Tiramisu: A Polyhedral Compiler with A Scheduling Language for Targeting High Performance Systems

Riyadh Baghdadi
MIT
baghdadi@mit.edu

Jessica Ray
MIT
jray@mit.edu

Malek Ben Romdhane
MIT
malek@mit.edu

Emanuele Del Sozzo
Politecnico di Milano
emanuele.delsozzo@polimi.it

Patricia Suriana
Google
psuriana@google.com

Abdurrahman Akkas
MIT
akkas@mit.edu

Shoaib Kamil
Adobe
kamil@adobe.com

Yunming Zhang
MIT
yunming@mit.edu

Saman Amarasinghe
MIT
saman@mit.edu

Abstract

This paper introduces Tiramisu, a polyhedral framework designed to generate high performance code for multiple platforms including multicores, GPUs, and distributed machines. Tiramisu introduces a scheduling language with novel extensions to explicitly manage the complexities that arise when targeting these systems. The extensions include explicit communication, synchronization, and mapping buffers to different memory hierarchies. Tiramisu relies on a flexible representation based on the polyhedral model and explicitly uses a well-defined four-level IR that allows full separation between the algorithms, loop transformations, data-layouts, and communication. This separation simplifies targeting multiple hardware architectures with the same algorithm. We evaluate Tiramisu by writing a set of image processing and stencil benchmarks and compare it with state-of-the-art compilers. We show that Tiramisu matches or outperforms existing compilers on different hardware architectures, including multicore CPUs, GPUs, and distributed machines.

1 Introduction

Generating efficient code for high performance systems is becoming more and more difficult as these architectures are increasing in complexity and diversity. Obtaining the best performance requires complex code and data layout transformations, management of complex memory hierarchies, and efficient data communication and synchronization.

For example, consider generalized matrix multiplication (gemm), which computes $C = \alpha AB + \beta C$ and is a building block of numerous algorithms, including simulations and convolutional neural networks. Highly-tuned implementations require fusing the multiplication and addition loops, as well as applying two-level tiling, vectorization, loop unrolling, array packing [15], register blocking, and data prefetching. Furthermore, tuned implementations separate partial tiles from full tiles, since partial tiles cannot fully benefit from the same optimizations. High performance GPU implementations require even more optimizations. Memory accesses should be coalesced, and data movement must be managed between global memory, shared memory, and registers, with synchronization primitives inserted where necessary. Automatically generating such complex code is still beyond the capabilities of state-of-the-art compilers. The importance of kernels such as gemm, along with the immense complexity of optimized implementations, motivate vendors to release highly hand-optimized libraries for such kernels. However, for most users, obtaining this level of performance for their own code is challenging, since the effort required to explore the space of possible implementations is intractable when hand-coding complex code transformations.

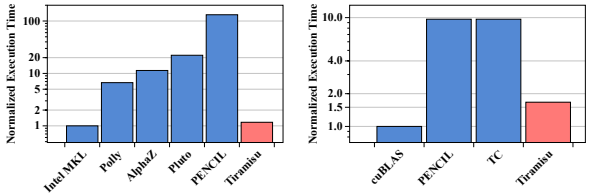

Figure 1. Normalized execution times of code generated for sgemm on CPU (left) and GPU (right).

Previous work using the polyhedral model has shown success in implementing complex iteration space transformations [8, 17, 33, 40, 42, 44], data locality optimizations [16, 23], and memory management optimizations [10, 12, 25, 34, 39]. Although polyhedral compilers are able to represent these program and data transformations, they are still not successful in selecting transformations for the best performance. They still do not match the performance of highly hand-optimized kernels such as gemm. Blue bars in Figure 1 show the performance of state-of-the-art polyhedral compilers for gemm compared to the Intel MKL [22] and Nvidia cuBLAS [31] libraries. Fully-automatic polyhedral compilers such as Polly [17], Pluto [8], and PENCIL [2, 3] improve

CGO'19, February 16–20, 2019, Washington DC, USA
2019.

productivity, but fail to obtain the desired level of performance since their search techniques consider only a subset of the necessary optimizations and they rely on less accurate machine models, leading the compiler to make suboptimal decisions. Other polyhedral frameworks, such as AlphaZ [46] and CHiLL [9], eschew full automation and instead expose a *scheduling language* that enables users to productively explore the space of possible transformations. While these frameworks achieve better performance, their scheduling languages are not designed to target GPUs and distributed systems. For example, they do not allow the user to partition computations, send data across nodes, map buffers to GPU shared or local memory, or insert required synchronization.

In this paper, we introduce Tiramisu, a polyhedral compiler with a scheduling language featuring *novel extensions for targeting multiple high performance architectures.* Tiramisu is well suited for implementing data parallel algorithms (loop nests manipulating arrays). It takes a high level representation of the program (pure algorithm and a set of scheduling commands), applies the necessary code transformations, and generates highly-optimized code for the target architecture. In addition to scheduling commands for loop and data-layout transformations, the Tiramisu scheduling language introduces novel commands for explicit communication and synchronization, and for mapping buffers to different memory hierarchies. In order to simplify the implementation of the scheduling language, Tiramisu explicitly divides the intermediate representation into four layers designed to hide the complexity and large variety of execution platforms by separating the architecture-independent algorithm from the code transformations, data-layout, and communication. Tiramisu targets multicore CPUs, GPUs (CUDA), distributed architectures and FPGA. This paper presents the first three backends while Sozzo et al. [38] present an FPGA backend.

The use of a scheduling language has been shown to be effective for generating efficient code by multiple compilers including CHiLL, AlphaZ, and Halide [35, 36]. Halide is the most successful example as it is now used in production by multiple companies. In comparison with Halide in particular, in addition to the introduction of novel scheduling extensions, Tiramisu also fundamentally differs from Halide in that it relies on the expressive polyhedral representation instead of the interval-based representation used by Halide. This allows Tiramisu to express non-rectangular iteration spaces, to support programs with cyclic data-flow graphs, and to apply any affine transformation (including iteration space skewing), all of which are not possible in Halide.

This paper makes the following contributions:

- We introduce a polyhedral compiler with a scheduling language that features *novel extensions for controlling data communication, synchronization, and for mapping to different memory hierarchies.* These extensions enable targeting multiple high-performance architectures including multicore CPUs, GPUs, and distributed machines.
- We explicitly divide the intermediate representation into four layers to simplify the implementation of the scheduling language. The four-layer IR separates the algorithm from code transformations and data-layout transformations, allowing for portability and simplifying the composition of architecture-specific lowering transformations.
- We evaluate Tiramisu on a set of image processing and stencil benchmarks and compare it with Halide, an industrial compiler with a scheduling language, and with PENCIL, a state-of-the-art polyhedral compiler. We show that Tiramisu matches or outperforms existing compilers on different hardware architectures, including multicore CPUs, GPUs, and distributed machines.

2 Related Work

Polyhedral compilers with automatic scheduling. Polyhedral compilers such as PENCIL [3, 4], Pluto [8], Polly [17], Tensor Comprehensions [42], and PolyMage [30] are fully automatic. Some of them are designed for specific domains (such as Tensor Comprehensions and PolyMage), while Pluto, PENCIL, and Polly are more general. While such fully automatic compilers provide productivity, they may not always obtain the best performance. This is due to many reasons: first, these compilers do not implement some key optimizations such as array packing [15], register blocking, data prefetching, and asynchronous communication (which are all supported by Tiramisu); second, they do not have a precise cost-model to decide which optimizations are profitable. For example, the Pluto [8] automatic scheduling algorithm (which is used for automatic scheduling in Pluto, PENCIL, Polly, and Tensor Comprehensions) tries to minimize the distance between producer and consumer statements while maximizing outermost parallelism, but it does not consider the data layout, redundant computations, or the complexity of the control of the generated code. PolyMage uses a custom scheduling algorithm designed for image processing code, but such an algorithm does not address all the complexities that arise when targeting GPUs and distributed systems. Instead of fully automatic scheduling, Tiramisu uses a more pragmatic approach and relies on a set of scheduling commands, giving the user full control over scheduling.

Polyhedral frameworks proposed by Amarasinghe and Lam [1] and Bondhugula [7] address the problem of code generation for distributed systems. Tiramisu makes a different design choice, relying on the user to provide scheduling commands to control choices in the generated code (synchronous/asynchronous communication, the granularity of communication, buffer sizes, when to send and receive, the cost of communication versus re-computation, etc.).

Even though Tiramisu focuses on providing mechanisms for code optimization, it is still possible to build a framework that provides policy on top of Tiramisu (i.e., a framework

Feature	Tiramisu	AlphaZ	PENCIL	Pluto	Halide
CPU code generation	Yes	Yes	Yes	Yes	Yes
GPU code generation	Yes	No	Yes	Yes	Yes
Distributed CPU code generation	Yes	No	No	Yes	Yes
Distributed GPU code generation	Yes	No	No	No	No
Support all affine loop transformations	Yes	Yes	Yes	Yes	No
Optimize data accesses	Yes	Yes	No	No	Yes
Commands for loop transformations	Yes	Yes	No	No	Yes
Commands for optimizing data accesses	Yes	Yes	No	No	Yes
Commands for communication	Yes	No	No	No	No
Commands for memory hierarchies	Yes	No	No	No	Limited
Expressing cyclic data-flow graphs	Yes	Yes	Yes	Yes	No
Support non-rectangular iteration spaces	Yes	Yes	Yes	Yes	Limited
Instance-wise, exact dependence analysis	Yes	Yes	Yes	Yes	No
Compile-time affine-set emptiness check	Yes	Yes	Yes	Yes	No
Implement support for parametric tiling	No	Yes	No	No	Yes

Table 1. Comparison between different frameworks.

that automates scheduling). This separation provides flexibility, making it possible to plug in different policies.

Polyhedral compilers with a scheduling language. AlphaZ [46], CHiLL [9, 19], URUK [14], and Transformation Recipes [20] are polyhedral frameworks and methods developed to allow users to express high-level transformations using scheduling commands. Since these frameworks are polyhedral, they can express any affine transformation. Their scheduling languages do not target distributed architectures, however, nor do they target GPUs (except the Transformation Recipes framework). In comparison with these frameworks, Tiramisu features scheduling commands for partitioning computations (for distributed systems), synchronization, distribution of data across nodes, mapping data to specific GPU memory hierarchy levels, and performing array packing. The first four columns of Table 1 compares between Tiramisu and three representative polyhedral frameworks.

Non-polyhedral compilers with a scheduling language. Halide [35] is an image processing DSL that has a scheduling language; however, it uses intervals to represent iteration spaces instead of the polyhedral model. This limits the expressiveness of Halide. For example, unlike Tiramisu, Halide cannot naturally represent non-rectangular iteration spaces. This is the reason why Halide distributed [11] overapproximates the amount of data to communicate (send and receive) when generating distributed code. This also makes certain Halide passes over-approximate non-rectangular iteration spaces, potentially leading to less efficient code (for example, it prevents Halide from performing precise bounds inference for non-rectangular iteration spaces). It also prevents Halide from performing many complex affine transformations, such as iteration space skewing.

Halide does not have dependence analysis and thus it relies on conservative rules to determine whether a schedule is legal; for example, Halide does not allow the fusion of two loops (using the `compute_with` command) if the second loop reads a value produced by the first loop. While this rule avoids illegal fusion, it prevents fusing many legal common cases which may lead to suboptimal performance. Halide also assumes the program has an acyclic dataflow graph in order to simplify checking the legality of a schedule. This prevents users from expressing many programs with cyclic dataflow. It is possible in some cases to work around the above restrictions, but such methods are not general. Tiramisu avoids over-conservative constraints by relying on dependence analysis to check for the correctness of code transformations, enabling more possible schedules. Table 1 shows a comparison between Tiramisu and Halide.

```
1 // Declare the iterators i, j and c.
2 var i(0, N-2), j(0, M-2), c(0, 3);
3
4 // Algorithm.
5 bx(i,j,c) = (in(i,j,c)+in(i,j+1,c)+in(i,j+2,c))/3;
6 by(i,j,c) = (bx(i,j,c)+bx(i+1,j,c)+bx(i+2,j,c))/3);
```

Figure 2. `Blur` algorithm without scheduling commands.

Other systems include POET [45], which uses an XML-based description of code and transformation behavior to parametrize loop transformations. It uses a purely-syntactic transformation system to produce transformed code. The use of syntactic transformations is less general than the polyhedral model used in Tiramisu.

3 The Tiramisu Embedded DSL

Tiramisu is a domain-specific language (DSL) embedded in C++. It provides a pure C++ API that allows users to write a high level, architecture-independent algorithm and a set of scheduling commands that guide code generation. The input Tiramisu code can either be written directly by a programmer, or generated by a different DSL compiler. Tiramisu then constructs a high level intermediate representation (IR), applies the user-specified loop and data-layout transformations, and generates optimized backend code that takes advantage of target hardware features (LLVM IR for multicores and distributed machines and LLVM IR + CUDA for GPUs).

Scope Tiramisu is designed for expressing data parallel algorithms, especially those that operate over dense arrays using loop nests and sequences of statements. These algorithms are often found in the areas of dense linear algebra and tensor algebra, stencil computations, image processing, and deep neural networks.

3.1 Specifying the Algorithm

The first part of a Tiramisu program specifies the algorithm without specifying loop optimizations (when and where the computations occur), data-layout (how data should be stored in memory), or communication. At this level there is no notion of data location; rather, values are communicated via explicit producer-consumer relationships.

The algorithm is a pure function that has inputs, outputs, and a sequence of statements. The statements are called *computations* in Tiramisu. Flow-control around these computations is restricted to `for` loops and conditionals; `while` loops, early exits and `GOTO`s cannot be expressed. To declare a computation, the user provides both the iteration domain of the computation and the expression to compute.

Figure 2 shows a blur algorithm written in Tiramisu, using bx and by, which are the two computations in this algorithm. The first computation, bx, computes a horizontal blur of the input, while the second computation, by, computes the final blur by averaging the output of the first stage. The iterators i, j and c in line 2 define the iteration domain of bx and by (for brevity we ignore boundary conditions). The algorithm is semantically equivalent to the following code.

```
1 for (i in 0..N-2)
2  for (j in 0..M-2)
3   for (c in 0..3)
4    bx[i][j][c] = (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
5 for (i in 0..N-2)
6  for (j in 0..M-2)
7   for (c in 0..3)
8    by[i][j][c] = (bx[i][j][c]+bx[i+1][j][c]+bx[i+2][j][c])/3
```

3.2 Scheduling Commands

Tiramisu provides a set of high-level scheduling commands for common optimizations. Table 2 shows examples of these commands. There are four types of scheduling commands:

- Commands for loop nest transformations: they include common affine transformations such as loop tiling, splitting, shifting, etc. For example, applying a 32×32 loop tiling on a computation C can be done by calling `C.tile(i,j,32,32,i0,j0,i1,j1)` where i and j are the original loop iterators and i0, j0, i1, and j1 are the names of the loop iterators after tiling.
- Commands for mapping loop levels to hardware. Examples of these include loop parallelization, vectorization, and mapping loop levels to a GPU block or thread dimension. For example calling `C.vectorize(j, 4)` splits the j loop by a factor of 4 and maps the inner loop to vector instructions.
- Commands for manipulating data: these include (1) allocating arrays; (2) setting array properties including whether the array is stored in host, device, shared, or local memory (GPU); (3) copying data (between levels of memory hierarchies or between nodes); and (4) setting array accesses. In most cases, users need only to use high level commands for data manipulation. If the high level commands are not expressive enough, the user can use the more expressive low level commands.
- Commands for adding synchronization operations: the user can either declare a barrier or use the send and receive functions for point-to-point synchronization.

The novel commands that Tiramisu introduces are highlighted in **bold** in Table 2. They include array allocation, copying data between memory hierarchies, sending and receiving data between nodes, and synchronization. Calls to `cache_shared_at()`, `cache_local_at()`, `allocate_at()`, `copy_at()`, `barrier_at()` return an operation that can be scheduled like any other computation (an operation in Tiramisu is a special type of computation that does not return any value). The operations `cache_shared_at()` and `cache_local_at()` can be used to create a cache for a buffer (GPU only). They automatically compute the amount of data that needs to be cached, perform the data copy, and insert any necessary synchronization.

The use of `allocate_at()`, `copy_at()`, and `barrier_at()` allows Tiramisu to automatically compute iteration domains for the data copy, allocation, and synchronization operations. This is important because it relieves the user from guessing or computing the iteration domain manually, especially when exploring different possible schedules. For example, consider the example of copying a buffer from global memory to shared memory in a loop nest executing on a GPU. The size of the area to copy and the iteration domain of the copy operation itself (which is a simple assignment in this case) depends on whether the loop is tiled, the tile size, and whether any other loop transformation has already been applied. Computing the area to copy in this case is non-trivial.

We assume that C and P are computations, b is a buffer, i and j are loop iterators	
Commands for loop nest transformations	
Command	**Description**
`C.tile( i,j,t1,t2, i0,j0,i1,j1)`	Tile the dimensions (i,j) of the computation C by $t1 \times t2$. The names of the new dimensions are (i0, j0, i1, j1), where (i0, j0) are the outer tiles and (i1, j1) are the inner tiles.
`C.interchange(i, j)`	Interchange the dimensions of C (loop interchange)
`C.shift(i, s)`	Loop shifting (shift the dimension i by s iterations)
`C.split(i, s, i0, i1)`	Split the dimension i by s. (i0, i1) are the new dimensions
`P.compute_at(C, j)`	Compute the computation *P* in the loop nest of *C* at loop level j. This might introduce redundant computations.
`C.unroll(i, v)`	Unroll the dimension i by a factor v
`C.after(B, i)`	Indicate that C should be ordered after B at the loop level i (they have the same order in all the loop levels above i)
`C.inline()`	Inline C in all of its consumers
`C.set_schedule()`	Transform the iteration domain of C using an affine relation (a *map* to transform Layer I to II expressed in the ISL syntax)
Commands for mapping loop levels to hardware	
`C.parallelize(i)`	Mark the dimension i as a space dimension (cpu)
`C.vectorize(i, v)`	Vectorize the dimension i by a vector size v
`C.gpu(i0, i1, i2)`	Mark the dimensions i0, i1 and i2 to be executed on the GPU
`C.tile_gpu(i0,i1,t1,t2)`	Tile the loops i0 and i1 by $t1 \times t2$ and map them to GPU
`C.distribute(i)`	Mark the dimension i as a space dimension (node)
High level commands for data manipulation	
`C.store_in(b,{i, j})`	Store the result of the computation C(i,j) in b[i,j].
`C.cache_shared_at(C,i)`	Cache (copy) the buffer of C in shared memory. The amount of data to copy, the access functions, and synchronization are computed automatically.
`C.cache_local_at(C, i)`	Similar to `cache_shared_at` but stores in local GPU memory.
`send(d, src, s, q, p)`	Create a send operation. d: vector of iterators to represent the iteration domain; src: source buffer; s: size; q: destination node; p: properties (synchronous/asynchronous, blocking, ...).
`receive(d,dst,s,q,p)`	Create a receive operation. Arguments similar to send except q which is the source node.
Low level commands for data manipulation	
`Buffer b(sizes, type)`	Declare a buffer (sizes: a vector of dimension sizes).
`b.allocate_at(p, i)`	Return an operation that allocate b at the loop i of p. An operation can be scheduled like a computation.
`C.buffer()`	Return the buffer associated to the computation C.
`b.set_size(sizes)`	Set the size of a buffer. sizes: a vector of dimension sizes.
`b.tag_gpu_global()`	Tag this buffer to be stored in global GPU memory.
`b.tag_gpu_shared()`	Tag this buffer to be stored in shared GPU memory.
`b.tag_gpu_local()`	Tag this buffer to be stored in local GPU memory.
`b.tag_gpu_constant()`	Tag this buffer to be stored in constant GPU memory.
`C.host_to_device()`	Return an operation that copies C.buffer() from host to device.
`C.device_to_host()`	Return an operation that copies C.buffer() from device to host.
`copy_at(p, i, bs, bd)`	Return an operation that copies the buffer bs to the buffer bd at the loop i of p. Used for copies between global, shared and local.
Commands for synchronization	
`barrier_at(p, i)`	Create a barrier at the loop p of i.

Table 2. Examples of Tiramisu Scheduling Commands

(a)

Tiramisu Scheduling Commands

```
// Scheduling commands for targeting GPU.
// Tile i and j and map the resulting dimensions to GPU
var i0, j0, i1, j1;
by.tile_gpu(i, j, 32, 32, i0, j0, i1, j1);
bx.compute_at(by, j0);
bx.cache_shared_at(by, j0);

// Use struct-of-array data layout for bx and by.
bx.store_in({c,i,j}); by.store_in({c,i,j});

// Create data copy operations
operation cp1 = in.host_to_device();
operation cp2 = by.device_to_host();

// Specify the order of execution of copies
cp1.before(bx, root); cp2.after(by, root);
```

Pseudocode Representing Code Generated by Tiramisu

```
host_to_device_copy(in_host, in);

GPUBlock for(i0 in 0..floor((N-2)/32))
 GPUBlock for(j0 in 0..floor((M-2)/32))
  shared bx[3,32,34];
  GPUThread for(i1 in 0..min(N-2,32+2)) // Tiling with redundancy
   GPUThread for(j1 in 0..min(M-2,32+2)) // Tiling with redundancy
    int i = i0*32+i1
    int j = j0*32+j1
    for (c in 0..3)
     bx[c][i1][j1]=
       (in[i][j][c]+in[i][j+1][c]+in[i1][j+2][c])/3
  GPUThread for(i1 in 0..min(N-2,32*i0+31))
   GPUThread for(j1 in 0..min(M-2,32*j0+31))
    for (c in 0..3)
     by[c][i][j]=(bx[c][i][j]+bx[c][i+1][j]+bx[c][i+2][j])/3

device_to_host_copy(by, by_host);
```

(b)

Tiramisu Scheduling Commands

```
// Scheduling commands for targeting a distributed system

// Declare additional iterators
var qs(1,Ranks), qr(0,Ranks-1), q, z;

// Split loop i into loops q and z and parallelize z
bx.split(i,N/Ranks,q,z); bx.parallelize(z);
by.split(i,N/Ranks,q,z); by.parallelize(z);

// Create communication and order execution
send s = send({qs}, in(0,0,0), N*2*3, qs-1, {ASYNC});
recv r = receive({qr}, in(0,N,0), N*2*3, qr+1,{SYNC},s);
s.before(r,root); r.before(bx,root)

// Distribute the outermost loops
bx.distribute(q); by.distribute(q);
s.distribute(qs); r.distribute(qr);
```

Pseudocode Representing Code Generated by Tiramisu

```
// The array in[][][] is initially distributed across nodes
// Exchange ghost zones of in[][][]
distributed for (qs in 1..Ranks)
  send(in(0,0,0), N*2*3, qs-1,{ASYNC})
distributed for (qr in 0..Ranks-1)
  recv(in(0,N,0), N*2*3, qr+1, {SYNC})

distributed for (q in 0..Ranks)
  parallel for (z in 0..N/Ranks)
   int i = q*(N/Ranks) + z
   for (j in 0..M)
     for (c in 0..3)
       bx[i][j][c] = (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
distributed for (q in 0..Ranks)
  parallel for (z in 0..N/Ranks)
   int i = q*(N/Ranks) + z
   for (j in 0..M)
     for (c in 0..3)
       by[i][j][c] = (bx[i][j][c]+bx[i+1][j][c]+bx[i+2][j][c])/3
// We assume that no gather operation on by[][][] is needed
```

Figure 3. Two examples illustrating Tiramisu scheduling commands (left) and the corresponding generated code (right). (a) shows scheduling commands for mapping to GPU; (b) uses commands to map to a distributed CPU machine.

Tiramisu simplifies this step by computing the iteration domain and the area of data to copy from the schedule.

To illustrate more Tiramisu scheduling commands, let us take the `blur` example again from Figure 2, and map the two outermost loops of `bx` and `by` to GPU. The necessary scheduling commands are shown in Figure 3-(a) (left). The `tile_gpu()` command tiles the computations then maps the new loops to GPU block and thread dimensions. The `compute_at()` command computes the tiles of `bx` that need to be read by `by`. This transformation introduces redundant computations (in this case) and is known as overlapped tiling [24]. `cache_shared_at()` instructs Tiramisu to store the results of the `bx` computation in shared memory. The subsequent scheduling command (`store_in()`) specifies the access functions of `bx` and `by`. In this case, it indicates that these computations are stored in a SOA (struct-of-array) data layout (to allow for coalesced accesses). The final commands create data copy operations (host-to-device and device-to-host) and schedule them.

Suppose now that we want to distribute the `blur` example and run it on a distributed system with multicore CPU nodes. Figure 3-(b) (left) shows the scheduling commands to use in this case. We assume that the array `in[][][]` is initially distributed across nodes such that the node with rank R has the chunk `in[R*(N/Ranks)...(R+1)*(N/Ranks), *, *]`. The `split()` command splits the outer loop, `i`, of `bx` by a splitting factor of `N/Ranks`. It creates two new loops `q` and `z` such that the outer loop, `q`, iterates over the `Ranks` (i.e. the number of processes we want to distribute). The `q` loop iterator will be mapped later to an MPI `rank`. We also parallelize the new inner loop, `z`, and perform the same transformations on `by`.

`create_send()` and `create_recv()` define communication, which in this case, sends $N \times 2 \times 3$ contiguous data elements between nodes starting from `in(0,0,0)`. The receiving node write the sent data starting from `in(0,N,0)`. Note that the buffer `in` in this case is local to one node. `qs` and `qr` represent the iteration domains of th send and receive operations. `qs-1` and `qr+1` represent the send's destination node and the receive's source node, respectively. `{ASYNC}` defines an asynchronous send and `{SYNC}` defines a synchronous receive. In this example, we explicitly specify the send and receive operations for illustrative reasons, instead of relying on Tiramisu to automatically insert them. Explicit send and receive operations are only useful if the array accesses or the iteration domain are not affine (more details

about non-affine code in Sec.5.2) because in such cases automatically computing the amount of data to exchange may not be accurate (the experimental section shows such cases). Finally, we tag the appropriate loops (the outer loops of bx, by, s, and r), to be distributed (i.e., we tag each iteration to be run on a different node).

All the other scheduling commands in Tiramisu can be composed with transfers and distributed loops, as long as the composition is semantically correct.

4 The Tiramisu IR

The main goal of Tiramisu's multi-layer intermediate representation is to simplify the implementation of scheduling commands by applying them in a specific order. This section illustrates why, and describes the layers of the Tiramisu IR.

4.1 Why a Multi-layer IR ?

Most intermediate representations use memory to communicate between program statements. This creates memory-based dependencies in the program, and forces data-layout to be chosen before deciding how the code is optimized and mapped to hardware. Optimizing a program for different hardware architectures usually requires modifying the data-layout and eliminating memory-based dependencies since they restrict optimization [27]. Thus, any data-layout specified before scheduling must be undone to allow more freedom for scheduling, and the code must be adapted to use the data-layout best-suited for the target hardware. Applying these data-layout transformations and the elimination of memory-based dependencies is challenging [10, 12, 18, 25, 26, 28, 29, 34, 41].

Another example that makes code generation complicated is mapping buffers to shared and local memory on GPU. The amount of data that needs to be copied to shared memory and when to perform synchronization all depend on how the code is optimized (for example, whether the code has two-level tiling or not). The same applies to deciding the amount of data to send or receive when generating distributed code. Therefore, buffer mapping to memory hierarchies, communication management, and synchronization should not be done before scheduling.

Tiramisu addresses these challenges by using a multi-layer IR that fully separates the architecture-independent algorithm from loop transformations, data-layout and communication. The top layer representation describes the pure algorithm using producer-consumer relationships without memory locations. The second layer specifies the order of the computations, along with which processor computes each value; this layer is suitable for performing a vast number of optimizations without dealing with concrete memory layouts. The third layer specifies where to store intermediate data before they are consumed. The fourth layer adds all the necessary communication and synchronization operations.

The separation of layers defines a specific order in which optimizations are applied and ensures that compiler passes

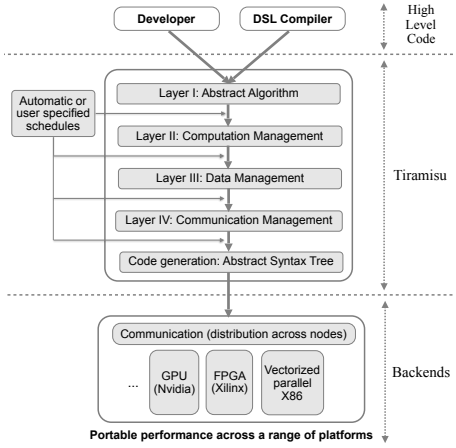

Figure 4. Tiramisu overview

in a given layer need not to worry about modifying or undoing a decision made in an earlier layer. For example, the phase that specifies the order of computations and where they occur can safely assume that no data-layout transformations are required, greatly simplifying the phase. This simple assumption allows Tiramisu to avoid the need to rely on a large body of research that focuses on data-layout transformations to allow scheduling [10, 12, 18, 25, 26, 28, 29, 34, 41].

4.2 Background

In this section, we provide an overview of two main concepts used in the polyhedral model: *integer sets* and *maps*. These two concepts will be used in later sections to define the different IR layers.

Integer sets represent iteration domains while *maps* are used to represent memory accesses and transform iteration domains and memory accesses (apply loop nest and memory access transformations). More details and formal definitions for these concepts are provided in [2, 32, 43].

An *integer set* is a set of integer tuples described using affine constraints. An example of a set of integer tuples is

$$\{(1, 1); (2, 1); (3, 1); (1, 2); (2, 2); (3, 2)\}$$

Instead of listing all tuples in a set, we describe the set using affine constraints over loop iterators and symbolic constants:

$$\{S(i, j) : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\}$$

where i and j are the dimensions of tuples in the set.

A *map* is a relation between two integer sets. For example

$$\{S1(i, j) \rightarrow S2(i + 2, j + 2) : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\}$$

is a map between tuples in the set S1 and tuples in the set S2 (e.g. the tuple $S1(i, j)$ maps to the tuple $S2(i + 2, j + 2)$).

All sets and maps in Tiramisu are implemented using the Integer Set Library (ISL) [43]. We also use the ISL library notation for sets and maps throughout the paper.

4.3 The Multi-Layer IR

A typical workflow for using Tiramisu is illustrated in Figure 4. The user writes the first layer of Tiramisu (which is the pure algorithm) and then provides a set of scheduling commands. The first layer of the IR is then transformed to

lower layers, and finally TIRAMISU generates LLVM or other appropriate low-level IR. TIRAMISU uses integer sets to represent each of the four IR layers and uses maps to represent transformations on the iteration domain and data-layout. The remainder of this section describes the four layers of the TIRAMISU IR.

4.3.1 Layer I (Abstract Algorithm)

Layer I of TIRAMISU specifies the algorithm without specifying when and where the computations occur or how data should be stored in memory (data-layout) or communication. As this level has no notion of data location, values are communicated via explicit producer-consumer relationships.

Let us take the example of the code in Figure 2 and let us only consider the computation by for simplicity. It is represented in Layer I as follows.

$\{by(i, j, c) : 0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3\} : (bx(i, j, c) + bx(i+1, j, c) + bx(i+2, j, c))/3$

The first part of this line, $\{by(i, j, c) : 0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3\}$, specifies the iteration domain of the statement, while the second part is the computed expression. The iteration domain is the set of tuples $by(i, j, c)$ such that $0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3$. Computations in Layer I are not ordered; declaration order does not affect the order of execution, which is specified in Layer II.

4.3.2 Layer II (Computation Management)

Layer II of TIRAMISU specifies the order of execution of computations and the processor on which they execute. This layer does not specify how intermediate values are stored in memory; this simplifies optimization passes since these transformations do not need to perform complicated data-layout transformations. The transformation of Layer I into Layer II is done automatically using scheduling commands.

Figure 3-(a) (right) shows the first optimized version of the code, produced by the set of scheduling and data-layout commands on the left side. The corresponding Layer II representation for the by computation is shown below:

$\{by(1, i0(gpuB), j0(gpuB), i1(gpuT), j1(gpuT), c) : i0 = floor(i/32) \wedge j0 = floor(j/32) \wedge i1 = i\%32 \wedge j1 = j\%32 \wedge 0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3\} : (bx(i0*32+i1, j0*32+j1, c) + bx(i0*32+i1+1, j0*32+j1, c) + bx(i0*32+i1+2, j0*32+j1, c))/3$

Computations in Layer II are ordered based on their lexicographical order[1]. The set (purple text in the previous example), is an ordered set of computations. The tag *gpuB* for the dimension $i0$ and $j0$ indicates that each iteration $(i0, j0)$ is mapped to the GPU block $(i0, j0)$. In Layer II, the total ordering of these tuples determines the execution order.

Unlike the first layer, computations in this layer are ordered and assigned to a particular processor; we know when and where they will run. This order is dictated by *time dimensions* and *space dimensions*. Time dimensions specify the order of execution relative to other computations while space dimensions specify on which processor each computation executes. The ordering of the time dimensions determines the execution order of each computation. Space dimensions only indicate where computations run, and are distinguished from time dimensions using tags, which consist of a processor type followed by zero or more properties. Currently, TIRAMISU supports the following space tags:

`cpu`	the dimension runs on a CPU in a shared memory system
`node`	the dimension maps to nodes in a distributed system
`gpuT`	the dimension maps to a gpu thread dimension.
`gpuB`	the dimension maps to a gpu block dimension.

Tagging a dimension with a processor type indicates that the dimension will be distributed over processors of that type; for example, tagging a dimension with *cpu* will execute each iteration of that loop dimension on a separate CPU.

Other tags that optimize a dimension include:

`vec(s)`	vectorize the dimension (s is the vector length)
`unroll`	unroll the dimension

Computations mapped to the same processor are ordered by projecting the computation set onto the time dimensions and comparing their lexicographical order.

4.3.3 Layer III (Data Management)

Layer III makes the data-layout concrete by specifying where intermediate values are stored. Any necessary buffer allocations/deallocations are also constructed in this level. This layer is generated automatically from Layer II by applying the scheduling commands for data mapping.

The data management layer specifies memory locations for storing computed values. It consists of the Layer II representation along with allocation/deallocation statements, and a set of *access relations*, which map a computation from Layer II to array elements read or written by that computation. Scalars are treated as single-element arrays. For each buffer, an allocation statement is created, specifying the type of the buffer and its size. Similarly, a deallocation statement is also added.

Possible data mappings in TIRAMISU include mapping computations to structures-of-arrays, arrays-of-structures, and contraction of multidimensional arrays into arrays with fewer dimensions or into scalars. It is also possible to specify more complicated accesses such as the storage of computations $c(i, j)$ into the array elements $c(i\%2, j\%2)$ or into $c(j, i)$.

In the example, setting the data access using `by.store_in(c,i,j)` indicates that the result of the computation $by(i, j, c)$ is stored in the array element $by[c, i, j]$. This command generates the following map in layer III:

$\{by(1, i0(gpuB), j0(gpuB), i1(gpuT), j1(gpuT), c) \rightarrow by[c, i0*32+i1, j0*32+j1] : i0 = floor(i/32) \wedge j0 = floor(j/32) \wedge i1 = i\%32 \wedge j1 = j\%32 \wedge 0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3\}$

Data mapping in TIRAMISU is an affine relation that maps a computation from Layer II to a buffer element. TIRAMISU

[1]For example the computation $S0(0, 0, 0)$ is lexicographically before the computation $S0(0, 0, 1)$ and the computations $S0(0, i, 0)$ are lexicographically before the computations $S0(1, i, 0)$

allows any data-layout mapping that can be expressed as an affine relation.

4.3.4 Layer IV (Communication Management)

Layer IV adds synchronization and communication operations to the representation, mapping them to the time-space doain, and concretizes when statements for buffer allocation/deallocation occur. This layer is generated automatically from Layer III by applying user-specified commands. Any allocation or deallocation operation added in Layer III is also mapped to the time-space domain in this layer.

5 Compiling the Tiramisu IR Layers

Since the main contribution of this paper is not in introducing new techniques for code generation, we only provide a high level overview of how Tiramisu generates the IR layers and target code. Throughout the section, we refer the reader to the necessary literature related to code generation for more details.

In order to generate code, a Tiramisu user provides an algorithm and a set of scheduling commands. Code generation is performed in five phases. In the first phase, the Layer I representation is created automatically from the algorithm. In the second phase, Layer II is generated automatically by applying scheduling commands that apply loop nest transformations and tag loop level dimensions for specific hardware units. Commands for buffer allocation and data-layout mapping are considered in the third phase. These commands augment the Layer II representation; the result constitutes Layer III. The newly added buffer allocation statements are not yet scheduled. In the fourth phase, scheduling commands that map buffers to different memory hierarchies (`tag_gpu_global()`, `tag_gpu_shared()`, `tag_gpu_local()` and `tag_gpu_constant()`), those that specify communication (`host_to_device()`, `device_to_host()`, `copy_at()`) and synchronization are applied. All the new operations that are added in this phase and in the previous one are scheduled. The result of this phase is the Layer IV representation. In the final phase, an annotated abstract syntax tree (AST) is generated from Layer IV; this AST is traversed to generate the target code.

In the rest of this section we describe how scheduling commands transform Layers I, II, III and IV. We also describe how target code is generated from Layer IV.

Transforming Layer I into Layer II Transforming Layer I into Layer II is done using two types of scheduling commands: (1) commands for loop nest transformations (such as `tile()`, `split()`, `shift()`, `interchange()`); and (2) commands for mapping loop levels to hardware (including `parallelize()`, `vectorize()`, `gpu()`).

The first type of scheduling command applies a map that transforms the iteration domain. For example, when a tiling command is applied on the by computation in Figure 2, it gets translated into the following map:

$$\{by(i, j, c) \rightarrow by(i0, j0, i1, j1, c) : i0 = floor(i/32) \wedge i1 = i\%32 \wedge j0 = floor(j/32) \wedge j1 = j\%32 \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

This map is then applied on the Layer I representation producing the Layer II representation (presented in the previous section). Composing transformations is done by composing different maps, since the composition of two affine maps is an affine map.

The second type of command adds space tags to dimensions to indicate whether which loop levels should be parallelized, vectorized, mapped to a GPU block, and so on.

Transforming Layer II into Layer III This is done by augmenting Layer II with access relations. By default, Tiramisu uses identity access relations (i.e., access relations that store a computation `C(i,j)` into a buffer `C[i,j]`). If the `store_in()` command is used, the access relation is deduced from that command instead. Buffer allocations are also added while transforming Layer II into Layer III. The scheduling command `b.allocate_at(C, i)` creates a new statement that allocates the buffer `b` in the same loop nest of the computation `C` but at the loop level `i`. The iteration domain of the allocation statement is deduced automatically from the iteration domain of `C` as follows: in Layer II, the iteration domain of `C` is already transformed into the time-space domain. The iteration domain of `b` is equal to the iteration domain of `C` except that the dimensions after `i` (inner loop levels) are projected out and removed.

Transforming Layer III into Layer IV Scheduling commands for data communication (send and receive), synchronization and for copying data between global, shared and local memory are all translated into statements. For example, the `send()` and `receive()` commands are translated into function calls that will be translated during code generation into MPI calls.

5.1 Code Generation

Generating code from the set of computations in Layer IV amounts to generating nested loops that visit each computation in the set, once and only once, while following the lexicographical ordering between the computations [5, 23, 34]. Tiramisu relies on an implementation of the Cloog [5] code generation algorithm provided by the ISL library [43]. The Tiramisu code generator takes Layer IV IR and generates an abstract syntax tree (AST). The AST is then traversed to generate lower level code for specific hardware architectures.

5.1.1 Multicore CPU

Tiramisu generates LLVM for multicore CPUs. When generating code that targets multicore shared memory systems, loop levels tagged with space *cpu* dimensions are translated into parallel loops in the generated code, using OpenMP-style parallelism. Loops tagged with the *vec* space dimensions are vectorized. Currently, we only support vectorization of loops that do not contain control flow.

5.1.2 GPU (CUDA)

The GPU code generator generates LLVM IR for the host code and CUDA code for the kernel code. Data copy commands and information about where to store buffers (shared, constant, or global memory) are all provided in Layer IV. Tiramisu translates these into the equivalent data copies and buffer allocations in the generated code. Computation dimensions tagged with GPU thread or GPU block tags are translated into the appropriate GPU thread and block IDs in the lowered code. The Tiramisu code generator can generate coalesced array accesses and can use shared and constant memories. It can also avoid thread divergence by separating full tiles (loop nests with a size that is multiple of the tile size) from partial tile (the remaining part of a loop). The final output of the GPU code generator is an optimized CUDA code.

5.1.3 Distributed Memory Systems

Tiramisu utilizes MPI to generate code for distributed memory systems. During code generation, we postprocess the generated code and convert each distributed loop into a conditional based on the rank of the executing process. For example:

```
for(q in 1..N-1) {...} // distribute on q
```

becomes:

```
q = get_rank(); if (q≥1 and q<N-1) {...}
```

5.2 Support for Non-Affine Iteration Spaces

Tiramisu represents non-affine array accesses, non-affine loop bounds, and non-affine conditionals in a way similar to Benabderrahmane et al. [6]. For example, a conditional is transformed into a predicate and attached to the computation. The list of accesses of the computation is the union of the accesses of the computation in the two branches of the conditional; this is an over-approximation. During code generation, a preprocessing step inserts the conditional back into the generated code. The efficiency of these techniques was confirmed in the PENCIL compiler [4]. Our experiences in general, as well as the experiments in this paper, show that these approximations do not hamper performance.

6 Evaluation

We evaluate Tiramisu on a set of image processing and stencil benchmarks. We compare it with two other compilers: Halide[35], an industrial-quality DSL for image processing that has a scheduling language, and PENCIL [3], a state-of-the-art fully automatic polyhedral compiler.

We performed the evaluation on a cluster of 16 nodes. Each node is a dual-socket machine with two 24-core Intel Xeon E5-2680v3 CPUs, 128 GB RAM, Ubuntu 14.04, and an Infiniband interconnect. We use the MVAPICH2 2.0 [21] implementation of MPI for the distributed tests. The multicore experiments (CPU) are performed on one of these nodes. GPU experiments are performed on an NVIDIA Tesla K40

Architectures	Frameworks	Benchmarks						
		edge Detector	cvtColor	convolution	warp Affine	gaussian	nb	ticket #2373
Single-node multicore	Tiramisu	1	1	1	1	1	1	1
	Halide	-	1	1	1	1	3.77	-
	PENCIL	2.43	2.39	11.82	10.2	5.82	1	1
GPU	Tiramisu	1.05	1	1	1	1	1	1
	Halide	-	1	1.3	1	1.3	1.7	-
	PENCIL	1	1	1.33	1	1.2	1.02	1
Distributed (16 Nodes)	Tiramisu	1	1	1	1	1	1	1
	Dist-Halide	-	1.31	3.25	2.54	1.57	1.45	-

Figure 5. A heatmap comparing the normalized execution times of code generated by Tiramisu with other frameworks (lower is better). Comparison is performed on three architectures: single-node multicore, GPU, distributed (16 nodes). "-" indicates unsupported benchmarks.

with 12 GB of RAM. Each experiment is repeated 30× and the median time is reported.

We used the following benchmarks in our evaluation: `edgeDetector`, a ring blur followed by Roberts edge detection [37]; `cvtColor`, which converts an RGB image to grayscale; `convolution`, a simple 2D convolution; `warpAffine`, which does affine warping on an image; `gaussian`, which performs a gaussian blur; `nb`, a synthetic pipeline composed of 4 stages that computes a negative and a brightened image from the same input image; and `ticket #2373`, a code snippet from a bug filed against Halide where the inferred bounds are over-approximated, causing the generated code to fail due to an assertion during execution. Four of these benchmarks have non-affine array accesses and non-affine conditionals for clamping (to handle boundary cases): `edgeDetector`, `convolution`, `warpAffine` and `gaussian`. We used a 2112 × 3520 RGB input image for the experiments.

Figure 5 compares the normalized execution time of code generated by Tiramisu to other state-of-the-art frameworks on three architectures: single-node multicore, GPU and distributed (16 nodes). For the single-node multicore and GPU we compare Tiramisu to Halide, and to PENCIL, a fully automatic polyhedral compiler. For the distributed architecture, we compare only to distributed Halide [11] since PENCIL does not generate distributed code and the only other polyhedral compiler that generates distributed code (pluto-mpi) does not support non-affine code (4/7 of our benchmarks have non-affine code).

Single-node multicore In four of the benchmarks, the performance of the code generated by Tiramisu matches the performance of Halide. We use the same schedule for both implementations; these schedules were hand-written by Halide experts. The results for `edgeDetector`, `convolution`, `warpAffine` and `gaussian` which have non-affine array accesses and conditionals show that Tiramisu handles such pattern efficiently.

Two of the other benchmarks, `edgeDetector` and `ticket #2373`, cannot be implemented in Halide. The following code snippet shows `edgeDetector`:

```
/* Ring Blur Filter */
R(i,j) = (Img(i-1,j-1) + Img(i-1,j) + Img(i-1,j+1)+
```

```
        Img(i,j-1)   +              Img(i,j+1)  +
        Img(i+1,j-1) + Img(i+1,j) + Img(i+1,j+1))/8;
/* Roberts Edge Detection Filter */
Img(i,j) = abs(R(i,j)-R(i+1,j-1)) + abs(R(i+1,j)-R(i,j-1));
```

`edgeDetector` cannot be implemented in Halide because it creates a cyclic dependence graph with a cycle length ≥ 1. Halide can only express programs with an acyclic dependence graph, with some exceptions; this restriction is imposed by the Halide language and compiler to avoid the need to prove the legality of some optimizations (since proving the legality of certain optimizations is difficult in the Halide interval-based representation). Tiramisu does not have this restriction since it checks transformation legality using dependence analysis [13].

In `ticket #2373`, which exhibits a triangular iteration domain, Halide's bounds inference over-approximates the computed bounds which leads the generated code to fail in execution. This over-approximation in Halide is due to the use of intervals to represent iteration domains, which prevents Halide from performing precise bounds inference for non-rectangular iteration spaces. Tiramisu can handle this case naturally since it relies on the polyhedral model where sets can include any affine constraint in addition to loop bounds. These examples show that the model exposed by Tiramisu naturally supports more complicated code patterns than an advanced, mature DSL compiler.

For `nb`, the code generated from Tiramisu achieves 3.77× speedup over the Halide-generated code. This is primarily due to loop fusion. In this code, Tiramisu enhances data locality by fusing loops into one loop; this is not possible in Halide, which cannot fuse loops if they update the same buffer. Halide makes this conservative assumption because otherwise it cannot prove the fusion is legal. This is not the case for Tiramisu which uses dependence analysis to prove correctness.

In comparison with PENCIL, the slowdown in `gaussian` is due to a suboptimal decision made by the PENCIL compiler. The `gaussian` kernel is composed of two successive loop nests (each of them has three loop levels). PENCIL decides to interchange the two innermost loop levels in order to enable the fusion of the two successive loop nests. This decision minimizes the distance between the producer and the consumer statements (first and second loop nests), but it also reduces spatial locality because it leads to non-contiguous memory accesses. The right decision in this cases is a trade-off. Such a trade-off is not captured by the Pluto automatic scheduling algorithm used within PENCIL. For the other kernels, both Tiramisu and Halide apply vectorization and unrolling on the innermost loops while PENCIL does not since the multi-core code generator of PENCIL does not implement these two optimizations. For `warpAffine`, both Tiramisu and Halide have a high speedup over PENCIL because the unique loop nest in this benchmark has 25 statements, vectorizing the innermost loop transforms all of these statements to their vector equivalent while unrolling increases register reuse and instruction level parallelism on the 24 cores of the test machine.

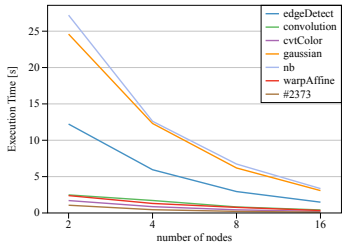

Figure 6. Execution time of distributed Tiramisu for 2, 4, 8, and 16 nodes

GPU For the GPU backend, the reported times are the total execution times (data copy and kernel execution). Code generated by Tiramisu for `convolution` and `gaussian` is faster than that of Halide because code generated by Tiramisu uses constant memory to store the weights array, while the current version of Halide does not use constant memory for its PTX backend. The only difference between the schedule of Tiramisu and Halide in these benchmarks is the use of `tag_gpu_constant()` in Tiramisu. Data copy times, for all the filters, are the same for Tiramisu and Halide. For `nb`, the code generated by Tiramisu achieves 1.7× speedup over that generated by Halide because Tiramisu is able to apply loop fusion which Halide cannot apply.

In comparison with PENCIL, the speedup in `convolution` and `gaussian` is due to the fact that PENCIL generates unnecessarily complicated control flow within the CUDA kernel which leads to thread divergence.

Distributed We assume the data are already distributed across the nodes by rows. Of these benchmarks, `nb`, `cvtColor` and `ticket #2373` do not require any communication; the other four require communication due to overlapping boundary regions in the distributed data.

Figure 5 compares the execution time of distributed Tiramisu and distributed Halide. Tiramisu is faster than distributed Halide in each case. It achieves up to 3.25× speedup for `convolution`. For the kernels involving communication, code generated by distributed Halide has two problems compared to Tiramisu: distributed Halide overestimates the amount of data it needs to send, and unnecessarily packs together contiguous data into a separate buffer before sending.

Halide overestimates the amount of data it needs to send because the benchmarks have array accesses that cannot be analyzed statically (the array accesses are clamped[2] to handle boundary cases), therefore Halide cannot compute the exact amount of data to send. To avoid this problem, Tiramisu uses explicit communication using the `send()` and `receive()` scheduling commands. The use of these two commands is the only difference between the Tiramisu and distributed Halide. These commands allow the user to specify exactly

[2] `clamp(i, 0, N)` returns 0 if $i < 0$, N if $i > N$ and returns i otherwise

the amount of data to send and also allow the compiler to avoid unnecessary packing.

Figure 6 shows the execution time of the kernels with distributed Tiramisu when running on 2, 4, 8, and 16 nodes. This graph shows that distributed code generated from Tiramisu scales well as the number of nodes increases (strong scaling).

7 Acknowledgement

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

8 Conclusion

This paper introduces Tiramisu, an optimization framework that features a scheduling language with scheduling commands for targeting multicore CPUs, GPUs, and distributed systems. A four-layer intermediate representation that separates the algorithm, when and where computations occur, the data layout and the communication is used to implement the scheduling language. We evaluate Tiramisu by targeting a variety of backends and demonstrate that it generates code matching and outperforming state-of-the-art frameworks.

References

[1] Saman P. Amarasinghe and Monica S. Lam. 1993. Communication Optimization and Code Generation for Distributed Memory Machines. *SIGPLAN Not.* 28, 6 (June 1993), 126–138. https://doi.org/10.1145/173262.155102

[2] Riyadh Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, A. Betts, J. Ketema, A. F. Donaldson, R. David, and E. Hajiyev. 2015. PENCIL: a Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *under review*. http://www.di.ens.fr/~baghdadi/public/papers/pencil.pdf

[3] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. 2015. PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 138–149. https://doi.org/10.1109/PACT.2015.17

[4] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven van Haastregt, Alexey Kravets, and Alastair F. Donaldson. 2015. *PENCIL Language Specification*. Research Rep. RR-8706. INRIA. 37 pages. https://hal.inria.fr/hal-01154812

[5] Cédric Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT–13 IEEE International Conference on Parallel Architecture and Compilation Techniques*. Juan-les-Pins, France, 7–16. http://hal.ccsd.cnrs.fr/ccsd-00017260 Classement CORE : A, nombre de papiers acceptés : 23, soumis : 122, student award.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The Polyhedral Model is More Widely Applicable Than You Think. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction (CC'10/ETAPS'10)*. Springer-Verlag.

[7] U. Bondhugula. 2013. Compiling affine loop nests for distributed-memory parallel architectures. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503289

[8] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI*. 101–113.

[9] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report 08-897. U. of Southern California.

[10] Alain Darte and Guillaume Huard. 2005. New Complexity Results on Array Contraction and Related Problems. *J. VLSI Signal Process. Syst.* 40, 1 (May 2005), 35–55. https://doi.org/10.1007/s11265-005-4937-3

[11] Tyler Denniston, Shoaib Kamil, and Saman Amarasinghe. 2016. Distributed halide. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 5.

[12] P. Feautrier. 1988. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*. ACM, St. Malo, France, 429–441. https://doi.org/10.1145/55364.55406

[13] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (Feb. 1991), 23–53. https://doi.org/10.1007/BF01407931

[14] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317.

[15] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.* 34, 3, Article 12 (May 2008), 25 pages. https://doi.org/10.1145/1356052.1356053

[16] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 66, 10 pages.

[17] Tobias Grosser, Armin Groslinger, and Christian Lengauer. 2012. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters* 22, 4 (2012). http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12

[18] M. Gupta. 1997. On privatization of variables for data-parallel execution. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*. IEEE, 533–541.

[19] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. *Loop Transformation Recipes for Code Generation and Auto-Tuning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 50–64.

[20] Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2010. Loop Transformation Recipes for Code Generation and Auto-tuning. In *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing (LCPC'09)*. Springer-Verlag, Berlin, Heidelberg, 50–64. https://doi.org/10.1007/978-3-642-13374-9_4

[21] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. 2006. Design of high performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, Vol. 1. IEEE, 43–48.

[22] Intel, Inc. 2018. Intel Math Kernel Library. https://software.intel.com/en-us/mkl. (April 2018).

[23] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Symp. on Principles of Programming Languages (POPL'88)*. San Diego, CA, 319–328.

[24] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. 2007. Effective Automatic Parallelization of Stencil Computations. *SIGPLAN Not.* 42, 6 (June 2007), 235–244. https://doi.org/10.1145/1273442.1250761

[25] Vincent Lefebvre and Paul Feautrier. 1998. Automatic storage management for parallel programs. *Parallel Comput.* 24 (1998), 649–671. https://doi.org/10.1016/S0167-8191(98)00029-5

[26] Zhiyuan Li. 1992. Array privatization for parallel execution of loops. In *Proceedings of the 6th international conference on Supercomputing*. ACM, Washington, D. C., United States, 313–322. https://doi.org/10.1145/143369.143426

[27] D Maydan, S Amarsinghe, and M Lam. 1992. Data dependence and data-flow analysis of arrays. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 434–448.

[28] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. 1993. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*. Charleston, South Carolina, United States, 2–15. https://doi.org/10.1145/158511.158515

[29] Samuel Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers.

[30] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic Optimization for Image Processing Pipelines. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 429–443. https://doi.org/10.1145/2786763.2694364

[31] Nvidia. 2012. *cuBLAS Library User Guide*. http://docs.nvidia.com/cuda/cublas

[32] Feautrier Paul and Lengauer Christian. 2011. The Polyhedron Model. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1581, 1592.

[33] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'11)*. ACM Press, Austin, TX, 549–562.

[34] F. Quilleré and S. Rajopadhye. 2000. Optimizing Memory Usage in the Polyhedral Model. *ACM Trans. on Programming Languages and Systems* 22, 5 (Sept. 2000), 773–815.

[35] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (July 2012), 12 pages.

[36] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. 519–530.

[37] Lawrence G. Roberts. 1963. *Machine perception of three-dimensional solids*. Ph.D. Dissertation. Massachusetts Institute of Technology. Dept. of Electrical Engineering.

[38] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. 2018. A Unified Backend for Targeting FPGAs from DSLs. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 1–8. https://doi.org/10.1109/ASAP.2018.8445108

[39] William Thies, Frédéric Vivien, Jeffrey Sheldon, and Saman Amarasinghe. 2001. A unified framework for schedule and storage optimization. In *Proc. of the 2001 PLDI Conf.*

[40] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. (Jan. 2010).

[41] Peng Tu and David Padua. 1994. Automatic array privatization. In *Languages and Compilers for Parallel Computing*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Lecture Notes in Computer Science, Vol. 768. Springer Berlin / Heidelberg, 500–521.

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

[43] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *ICMS*, Vol. 6327. 299–302.

[44] Michael E Wolf and Monica S Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on parallel and distributed systems* 2, 4 (1991), 452–471.

[45] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *Proc. Wkshp. Performance Optimization of High-level Languages and Libraries (POHLL), at IEEE Int'l. Par. Distrib. Processing Symp. (IPDPS)*. Long Beach, CA, USA, 1–8. https://doi.org/10.1109/IPDPS.2007.370637

[46] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 17–31.