 1] + sdata[ty][tx + 2];
    }
}

void correct(float * input, float * output, int N, int M)
{
    for (int i = 0; i < N; i ++)
        for (int j = 0; j < M; j ++)
        {
            float sum = 0;
            for (int k = 0; k < 3; k++)
            {
                sum += input[i * (M + 2) + j + k];
            }
            output[i * M + j] = sum;
        }
}

float * generate(int N, int M)
{
    float * result = new float[N * M];
    for (int i = 0; i < N; i ++)
        for (int j = 0; j < M; j ++)
        {
            result[i * M + j] = (i + j - i * j) / 20.;
        }
    return result;
}

bool compare(float * ref, float * out, int N, int M)
{
    for (int i = 0; i < N; i ++)
        for (int j = 0; j < M; j ++)
        {
            if (out[i * M + j] != ref[i * M + j])
            {
                std::cout <<
                        "Error at (" << i << ", " << j << "): "
                        << out[i * M + j] << " != "
                        << ref[i * M + j] << std::endl;
                return false;
            }
        }
    return true;
}

void gpuAssert(cudaError_t code, int line)
{
  if (code != cudaSuccess)
  {
    std::cerr <<
        "GPU error: " << cudaGetErrorString(code) <<
        " at line " << line << std::endl;
    exit(code);
  }
  else
  {
    std::cerr << "GPU command at line " << line <<
                         " successful." << std::endl;
  }
}

#define verifyErrors() gpuAssert(cudaPeekAtLastError(), \
                __LINE__)

#define iceil(a, b) ((a) + (b) - 1) / (b)

int main()
{
    int N = 6000, M = 7500;
    float * input = generate(N, M + 2);
    float * output = new float[N * M];
    float * output1 = new float[N * M];
    float * output2 = new float[N * M];
    correct(input, output, N, M);
    float * inputgpu, * output1gpu, * output2gpu ;
    cudaMalloc(&inputgpu, N * (M + 2) * sizeof(float));
    cudaMalloc(&output1gpu, N * M * sizeof(float));
    cudaMalloc(&output2gpu, N * M * sizeof(float));
    cudaMemcpy(inputgpu, input, N * (M + 2) * sizeof(float),
               cudaMemcpyHostToDevice);

    const int blockSize = 32;

    for (int i = 0; i < 60; i++)
    {
        dim3 grid(iceil(M, blockSize), iceil(N, blockSize));
        dim3 blocks(blockSize, blockSize);
        case_1<blockSize><<<grid, blocks>>>(
               inputgpu, output1gpu, N, M);
    }
    verifyErrors();

    for (int i = 0; i < 60; i++)
    {
        dim3 grid(iceil(M, blockSize - 2), iceil(N, blockSize));
        dim3 blocks(blockSize, blockSize);
        case_2<blockSize><<<grid, blocks>>>(
               inputgpu, output2gpu, N, M);
    }
    verifyErrors();

    cudaMemcpy(output1, output1gpu, N * M * sizeof(float),
        cudaMemcpyDeviceToHost);
    cudaMemcpy(output2, output2gpu, N * M * sizeof(float),
        cudaMemcpyDeviceToHost);

    cudaFree(inputgpu);
    cudaFree(output1gpu);
    cudaFree(output2gpu);

    std::cout << "case 1 test: " <<
        compare(output, output1, N, M) << std::endl;
    std::cout << "case 2 test: " <<
        compare(output, output2, N, M) << std::endl;

    return 0;
}
```

Bibliography

[1] Todd A Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. Parallelizing julia with a non-invasive dsl. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[2] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F Donaldson, Jeroen Ketema, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*, pages 138–149. IEEE, 2015.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A code optimization framework for high performance systems, 2018.

[4] Soufiane Baghdadi, Armin Größlinger, and Albert Cohen. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, July 2010.

[5] Michel Barreteau and Claudia Cantini. Signal processing: Radar. In *Smart Multicore Embedded Systems*, pages 125–138. Springer, 2014.

[6] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic c-to-cuda code generation for affine programs. In Rajiv Gupta, editor, *Compiler Construction*, pages 244–263, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Ulysse Beaugnon, Alexey Kravets, Sven Van Haastregt, Riyadh Baghdadi, David Tweed, Javed Absar, and Anton Lokhmotov. Vobla: A vehicle for optimized basic linear algebra. In *ACM SIGPLAN Notices*, volume 49, pages 115–124. ACM, 2014.

[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[10] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[11] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 8. ACM, 2014.

[12] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm.

[13] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[14] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838. IEEE, 2008.

[15] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[16] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[17] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench*, 2012.

[18] Jarrett Revels. Cassette.jl. `https://github.com/jrevels/Cassette.jl`, 2018.

[19] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. Splatt: Efficient and parallel sparse tensor-matrix multiplication. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 61–70. IEEE, 2015.

[20] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[21] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[22] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[23] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *International Congress on Mathematical Software*, pages 299–302. Springer, 2010.