Efficient Memory and GPU Operations for Tiramisu Compiler

by

Abdurrahman Akkas

S.B., Massachusetts Institute of Technology (2018)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....
Department of Electrical Engineering and Computer Science
May 28, 2019

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

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

Efficient Memory and GPU Operations for Tiramisu Compiler

by

Abdurrahman Akkas

Submitted to the Department of Electrical Engineering and Computer Science
on May 28, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The increasing complexity of computer architectures with different memory hierarchies and parallelism characteristics makes generating efficient code a difficult task. Achieving high performance requires complex schedules and data layout transformations which might not be easy to express in a low level language.

Tiramisu [3] is an optimization framework for generating efficient code for different platforms including CPU, GPU, and distributed systems. It combines the polyhedral intermediate representation with rich scheduling and data layout commands, creating a high level interface to generate high performance code.

In this thesis, we present new memory interfaces and GPU operators implemented to extend Tiramisu compiler. We demonstrate that these features enable users to generate high performance GPU code with concise Tiramisu programs. We also evaluate Tiramisu's GPU backend with two benchmarks, matrix multiplication and a recurrent neural network architecture, showing that Tiramisu outperforms other polyhedral compilers and popular library implementations.

Thesis Supervisor: Saman P. Amarasinghe
Title: Professor

Acknowledgments

First, I would like to thank my supervisor, Saman Amarasinghe, for giving me the opportunity to be a part of the COMMIT group. I would not be able to develop academically and professionally without his guidance and supervision.

I also would like to thank Riyadh Baghdadi for his patience and support. His knowledge, experience, and ever smiling face were the greatest motivators of my work and thesis.

I would like to thank my parents, Aysel and Bekir Akkas, for always believing in and supporting me, and my brother Behzat Akkas who kept inspiring me from the very beginning and made me the person who I am today.

Last but not least, I would like thank my close friends Mustafa Camurcu, Enes Kocabey, and Nusret Tas for making my Master's journey fun and bearable, and Bahrudin Trbalic for being a constant source of motivation and spiritual fellowship.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **11**

2 Background **13**

2.1 Tiramisu 13

2.2 CUDA Programming Model 15

2.2.1 Memory Spaces 16

2.3 Tiramisu GPU Backend 18

2.4 Related Work 19

3 Tiramisu GPU Benchmarking **21**

3.1 Matrix Multiplication (GEMM) 21

3.1.1 Method 22

3.1.2 Evaluation 25

3.2 Recurrent Neural Networks (LSTM) 25

3.2.1 Method 27

3.2.2 Evaluation 28

4 High Level Interfaces for Tiramisu **31**

4.1 Motivation 31

4.2 Implicit Buffers 31

4.3 Block API 32

4.4 Shared Memory Interface 34

4.4.1 Background 34

4.4.2 Interface 35
4.4.3 Implementation 36
4.5 cuBLAS GEMM Interface 37
4.6 CUDA Streams Interface 38
4.7 Evaluation 39

5 Conclusion **43**

A GEMM Implementation in Tiramisu **45**

B LSTM Implementation in Tiramisu **51**

List of Figures

3-1 Data dependencies in a multilayer network 27
3-2 Traversal of LSTM cells before and after skew-interchange operation . 28

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

As we are approaching the end of the Moore's law which predicted computational power of integrated circuits will double every two years, the advancements in performance are now coming from application specific architecture designs and better utilization of these systems instead of mere increase in clock speeds and transistor densities. The computational systems available for programmers in the last few decades proliferated from a single general purpose CPU to multi-core CPUs, multi-CPU distributed systems, GPUs, FPGAs, and so on. All these systems have different use cases with different performance characteristics. Therefore the user should consider different optimization strategies for each system. Moreover, the optimization parameters depend on the system specifications such as the size of the memory space at different levels, cache sizes, and available parallelism. Thus one should try different approaches and tune several parameters to get the maximum performance for each system even though the end goal of the program is the same for all. This makes it difficult to write high performance programs especially with low level languages.

Tiramisu is a compiler that offers a unified optimization framework for different target architectures and aims to achieve high performance on data parallel programs operating on dense arrays. By separating the algorithm from scheduling and data layout transformations Tiramisu provides a clean an easy way to express different optimization techniques.

In this thesis, we focus on the design and performance of the GPU backend of

Tiramisu. Even though Tiramisu offers a comprehensive GPU and buffer interfaces, implementing GPU programs is a tedious task since certain memory operations are needed to be done manually. In chapter 4 we present several features implemented in Tiramisu that increase the ease of use of the framework and offer new ways of optimizing. This includes an implicit buffer API, a high level interface for GPU shared memory, and integration of CUDA streams API to Tiramisu. In chapter 3 we show that Tiramisu is able to generate efficient GPU code by implementing two benchmarks. In GEMM benchmark Tiramisu implementation significantly outperforms the other polyhedral language implementations while having comparable results to cuBLAS library. In LSTM benchmark Tiramisu outperforms the cuDNN library on deep recurrent neural networks.

Chapter 2

Background

In this chapter we give an overview of the existing work. Section 2.1 introduces Tiramisu and gives an example Tiramisu program. Sections 2.2 and 2.3 explains CUDA programming and Tiramisu GPU backend. Section 2.4 discusses the related work.

2.1 Tiramisu

Tiramisu [3] is a code optimization framework embedded in C++. It divides its intermediate representation into four layers. At the first layer the programmer constructs the abstract algorithm without specifying the loop nests or data layouts. The second layer determines the order of computations as well as what kind of loop nest transformations should be applied to each computation. In the third layer, programmer declares the input and output buffers of the function as well as where should the intermediate results should be stored. The final layer specifies the communication in the case of distributed computing. Listing 2.1 shows an example Tiramisu program. After generating code, user can link the resulting binary to use the function.

Tiramisu internally uses the Integer Set Library [7] to represent the loop nests and buffer mappings, and to generate the abstract syntax tree of the program. This allows the expression of affine transformations and polyhedral iteration spaces within the program.

Tiramisu can target GPU architectures by generating CUDA code. This is done by mapping loop levels to block and thread dimensions in CUDA programming model and adding instructions to move data between different memory levels of GPU.

Listing 2.1: An example Tiramisu function

```
// Initialize Tiramisu with the name of the function
tiramisu::init("my_program");
// Define loop variables and bounds
var i("i", 0, 128), j("j", 0, 128);
// Layer I
input A({i, j}, p_float32);
computation B({i, j}, expr(o_sqrt, A(i, j)));
computation C({i, j}, (B(i, j) + B(i, j + 1)) / 2);
C.add_predicate(j < 127);

// Layer II
// Scheduling operations
C.tile(i, j, 16, 16);
// Order of computations
B.then(C, computation::root);

// Layer III
// We store B and C in the same buffer to save space
B.store_in(C.get_buffer());

// Layer IV
// Nothing is needed here since there is no communication

// Code generation
tiramisu::codegen({A.get_buffer(), B.get_buffer()}, "out.o");
```

2.2 CUDA Programming Model

For the computational tasks that can be expressed as data parallel programs GPUs provide significant performance benefits over CPUs. In the past, GPUs were only specialized in graphical processing tasks. However, users can now utilize GPUs for more general tasks thanks to new interfaces like CUDA.

CUDA is an application programming interface for Nvidia GPUs [5]. It offers an interface similar to C language where users can express programs to be executed on GPU threads. GPU programs, which are also called kernels, are annotated C functions. When launching the kernel on GPU, user specifies how many instances of the kernel they want to execute. These instances run in parallel on GPU cores in different threads. User splits the computational task to different threads by using the unique index that each thread gets. User may specify one to three dimensional indexing schemes depending on the use case. For example, using a 1D indexing is convenient when working on a single dimensional data such as sound processing, while image processing with 2D indexing would be more convenient where thread indices correspond to pixel coordinates.

Listing 2.2: An example CUDA C kernel function and call

```
// A simple kernel that copies values from A to B
__global__ void mykernel(float *A, float *B) {
    int index = blockIdx.x * 16 + threadIdx.x;
    B[index] = A[index];
}
int main() {
    float *A, *B;
    // Allocate A and B of size 160
    dim3 blocks(16, 1, 1);
    dim3 threads(10, 1, 1);
    mykernel<<<blocks, threads>>>(A, B);
}
```

CUDA threads are tiled into fixed size groups called thread blocks. While launching the kernel user specifies the size of the thread block and how many of these blocks should be created. The number of threads per block is limited to 1024. Thread blocks are executed by Streaming Multiprocessors (SM) on GPU, which manage the registers, shared memory spaces, and other resources. Blocks are further divided into *warps* of size 32. Warps run in the single-instruction-multiple-thread fashion, where all threads of a warp execute the same instruction simultaneously. In the case of branching, diverging thread groups wait for each other to execute until they get back to the same instruction. Thus it is important to reduce branch divergence within a warp to increase warp utilization and get good performance.

2.2.1 Memory Spaces

There are several different memory spaces on an Nvidia GPU with different size and latency characteristics.

Global Memory

Global memory, which is also called the device memory, is the largest memory space on the GPU usually with capacities of tens of gigabytes. It is an off-chip DRAM where all threads of the GPU access with a common address space. Before executing a program, user should copy the data from CPU memory to global memory to access from the kernels. Since it is off-chip accesses to global memory are slow compared to other GPU operations and it is often the case that programs are bounded by global memory access. Accesses to global memory are cached on one or two layers depending on the device. L1 cache has 128-byte lines whereas L2 cache has 32-byte lines. Accesses to global memory from the same warp are coalesced if they are on the same cache line. Thus accessing the same or consecutive addresses from the same warp is the optimal way to use the global memory.

Shared Memory

Shared memory is an on-chip memory space that is local to each thread block. It is not accessible from the CPU, and has much lower latency compared to global memory. Shared memory can be used to communicate between different threads of the same block, and enables coalescing the global memory accesses and eliminating redundant copies in several cases. Shared memory space is divided into 32 banks, where successive 32-bit words are served from a different bank in a circular way. Accesses to the same memory bank are serialized and user should make sure to use different banks from different threads to get good performance. However, if threads of the same warp read the same address the access is broadcasted, meaning that there is no serialization overhead. The size of the shared memory per block depends on the device and configuration, and it is usually 48KB.

Local Memory/Registers

In addition to global and shared memory, user can define variables and arrays local to the GPU thread. Variables are hosted on the register space of the chip. This is also the case for local arrays if accesses can be inferred statically during compile time. However, the per-thread register space is limited by usually 255 32-bit registers per thread. If the number of variables used exceeds this limit the data will be placed in global memory instead. This greatly reduces the performance of the kernel as global memory access is much slower than register access.

Constant Memory

In the case that kernels do not need to modify the data, constant memory is the optimal solution for storage. It is located off-chip but is cached, thus it can be as fast as a register access if same address is accessed multiple times. However, accesses to constant memory from the same warp are serialized, thus number of distinct address accessed should be minimized.

2.3 Tiramisu GPU Backend

In TIRAMISU, GPU programs are created by mapping loop iterators to GPU block and thread dimensions. `computation::tag_gpu_level` method takes a list of consecutive loop variables as argument where first half of the variables are mapped to block dimensions and second half is mapped to thread dimensions. The function has three overloads in case user wants to user 1, 2, or 3d indexing for threads:

```
comptutation::tag_gpu_level(i0, i1)
comptutation::tag_gpu_level(i0, j0, i1, j1)
comptutation::tag_gpu_level(i0, j0, k0, i1, j1, k1)
```

Loop iterators mapped to thread indices need to have constant bounds. Thus it is common practice to tile two loop levels and map resulting dimensions to GPU. `computation::gpu_tile` method offers a convenient way to do this in one step.

```
comptutation::gpu_tile(i, j, sizeX, sizeY, i0, j0, i1, j1)
```

TIRAMISU generates CUDA C kernel code for the computations that are mapped to GPU, and compiles and links the GPU program to the main binary. Loop nests that are tagged as block and thread indices are replaced with kernel calls during code generation. TIRAMISU maps whole loop nests to kernels instead of individual computations, thus computations that are fused within a GPU tagged loop nest will appear in the same kernel.

CPU memory is not accessible from the GPU kernel, thus user needs to use GPU buffers to store GPU computations. GPU buffers are created by tagging regular buffers with global, shared, constant, local, or register tags. Allocation of global and constant buffers are handled by TIRAMISU. However, shared and local buffers, and registers need to be allocated manually by user. This should be done by creating a computations of `allocate(buffer)` expressions and scheduling them in the same kernel as the computation that uses the buffer. Since TIRAMISU functions take in CPU buffers as input, user needs to handle the copies back and forth between CPU and GPU buffers with `memcpy` function.

Listing 2.3: An example GPU program in Tiramisu

```
tiramisu::init("my_gpu_program");
var i("i", 0, 128), i("j", 0, 128);
// Define inputs and computations
input A({i, j});
computation B({i, j}, A(i, j) + A(i + 1, j + 1));
B.add_predicate(i < 127 && j < 127);
// Tag computation buffers as global GPU buffer
A.get_buffer()->tag_gpu_global();
B.get_buffer()->tag_gpu_global();
// Declare CPU buffers
buffer A_cpu("A_cpu", {128, 128}, p_float32, a_temporary);
buffer B_cpu("B_cpu", {128, 128}, p_float32, a_temporary);
// Declare CPU-GPU copies
computation copy_A({}, memcpy(A_cpu, *A.get_buffer()));
computation copy_B({}, memcpy(*B.get_buffer(), B_cpu));
// Map computation B to GPU
B.gpu_tile(i, j, 16, 16);
// Schedule copies and computation
copy_A.then(B, computation::root)
      .then(copy_B, computation::root);
// Generate object file
tiramisu::codegen({A.get_buffer(), C.get_buffer()}, "out.o");
```

2.4 Related Work

There are several existing domain specific languages and polyhedral compilers that target GPUs. Tiramisu differs from most by allowing programmers to manually specify transformations and hand tune the optimizations. PPCG [8] [2] is a polyhedral compiler that accepts C programs as inputs and transforms them into CUDA code.

PPCG does not implement optimizations like prefetching and register blocking that are available in TIRAMISU and scheduling is fully automated.

Tensor Comprehensions (TC) [6] is a domain specific language developed by Facebook that uses a polyhedral back end. It aims to generate optimized GPU code for custom deep learning operators. Unlike TIRAMISU, TC autotunes the optimization parameters and relies on the scheduling that PPCG provides which is fully automatic. Diesel [4] is another language developed by Nvidia that generates fully automated schedules from high level algorithms. It has comparable results to hand tuned library functions for linear algebra and neural networks for certain problem sizes.

Chapter 3

Tiramisu GPU Benchmarking

To show the capabilities of TIRAMISU as a high performance compiler several benchmarks have been implemented and compared against implementations of standard libraries as well as other high level compilers in terms of performance.

3.1 Matrix Multiplication (GEMM)

Matrix multiplication is an essential component of the most computation heavy tasks. In the context of deep learning, fully connected layers of a network is computed by matrix multiplication of the batched input with the weight matrix. Other seemingly unrelated operations such as convolution can also be expressed as a matrix multiplication. Indeed, about 95% of the time spent in the popular ImageNet architecture is on operations implemented with matrix multiplication [9]. Therefore, achieving good performance in matrix multiplication will prove the effectiveness of TIRAMISU on variety of tasks.

Generalized Matrix Multiplication (GEMM) is the specification used in the BLAS linear algebra libraries for matrix multiplication programs. Given input matrices A, B, and C, and coefficients α and β, it is defined as an update operation on C:

$$C \leftarrow \alpha\ A \times B + \beta\ C$$

Listing 3.1: A naive implementation of the GEMM algorithm in C

```
for (int i = 0; i < M; i++) {}
    for (int j = 0; j < N; j++) {
        float sum = 0;
        for (int k = 0; k < K; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = alpha * sum + beta * C[i][j];
    }
}
```

3.1.1 Method

Matrix multiplication is a massively data parallel task as the same reduction operation is applied to all different combinations of rows of matrix A and columns of matrix B. Therefore, GPU stands out as the right tool to optimize GEMM.

Listing 3.1 gives a naive implementation of GEMM on CPU. A natural way of mapping the computation to GPU is to make each GPU thread compute a single $C[i][j]$ point in the output matrix. To achieve this in TIRAMISU, i, j loop nest is `gpu_tile`d with blocks of size 16x16.

After tiling, each thread block computes a 16x16 region in the C matrix, reading a row band of height 16 from A and a column band of width 16 from B. However, each thread fetches values from the global memory separately, even though threads of the same row and column are using the same A and B values. This gives an opportunity to utilize the shared memory space on GPU. After splitting k loop by 16, the 16x16 submatrices of A and B used in the inner loop nest are copied to shared memory at each iteration of the outer loop, and then accesses to global memory are replaced with shared memory.

Listing 3.2: Inner product after splitting

```
for (int k0 = 0; k0 < K / 16; k0++) {
    // Shared memory copies should be made here
    for (int k1 = 0; k1 < 16; k1++) {
        sum += A[i][k0 * 16 + k1] * B[k0 * 16 + k1][j];
    }
}
```

Performance of GPU programs are usually bounded by global memory access latencies. This is the case as well in the serial copy-compute algorithm described above. For each iteration kernel needs to wait for the data to be copied from the global memory before proceeding to multiplication. However, GPU supports instruction level parallelism (ILP), where memory copies can be executed simultaneously with floating point instructions if operations are at different addresses. ILP is utilized in the GEMM algorithm by prefetching the data blocks. At each iteration of $k0$, global memory copy for the next block as well as the multiplication on the current block is executed simultaneously, and threads are synchronized at the end of iteration. Shared memory buffer is duplicated such that copy and multiplication operations alternate between two buffers at each iteration. An additional copy computation is added to the beginning of the kernel fetch the first block.

In addition to global and shared memory spaces, GPUs also offer big register space per kernel. A single thread can use up to 255 registers, and algorithm explained above only uses a small portion of this since each kernel is computing a single location in the output matrix. As a final step to the algorithm, utilization of the register space is increased by tiling the i, j loop nest one more time. The inner block of the tile stays inside the GPU kernel so that each GPU thread computes a small rectangular region of the output matrix instead of a single point. This greatly reduces the number of share memory accesses since threads reuse the values copied from shared memory to registers to compute different outputs. To maximize the number of registers used while keeping the shared memory allocation within the limit, a rectangular $16x6$ tile

size is used. Listing 3.3 shows the final pseudocode in an imperative style. The complete implementation in TIRAMISU is given in Appendix A.

Listing 3.3: Pseudocode of the final kernel

```
// Prefetch:
// Allocate accumulators in register space
allocate C_reg[16][16]
A_sh <- A_glb
B_sh <- B_glb
synchronize kernels
for k0 = 0:K/16
  // Fetch blocks of next iteration
  A_sh <- A_glb
  B_sh <- B_glb
  for k1 = 0:16
    // Copy values to registers
    A_reg <- A_sh
    B_reg <- B_sh
    for i = 0:16
      for j = 0:6
        // Multiply
        C_reg[i][j] += A_reg[i] * B_reg[j]
  synchronize kernels
// Copy values to global memory
for i = 0:16
  for j = 0:6
    C_glb[i][j] = alpha * C_reg[i][j] + beta * C_glb[i][j]
```

3.1.2 Evaluation

The performance of the algorithm is tested on multiplication of square matrices of size 3072x3072. Run time is compared against cuBLAS, which is a linear algebra library developed by Nvidia. cuBLAS is highly optimized with assembly level hand tuning and it uses several operations like vector copies which are not available in Tiramisu. Table 3.1 shows the run time of the algorithm on two different GPU models.

Through only high level scheduling and data layout commands, Tiramisu is able to achieve 1.4-2x runtime compared to cuBLAS. This is significantly better than other polyhedral compilers such as PENCIL [2] and Tensor Comprehensions [6], which are about 10x slower [3].

GPU Model	Tiramisu	cuBLAS
K80	42ms	17ms
P4	15ms	11ms

Table 3.1: Execution time of the GEMM implementation

Even though Tiramisu interface is high level compared to CUDA C, using shared memory and other GPU operation are still tedious and error prone since declarations, copies, and access updates should be done manually. This inspired the implementation of a higher level API for shared memory, which is discussed in section 4.4.

3.2 Recurrent Neural Networks (LSTM)

In recent years, Recurrent Neural Networks (RNN) became an essential tool for solving important Deep Learning problems. RNNs are used in the tasks where input data comes in a streaming fashion, such as speech recognition, robot control, or time series prediction. The network holds an internal state, and updates it at each step while computing the output. This allows network to infer and store high level information about the input, which is not possible to do with fixed size feed forward neural networks.

One of the most prominent RNN architectures is the Long Short-Term Memory (LSTM). In recent years, LSTM is used with great success for variety of tasks includ-

ing speech recognition, machine translation, and training video game agents that can exceed human performance.

The common LSTM architecture used in benchmarking TIRAMISU is made up of cells. Each cell keeps track of two internal state vectors h and c, and takes in a single input vector x. Using these three vectors, the cell computes new internal states h' and c' through several matrix multiplication and non-linear operations, and outputs h' to the next layer of the network. Equation 3.1 shows the operations in the LSMT cell.

$$\begin{aligned} i &= \sigma(W_i x + R_i h + b_i) \\ f &= \sigma(W_f x + R_f h + b_f) \\ o &= \sigma(W_o x + R_o h + b_o) \\ c' &= f * c + i * tanh(W_c x + R_c h + b_c) \\ h' &= o * tanh(c') \end{aligned} \tag{3.1}$$

LSTM cells can be stacked on top of each other, where output of one cell becomes the input of the next. Parallelizing LSTM is a difficult task. Due to sequential data flow in LSTM, a single layer network's cells cannot run in parallel. Similarly, in a single iteration of multilayer LSTM, data flows sequentially from one layer to next, thus different layers cannot be executed in parallel. However, sequential iteration of multilayer LSTM has diagonal bands of cells which do not have data dependencies among themselves. Therefore one can implement wavefront parallelism when executing a multilayer LSTM. The polyhedral model is a good fit for this implementation since it requires affine transformations to the loop nest before parallelization can be done. Figure 3-1 represents the data dependencies between different cells which shows that diagonal cells are indeed independent and can be parallelized.

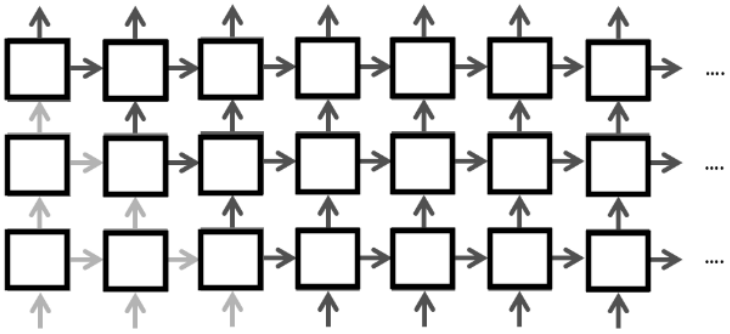

Figure 3-1: Data dependencies in a multilayer network

3.2.1 Method

The LSTM algorithm implemented in Tiramisu is based on the ideas from Appleyard's cuDNN paper [1]. Two outer loop nests span the layers and iterations of LSTM cells in the network, where each LSTM cell does two GEMM operations inputs coming from two channels and then non-linear operations to compute the output.

GEMM operations are executed using cuBLAS library in order to reduce the program complexity and utilize the performance of the external library. Tiramisu offers a convenient interface for multiplying Tiramisu buffers using cuBLAS. Section 4.5 discusses the implementation of this interface.

The layer-iteration loop nest needs to be traversed diagonally to achieve wavefront parallelism. In an imperative language this would require calculating nontrivial loop bounds. However, thanks to polyhedral nature of Tiramisu, this can be easily achieved using high level commands. First, the rectangular iteration space is skewed into a parallelogram. Then iterators are interchanged, which makes the inner loop traverse a diagonal path in the original rectangular iteration space.

Listing 3.4: Wavefront Parallelism in Tiramisu

```
lstm_block.skew(l, s, 1, l_s, s_s);
lstm_block.interchange(l_s, s_s);
lstm_block.parallelize(l_s);
```

Listing 3.4 lists the operations needed to implement wavefront parallelism in Tiramisu. `skew` method takes in two original consecutive loop nests `l` and `s`, a skewing factor that is `1` in this case, and two new loop iterators for the new iteration domain, `l_s` and `s_s`. `interchange` method takes two consecutive loop iterators and flips their order in the iteration domain. Static and dynamic loop bounds of the new iterators are computed automatically by the compiler.

Finally, the inner loop nest of the diagonal traversal is parallelized to execute cuBLAS GEMM operations and kernel calls of different LSTM cells in parallel. Tiramisu uses CUDA streams API to achieve concurrency on parallelized GPU operations, as discussed in section 4.6.

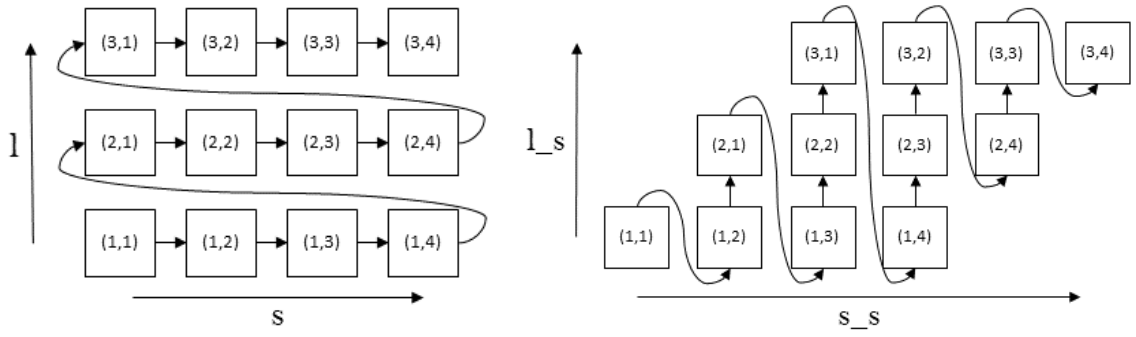

Figure 3-2: Traversal of LSTM cells before and after skew-interchange operation

As a final optimization step, GEMM operations on the inputs from previous layers are batched as they use the same weight matrices for same layers. This reduces the overhead of GEMM calls and it is easily combined with the wavefront parallelism optimization. Appendix B gives the complete implementation of the program in Tiramisu.

3.2.2 Evaluation

The LSTM algorithm is evaluated on an Nvidia K80 GPU and compared against cuDNN 7 library. For deep LSTM networks Tiramisu matches or outperforms the performance of cuDNN. The algorithm also tested on double precision input, which gives similar results. The performance difference is especially significant on deeper

networks, which suggests that TIRAMISU utilizes better multi kernel parallelism than cuDNN.

	TIRAMISU	cuDNN7
4 layers float	57.9ms	57.1ms
8 layers float	112.1ms	127.0ms
4 layers double	106.7ms	111.8ms
8 layers double	215.4ms	234.9ms

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

High Level Interfaces for Tiramisu

4.1 Motivation

Implementation of two different benchmarks are shown in Chapter 3. As discussed earlier, insights gained from these implementations show that Tiramisu requires too many low level operations to get good performance, especially for GPU backend, and some existing interfaces are tedious and require too much boilerplate code (see Appendix A). This chapter discusses the new high level API and features added to Tiramisu to address these problems.

4.2 Implicit Buffers

In Tiramisu all typed computations need to be assigned to buffers, as their results need to be stored somewhere on the memory. Users need to create buffer objects and specify the mapping for each computation. However, in the general case the sizes and types of buffers can be inferred from the context of the program.

Implicit buffer API reduces the boilerplate code needed for the memory layout mapping of Tiramisu programs by automatically allocating buffers during computation declarations. By default, the implicit buffer has the same dimensionality and size of the loop nest of the computation and a one-to-one mapping between the computation and buffer is created. If the size of the loop nest cannot be inferred from the

computation iterators or computation does not have a scalar data type buffer declaration is skipped and user needs to handle memory layout manually. The implicit buffer can be accessed using `computation::get_buffer` method.

If user wants to declare a custom memory mapping, `computation::store_in(mapping, size)` method can be used, which updates the size of the implicit buffer and memory mapping. The type information of the implicit buffers (whether they are a_input, a_output, or a_temporary) are inferred from whether the computation is scheduled or not and whether the buffer appears on the argument list of the TIRAMISU function. Listing 4.1 shows an example use case where two successive stencils are applied on an input image. TIRAMISU allocates two buffers of size 128x128 for A and C, and computation B writes to the same buffer as C. See listings 4.3-4.4 for a comparison between the new and old APIs.

Listing 4.1: Implicit Buffer API

```
var i("i", 0, 128), i("j", 0, 128);
input A({i, j});
computation B({i, j}, A(i, j) + A(i + 1, j + 1));
computation C({i, j}, B(i + 1, j) + B(i, j + 1));
B.store_in(C.get_buffer());
tiramisu::codegen({A.get_buffer(), C.get_buffer()});
```

4.3 Block API

It is often the case that TIRAMISU users want to apply the same transformations to a set of different computations, especially for the computations in GPU kernels. For example, lets say user wants to apply a chain of stencils to an input image:

```
A(i, j) = (I(i + 1, j) + I(i, j + 1)) / 2
B(i, j) = (A(i, j) + A(i + 1, j + 1)) / 2
C(i, j) = tanh(A(i, j) + B(i, j))
```

To get good performance user will apply several scheduling operations to these computations. Since these are successive computations, user would want to apply similar schedules:

```
A.tile(i, j, 16, 12, i0, j0, i1, j1)
B.tile(i, j, 16, 12, i0, j0, i1, j1)
C.tile(i, j, 16, 12, i0, j0, i1, j1)
A.parallelize(i0)
B.parallelize(i0)
C.parallelize(i0)
A.vectorize(j1)
B.vectorize(j1)
C.vectorize(j1)
...
```

This becomes a tedious process as user will need to apply each transformation to every computation. It is also difficult to maintain since whenever a parameter such as tile size is changed, it needs to be replaced for all instances of the computations to maintain correctness of the program.

The Tiramisu block interface is implemented to solve this problem. In the new interfce user groups computations into blocks, and apply the scheduling operations on blocks instead. The block class is implemented as a child class of computation, which allows the creation of nested blocks if need be. Other than distributing the scheduling commands, blocks do not impose any ordering or loop nest constraints:

```
block stencils({&A, &B, &C});
stencils.tile(i, j, 16, 12, i0, j0, i1, j1)
stencils.parallelize(i0)
stencils.vectorize(j1)
...
```

4.4 Shared Memory Interface

The performance bottleneck of an unoptimized GPU program is usually the global memory access latencies, since a global memory fetch can be several orders of magnitude slower compared to operations on the register space. To get good performance, it is important to reduce the global memory access as much as possible and utilize instruction level parallelism by pipelining the data copies.

The on-chip shared memory serves as an intermediate layer for global memory accesses by providing a common memory space for thread blocks. It is common practice to copy values needed by the thread block to shared memory first and then use the values from shared memory in order to reduce the number of redundant copies and increase the coalescing. Section 3.1 is an example for this in the context of matrix multiplication.

4.4.1 Background

In TIRAMISU, shared memory space is accessed by tagging TIRAMISU buffers with `tag_gpu_shared` method. This tag causes the buffer to be removed from auto allocation list used for Halide buffers, as the buffer needs to be declared within the GPU kernel. However, user needs to declare the buffer manually as there is no auto allocation feature in TIRAMISU for shared memory buffers. To do that, user creates an allocation computation and schedule it to the beginning of the kernel.

To copy values from global memory to shared memory, user needs to declare another computation. To get good performance for this computation, user needs to make sure that consecutive values in the global memory are accessed from the same warp to coalesce the copies and different shared memory banks are used for concurrent copies to reduce bank conflict. This often requires complicated indexing schemes in global and shared memory buffers as well as array transposition. User also needs to manually schedule the copy computation and extra loop nests and predicates might be needed if the thread block size is not the same as the size of the data that is copied. After the copy computation, user also needs to replace the original access to

the global memory with shared memory.

As explained above, the low level TIRAMISU API for using shared memory is tedious and error prone, and it requires advanced knowledge on hardware. Listing 4.3 gives an example use case for the shared memory, where the program computes the transpose of an input matrix. Users need to create several computations and get complicated indexings right before they can use the shared memory. Another problem is that TIRAMISU does not support operations like module or division in the access expressions of computations. Therefore, a workaround input computation needs to be created whenever user needs an access pattern with modulo. However, all these information can be generated by the compiler correctly and efficiently. The `cache_shared` method solves this problem by presenting a simple interface to the user.

4.4.2 Interface

The `cache_shared` method has the signature:

```
computation *computation::cache_shared(
        computation &inp,
        const var &level,
        const std::vector<int> buffer_shape,
        const std::vector<expr> copy_offsets,
        bool pad_buffer=false)
```

It is implemented as a method for the `tiramisu::computation` class. For example if computation C uses computation A as input, user should call `C.cache_shared` with A as the first argument of the method. The second argument is the loop level where copy operation should be performed. `cache_shared` creates necessary computations and access updates to declare and allocate a shared memory buffer, copy subarray of A used under the given loop level to the new buffer, and replace accesses in C to use the shared memory instead of global memory when accessing A. The function is not fully automated since user needs to pass the size of the buffer needed and the

offset of the buffer into A at each iteration as arguments. The optional `pad_buffer` argument can be set to true, which pads the shared memory buffer by 1 in the innermost dimension. This slightly increases the shared memory usage, however it can give significant performance benefits by reducing shared memory bank conflicts if access pattern is poorly aligned with the buffer shape. The method returns an access computation to the shared memory buffer with the same signature as the input computation, in case user wants to access to shared memory buffer later on.

4.4.3 Implementation

`cache_shared` method is implemented in several steps. First, a shared buffer is created with the size given in the arguments. Then a new access computation is generated that wraps the input buffer and accesses to the old input are replaced with this new computation. This means whenever the computation accesses the input, it uses shared memory instead of global memory. After, the computation that declares the buffer inside kernel is created. This computation should appear in the beginning of the kernel, thus the iteration domain of the declaration is generated by projecting out dimensions inside the kernel body.

The next step of the method is to copy the data from global memory to shared memory before it is used by the computation. This copy is done collectively by the thread block and each memory address is assigned to a different thread. If there are more values to be copied than there are threads, assignment is done in a round robin way. Assignment also ensures the maximum coalescing in global memory accesses. The iteration domain for the copy computation is generated similar to the buffer declaration. However, additional loop nests are added if threads copy more than one value.

Finally, the declaration and copy computation are scheduled by traversing the schedule graph to find the appropriate locations. Synchronization calls are added before and after the copies to make sure there are no data races between threads.

4.5 cuBLAS GEMM Interface

As a program developer, it is important to be able incorporate the existing libraries to your program in order to reduce the program complexity and to utilize the performance of the optimized binaries. Section 3.2 is an example where matrix multiplication functions from cuBLAS library is used to implement LSTM cells. TIRAMISU implements a convenient interface to allow TIRAMISU buffers to be multiplied with cuBLAS sgemm or dgemm functions. The interface also allows more complicated operations through optional arguments:

```
expr cublas_gemm(const buffer &A, const buffer &B, buffer &C,
        expr M, expr N, expr K,
        expr alpha = 1, expr beta = 0,
        expr ldA = 0, expr ldB = 0, expr ldC = 0,
        expr offsetA = 0, expr offsetB = 0, expr offsetC = 0,
        expr transposeA = false, expr transposeB = false);
```

The `cublas_gemm` function takes in three input buffers of types float or double and three variables M, N, and K which determine the sizes of the buffers. *alpha* and *beta* are provided as optional arguments in case user wants to use the generalized matrix multiplication interface. User is also able to multiply submatrices which might be transposed as well by setting the stride, offset, and transpose parameters accordingly.

One important caveat about the implementation of `cublas_sgemm` is that cuBLAS—and CUDA libraries in general—work in column-major format, as they follow the Fortran convention. However, TIRAMISU is C-based and buffers are stored in row-major format instead. One can pretranspose the input matrices and transpose the output matrix back in order to get two row-major matrices multiplied with cuBLAS. However, transposition is costly and can be avoided with a small observation. From a row-major perspective, the column-major cublasSgemm function basically works on

transposed versions of matrices.

$$\begin{aligned}(\text{cublasSgemm}(A,B))^T &= A^T \times B^T \\ \text{cublasSgemm}(A,B) &= (A^T * B^T)^T \\ \text{cublasSgemm}(A,B) &= B * A \\ \text{cublasSgemm}(B,A) &= A * B\end{aligned} \tag{4.1}$$

As derivation 4.1 shows, simply swapping the input matrices give the same outcome as a row major multiplication. The additional arguments that cublasSgemm take are configured accordingly to accomodate this change.

4.6 CUDA Streams Interface

An Nvidia GPU has several streaming multiprocessor (SM) units executing thread blocks. If number of blocks launched is less than the number of SMs available, the GPU is not fully utilized. To increase the blocks available to run, CUDA allows multiple kernels to be launched simultaneously through streams API, where user specifies which stream each kernel should be launched with. Kernels running in the same stream are executed serially, while different streams might be executed in parallel. User should use the synchronization and events APIs of CUDA to enforce orderings for executions of kernels and pipeline the streams.

Tiramisu avoids incorporating the low level CUDA streams/events API since it would unnecessarily increase the complexity of the GPU interface. Instead, a more intuitive streaming API is provided using per-thread CUDA streams. CUDA 7 introduces a new compiler flag "`--default-stream per-thread`", which initializes a new CUDA stream for each CPU thread that launches a kernel. Thus, kernels launched from different threads can run on the GPU in parallel.

To achieve concurrency with streams API in Tiramisu, user can simply parallelize a loop nest that wraps a GPU kernel, which launches the kernels from different thread streams. However, kernel calls are non-blocking and user should make sure that they

are terminated before parallel threads join back to the main thread, in order to prevent data races. `cuda_stream_synchronize` expression is provided in TIRAMISU for this purpose, which blocks until all kernels of the current thread are executed. `cublas_gemm` interface discussed in section 4.5 is also configured to use per-thread stream instead of the default global stream.

Listing 4.2: An example use of stream API

```
computation C({i, j, k}, 0);
computation sync({i}, cuda_stream_synchronize());
C.gpu_tile(j, k, 16, 16);
C.parallelize(i);
C.then(sync, i);
```

4.7 Evaluation

Listings 4.3 and 4.4 show an example program written in the old and new APIs. The program does a matrix transposition on GPU. In the old API user defines all buffers and mappings manually. Shared memory is used to ensure memory coalescing in global memory accesses and this involves using complex indexing schemes for memory layout commands in the old API.

The new API offers a much cleaner and understandable interface for the memory operations without sacrificing performance. The underlying buffers for A and B are created automatically. Shared memory operation is handled by `cache_shared` method which creates necessary buffers and copy computations automatically.

Listing 4.3: An example transposition program with low level shared memory API

```
// Skipping the CPU-GPU copies
var i("i", 0, 128), j("j", 0, 256);
var i0("i0"), i1("i1");
var j0("j0"), j1("j1");
```

```
// Buffers
buffer A_buf("A_buf", {128, 256}, p_float32, a_input);
buffer B_buf("B_buf", {256, 128}, p_float32, a_input);
buffer A_shr("A_shr", {16, 16}, p_float32, a_temporary);
A_buf.tag_gpu_global();
B_buf.tag_gpu_global();
A_shr.tag_gpu_shared();

// Input matrix
input A({i, j}, p_float32);

// Shared memory operations
computation A_shr_alloc({j, i}, allocate(A_shr));
input A_shr_copy_access({j, i}, p_float32);
computation A_shr_copy({j, i}, A_shr_copy_access(j, i));
input A_shr_access({j, i}, p_float32);
computation sync({j, i}, tiramisu::sync());

// Output matrix
computation B({j, i}, A_shr_access(j, i));

// Scheduling commands
int block = 16;
B.gpu_tile(j, i, block, block, j0, i0, j1, i1);
A_shr_alloc.gpu_tile(j, i, block, block, j0, i0, j1, i1);
A_shr_copy.gpu_tile(j, i, block, block, j0, i0, j1, i1);
sync.gpu_tile(j, i, block, block, j0, i0, j1, i1);

// Memory layout commands
A_shr_copy_access.store_in(&A_buf,
```

```
        {i - i % 16 + j % 16, j - j % 16 + i % 16});
A_shr_copy.store_in(&A_shr, {j % 16, i % 16});
A_shr_access.store_in(&A_shr, {i % 16, j % 16});
A.store_in(&A_buf, {i, j});
B.store_in(&B_buf, {j, i});

copy_A_to_device.then(A_shr_alloc, computation::root)
                .then(A_shr_copy, i1)
                .then(sync, i1)
                .then(B, i1)
                .then(copy_B_to_host, computation::root);
```

Listing 4.4: The same transposition program with the new API

```
// Skipping the CPU-GPU copies
var i("i", 0, 128), j("j", 0, 256);
var i0("i0"), i1("i1");
var j0("j0"), j1("j1");

input A({i, j}, p_float32);
A.get_buffer()->tag_gpu_global();
computation B({j, i}, A(i, j));
B.get_buffer()->tag_gpu_global();

int block = 16;
B.gpu_tile(j, i, block, block, j0, i0, j1, i1);

B.cache_shared(A, i1, {16, 16}, {i0 * block, j0 * block});
```

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Conclusion

In this work we showed that Tiramisu is able to generate high efficiency code for GPU backend. Achieving state of the art performance especially in LSTM task, Tiramisu stands out as a strong candidate as a platform for implementing high performance applications for deep learning and other similar fields. We also presented several features implemented which create novel ways of optimization and improve the usability of Tiramisu.

As a next step, simplifying and automating the certain interfaces of Tiramisu would increase the portability of Tiramisu programs and ease of use of the framework. This includes `cache_shared` method, where user passes the buffer size and copy offsets manually. However, these arguments can be determined automatically by analyzing the iteration domain of the computation. The performance of `cache_shared` could also be improved by adding automated pipelining for copies and computations.

Currently Tiramisu functions can only take Halide CPU buffers as arguments. Which means to use a Tiramisu GPU GEMM function in another Tiramisu GPU program buffers should be copied to CPU before they can be passed between function. This creates a huge performance bottleneck and it is one of the reasons LSTM program is implemented using cuBLAS library calls.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

GEMM Implementation in Tiramisu

```
tiramisu::init("matmul");

// -------------------------------------------------------
// Layer I
// -------------------------------------------------------

// Declare loop iterators
var i("i", 0, M), j("j", 0, N), k("k", 0, K);
var i0("i0", 0, M / R_BLOCK_I), i1("i1", 0, R_BLOCK_I);
var j0("j0", 0, N / R_BLOCK_J), j1("j1", 0, R_BLOCK_J);
var k0("k0", 0, K / BLOCK), k1("k1", 0, BLOCK);
var k0_skiplast("k0", 0, K / BLOCK - 1);
var i00("i00"), i01("i01"), j00("j00"), j01("j01");

// Declare cpu buffers.
buffer b_A("b_A", {M, K}, p_float32, a_input);
buffer b_B("b_B", {K, N}, p_float32, a_input);
```

```
buffer b_C("b_C", {M, N}, p_float32, a_output);
buffer b_Consts("b_Consts", {2}, p_float32, a_input);
// Declare gpu buffers.
buffer b_A_glb("b_A_glb", {M, K}, p_float32, a_temporary);
buffer b_B_glb("b_B_glb", {K, N}, p_float32, a_temporary);
buffer b_C_glb("b_C_glb", {M, N}, p_float32, a_temporary);
// "+ 1" to reduce shared memory bank conflicts
buffer b_A_shr("b_A_shr", {2, BLOCK, BLOCK * R_BLOCK_I + 1},
               p_float32, a_temporary);
buffer b_B_shr("b_B_shr", {2, BLOCK, BLOCK * R_BLOCK_J},
               p_float32, a_temporary);
buffer b_A_reg("b_A_reg", {1}, p_float32, a_temporary);
buffer b_B_reg("b_B_reg", {R_BLOCK_J}, p_float32,
               a_temporary);
buffer b_acc("b_acc", {R_BLOCK_I, R_BLOCK_J}, p_float32,
             a_temporary);
b_A_glb.tag_gpu_global();
b_B_glb.tag_gpu_global();
b_C_glb.tag_gpu_global();
b_A_shr.tag_gpu_shared();
b_B_shr.tag_gpu_shared();
b_A_reg.tag_gpu_register();
b_B_reg.tag_gpu_local();
b_acc.tag_gpu_local();

// Declare input wrappers
input c_A_glb({i0, j0, k0, i1}, p_float32);
input c_A_shr({i0, j0, k0, k1, i1}, p_float32);
input c_A({i, k}, p_float32);
input c_B_glb({i0, j0, k0, j1}, p_float32);
```

```
input c_B_shr({i0, j0, k0, k1, j1}, p_float32);
input c_B({k, j}, p_float32);
input c_Consts({i}, p_float32);
constant c_alpha("alpha", c_Consts(0));
constant c_beta("beta", c_Consts(1));
// Declare computations
computation c_A_glb_to_shr_pre({i0, j0, i1},
    c_A_glb(i0, j0, 0, i1));
computation c_A_glb_to_shr({i0, j0, k0_skiplast, i1},
    c_A_glb(i0, j0, k0_skiplast + 1, i1));
computation c_A_shr_to_reg({i0, j0, k0, k1, i1},
    c_A_shr(i0, j0, k0, k1, i1));
computation c_B_glb_to_shr_pre({i0, j0, j1},
    c_B_glb(i0, j0, 0, j1));
computation c_B_glb_to_shr({i0, j0, k0_skiplast, j1},
    c_B_glb(i0, j0, k0_skiplast + 1, j1));
computation c_B_shr_to_reg({i0, j0, k0, k1, j1},
    c_B_shr(i0, j0, k0, k1, j1));
computation c_acc_init({i, j}, (float) 0);
computation c_acc({i, j, k}, p_float32);
c_acc.set_expression(c_acc(i, j, 0) + c_A(i, k) * c_B(k, j));
computation c_C({i, j}, p_float32);
c_C.set_expression(
    c_acc(i, j, 0) * c_alpha + c_C(i, j) * c_beta);
// Declare declarations
computation c_A_shr_dec({i0, j0}, allocate(b_A_shr));
computation c_A_reg_dec({i0, j0}, allocate(b_A_reg));
computation c_B_shr_dec({i0, j0}, allocate(b_B_shr));
computation c_B_reg_dec({i0, j0}, allocate(b_B_reg));
computation c_acc_dec({i0, j0}, allocate(b_acc));
```

```
// Declare synchronizer computations
computation c_sync1({i0, j0}, tiramisu::sync());
computation c_sync2({i0, j0, k0}, tiramisu::sync());
// Declare host-gpu transfer computations.
computation copy_A_to_device({}, memcpy(b_A, b_A_glb));
computation copy_B_to_device({}, memcpy(b_B, b_B_glb));
computation copy_C_to_device({}, memcpy(b_C, b_C_glb));
computation copy_C_to_host({}, memcpy(b_C_glb, b_C));

// -------------------------------------------------------
// Layer II
// -------------------------------------------------------

// Scheduling commands
c_acc_init.tile(i, j, R_BLOCK_I, R_BLOCK_J, i0, j0, i1, j1);
c_acc.tile(i, j, R_BLOCK_I, R_BLOCK_J, i0, j0, i1, j1);
c_acc.interchange(j1, k);
c_acc.interchange(i1, k);
c_acc.split(k, BLOCK, k0, k1);
c_C.tile(i, j, R_BLOCK_I, R_BLOCK_J, i0, j0, i1, j1);

block kernel_block({&c_acc_dec, &c_acc_init, &c_acc, &c_C,
    &c_A_shr_dec, &c_A_reg_dec, &c_B_shr_dec, &c_B_reg_dec,
    &c_A_glb_to_shr_pre, &c_A_glb_to_shr, &c_A_shr_to_reg,
    &c_B_glb_to_shr_pre, &c_B_glb_to_shr, &c_B_shr_to_reg,
    &c_sync1, &c_sync2});
kernel_block.gpu_tile(i0, j0, BLOCK, BLOCK,
                      i00, j00, i01, j01);

copy_A_to_device.then(copy_B_to_device, computation::root)
```

```
                .then(copy_C_to_device, computation::root)
                .then(c_A_shr_dec, computation::root)
                .then(c_B_shr_dec, j01)
                .then(c_A_reg_dec, j01)
                .then(c_B_reg_dec, j01)
                .then(c_acc_dec, j01)
                .then(c_acc_init, j01)
                .then(c_A_glb_to_shr_pre, j01)
                .then(c_B_glb_to_shr_pre, j01)
                .then(c_sync1, j01)
                .then(c_A_glb_to_shr, j01)
                .then(c_B_glb_to_shr, k0)
                .then(c_B_shr_to_reg, k0)
                .then(c_A_shr_to_reg, k1)
                .then(c_acc, i1)
                .then(c_sync2, k0)
                .then(c_C, j01)
                .then(copy_C_to_host, computation::root);

// ------------------------------------------------------------
// Layer III
// ------------------------------------------------------------

c_A_glb.store_in(&b_A_glb,
    {i0 * R_BLOCK_I + i1, k0 * BLOCK + j0 % BLOCK});
c_A_glb_to_shr_pre.store_in(&b_A_shr,
    {0, j0 % BLOCK, i0 % BLOCK * R_BLOCK_I + i1});
c_A_glb_to_shr.store_in(&b_A_shr,
    {(k0_skiplast + 1) % 2, j0 % BLOCK,
     i0 % BLOCK * R_BLOCK_I + i1});
```

```
c_A_shr.store_in(&b_A_shr,
    {k0 % 2, k1, i0 % BLOCK * R_BLOCK_I + i1});
c_A_shr_to_reg.store_in(&b_A_reg, {0});
c_B_glb.store_in(&b_B_glb, {k0 * BLOCK + i0 % BLOCK,
    (j0 - j0 % BLOCK) * R_BLOCK_J
    + j1 * BLOCK + j0 % BLOCK});
c_B_glb_to_shr_pre.store_in(&b_B_shr,
    {0, i0 % BLOCK, j1 * BLOCK + j0 % BLOCK});
c_B_glb_to_shr.store_in(&b_B_shr,
    {(k0_skiplast + 1) % 2,
     i0 % BLOCK, j1 * BLOCK + j0 % BLOCK});
c_B_shr.store_in(&b_B_shr,
    {k0 % 2, k1, j0 % BLOCK * R_BLOCK_J + j1});
c_B_shr_to_reg.store_in(&b_B_reg, {j1});
c_A.store_in(&b_A_reg, {i % R_BLOCK_I});
c_B.store_in(&b_B_reg, {j % R_BLOCK_J});
c_acc_init.store_in(&b_acc, {i % R_BLOCK_I, j % R_BLOCK_J});
c_acc.store_in(&b_acc, {i % R_BLOCK_I, j % R_BLOCK_J});
c_C.store_in(&b_C_glb);
c_Consts.store_in(&b_Consts, {i});

// ------------------------------------------------------------
// Code Generation
// ------------------------------------------------------------

// Generate object files.
tiramisu::codegen({&b_Consts, &b_A, &b_B, &b_C},
                  "fct.o", true);
```

Appendix B

LSTM Implementation in Tiramisu

```
tiramisu::init("lstm");

// -------------------------------------------------------------------
// Layer I
// -------------------------------------------------------------------

// Inner dimensions
var i("i", 0, FEATURE_SIZE), j("j", 0, FEATURE_SIZE);
var k("k", 0, BATCH_SIZE);
var i_merged("i_merged", 0, 4 * FEATURE_SIZE);
var i0("i0"), i1("i1"), k0("k0"), k1("k1");
// Outer dimensions
var l("l", 0, NUM_LAYERS), s("s", 0, SEQ_LENGTH);
var s0("s0", 0, SEQ_LENGTH / GEMM_BATCH);
var s1("s1", 0, GEMM_BATCH);
// After skewing
var l_s("l_s"), s_s("s_s");

// Input-output CPU buffers
buffer buf_Weights_cpu("buf_Weights_cpu",
```

```
    {NUM_LAYERS, 2, 4 * FEATURE_SIZE, FEATURE_SIZE},
    p_float32, a_input);
buffer buf_biases_cpu("buf_biases_cpu",
    {NUM_LAYERS, 4 * FEATURE_SIZE},
    p_float32, a_input);
buffer buf_x_cpu("buf_x_cpu",
    {SEQ_LENGTH, BATCH_SIZE, FEATURE_SIZE},
    p_float32, a_input);
buffer buf_y_cpu("buf_y_cpu",
    {SEQ_LENGTH, BATCH_SIZE, FEATURE_SIZE},
    p_float32, a_output);

// GPU buffers
input x("x", {s, k, i}, p_float32);
input weights("weights", {l, var("w_i", 0, 2), i_merged, j},
    p_float32);
input biases("biases", {l, i_merged}, p_float32);
input tmp("tmp", {s, k, i_merged}, p_float32);
x.get_buffer()->tag_gpu_global();
weights.get_buffer()->tag_gpu_global();
biases.get_buffer()->tag_gpu_global();
tmp.get_buffer()->tag_gpu_global();

// Transpose Weights
var w_t_i("w_i", 0, 2);  // Dummy variable
computation weights_T({l, w_t_i, j, i_merged},
    weights(l, w_t_i, i_merged, j));
weights_T.get_buffer()->tag_gpu_global();

// h(l, s) is the output of the block (l, s)
```

```
// which takes h(l, s - 1) and h(l - 1, s) as inputs
// Initial hidden states are h(l, -1) and c(l, -1)
// Input x is copied to h(-1, s)
computation h({l, s, k, i}, p_float32);
computation c({l, s, k, i}, p_float32);
// Pad buffers to make room for edges
h.store_in({l + 1, s + 1, k, i},
    {NUM_LAYERS + 1, SEQ_LENGTH + 1, BATCH_SIZE,
     FEATURE_SIZE});
c.store_in({l, s + 1, k, i},
    {NUM_LAYERS, SEQ_LENGTH + 1, BATCH_SIZE, FEATURE_SIZE});
h.get_buffer()->tag_gpu_global();
c.get_buffer()->tag_gpu_global();
// Initial sets and stores
computation h_init({l, k, i}, expr(DATA_TYPE(0)));
computation c_init({l, k, i}, expr(DATA_TYPE(0)));
computation h_copy_x({s, k, i}, x(s, k, i));
// Multiplication from input is batched
computation sum1({l, s0},
    cublas_gemm(*h.get_buffer(),
        *weights_T.get_buffer(),
        *tmp.get_buffer(),
        GEMM_BATCH * BATCH_SIZE,
        4 * FEATURE_SIZE,
        FEATURE_SIZE,
        1, 0,  // alpha, beta
        0, 0, 0,  // ldABC
        (l * (SEQ_LENGTH + 1) + s0 * GEMM_BATCH + 1)
        * BATCH_SIZE * FEATURE_SIZE,
        (l * 2) * 4 * FEATURE_SIZE * FEATURE_SIZE,
```

```
        s0 * GEMM_BATCH * BATCH_SIZE * 4 * FEATURE_SIZE,
        false, false));
computation sum2({l, s},
    cublas_gemm(*h.get_buffer(),
        *weights_T.get_buffer(),
        *tmp.get_buffer(),
        BATCH_SIZE, 4 * FEATURE_SIZE, FEATURE_SIZE,
        1, 1,  // alpha, beta
        0, 0, 0,  // ldABC
        ((l + 1) * (SEQ_LENGTH + 1) + s) * BATCH_SIZE
        * FEATURE_SIZE,
        (l * 2 + 1) * 4 * FEATURE_SIZE * FEATURE_SIZE,
        s * BATCH_SIZE * 4 * FEATURE_SIZE,
        false, false));
// Nonlinear operations as well as biases
#define sigmoid(x) expr(float(1)) / (1 + expr(o_expo, -(x)))
computation sig_i({l, s, k, i},
    sigmoid(tmp(s, k, i + 0 * FEATURE_SIZE)
    + biases(l, i + 0 * FEATURE_SIZE)));
computation sig_f({l, s, k, i},
    sigmoid(tmp(s, k, i + 1 * FEATURE_SIZE)
    + biases(l, i + 1 * FEATURE_SIZE)));
computation tnh_z({l, s, k, i},
    expr(o_tanh, tmp(s, k, i + 2 * FEATURE_SIZE)
    + biases(l, i + 2 * FEATURE_SIZE)));
computation sig_o({l, s, k, i},
    sigmoid(tmp(s, k, i + 3 * FEATURE_SIZE)
    + biases(l, i + 3 * FEATURE_SIZE)));
// Update cells
c.set_expression(sig_i(l, s, k, i) * tnh_z(l, s, k, i)
```

```
    + sig_f(l, s, k, i) * c(l, s - 1, k, i));
h.set_expression(
    expr(o_tanh, c(l, s, k, i)) * sig_o(l, s, k, i));
// Synchronize GEMMS and kernels before thread is destroyed
computation stream_sync({l, s0}, cuda_stream_synchronize());
// Output is the last layer
computation y({s, k, i}, h(NUM_LAYERS - 1, s, k, i));
y.get_buffer()->tag_gpu_global();
// Copies
computation copy_Weights_to_device({},
    memcpy(buf_Weights_cpu, *weights.get_buffer()));
computation copy_biases_to_device({},
    memcpy(buf_biases_cpu, *biases.get_buffer()));
computation copy_x_to_device({},
    memcpy(buf_x_cpu, *x.get_buffer()));
computation copy_y_to_host({},
    memcpy(*y.get_buffer(), buf_y_cpu));

// ------------------------------------------------------------
// Layer II
// ------------------------------------------------------------

// Fuse kernels by moving gpu iterators out
weights_T.interchange(w_t_i, j);
weights_T.interchange(w_t_i, i_merged);
weights_T.interchange(l, j);
weights_T.interchange(l, i_merged);
h_init.interchange(l, k);
h_init.interchange(l, i);
c_init.interchange(l, k);
```

```
c_init.interchange(l, i);
h_copy_x.interchange(s, k);
h_copy_x.interchange(s, i);
y.interchange(s, k);
y.interchange(s, i);

weights_T.gpu_tile(j, i_merged, 16, 16);
block nonlinear_block(
    {&sig_i, &tnh_z, &sig_o, &sig_f, &c, &h});
// Batch Input GEMMs
block({&sum2, &nonlinear_block})
    .split(s, GEMM_BATCH, s0, s1);
block({&h_init, &c_init, &h_copy_x, &nonlinear_block, &y})
    .gpu_tile(k, i, 16, 16, k0, i0, k1, i1);
block lstm_block(
    {&sum1, &sum2, &nonlinear_block, &stream_sync});
// Skew and interchange to get diagonal traversal
lstm_block.skew(l, s0, 1, l_s, s_s);
lstm_block.interchange(l_s, s_s);
// Parallelize diagonal traversal
// Due to a bug in tagging system we only need to parallelize
// a single computation
sum1.parallelize(l_s);

// Scheduling commands
copy_Weights_to_device
        .then(copy_biases_to_device, computation::root)
        .then(copy_x_to_device, computation::root)
        .then(weights_T, computation::root)
        .then(h_init, computation::root)
```

```
        .then(c_init, computation::root)
        .then(h_copy_x, computation::root)
        .then(sum1, computation::root)
        .then(sum2, l_s)
        .then(sig_i, s1)
        .then(sig_f, i1)
        .then(tnh_z, i1)
        .then(sig_o, i1)
        .then(c, i1)
        .then(h, i1)
        .then(stream_sync, l_s)
        .then(y, computation::root)
        .then(copy_y_to_host, computation::root);

// ------------------------------------------------------------
// Layer III
// ------------------------------------------------------------

sig_i.store_in(tmp.get_buffer(),
    {s, k, i + 0 * FEATURE_SIZE});
sig_f.store_in(tmp.get_buffer(),
    {s, k, i + 1 * FEATURE_SIZE});
tnh_z.store_in(tmp.get_buffer(),
    {s, k, i + 2 * FEATURE_SIZE});
sig_o.store_in(tmp.get_buffer(),
    {s, k, i + 3 * FEATURE_SIZE});
h_init.store_in(h.get_buffer(), {l + 1, 0, k, i});
c_init.store_in(c.get_buffer(), {l, 0, k, i});
h_copy_x.store_in(h.get_buffer(), {0, s + 1, k, i});
```

```
// ------------------------------------------------------------------
// Code Generation
// ------------------------------------------------------------------

// Generate object files.
tiramisu::codegen({
        &buf_Weights_cpu,
        &buf_biases_cpu,
        &buf_x_cpu,
        &buf_y_cpu,
    }, "lstm.o", true);
```

Bibliography

[1] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. *arXiv e-prints*, page arXiv:1604.01946, Apr 2016.

[2] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, Oct 2015.

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 193–205, Piscataway, NJ, USA, 2019. IEEE Press.

[4] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: DSL for linear algebra and neural net computations on gpus. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 42–51, New York, NY, USA, 2018. ACM.

[5] Nvidia. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Accessed: 2019-05-20.

[6] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *arXiv e-prints*, page arXiv:1802.04730, February 2018.

[7] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[9] Pete Warden. Why gemm is at the heart of deep learning. https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/. Accessed: 2019-05-20.