TIRAMISU: A Polyhedral Compiler for Expressing Fast and Portable Code

Riyadh Baghdadi
MIT, USA
baghdadi@mit.edu

Jessica Ray
MIT
jray@csail.mit.edu

Malek Ben Romdhane
MIT
malek@mit.edu

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

Abdurrahman Akkas
MIT
akkas@mit.edu

Yunming Zhang
MIT
yunming@mit.edu

Patricia Suriana
Google
psuriana@google.com

Shoaib Kamil
Adobe
kamil@adobe.com

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 commands to explicitly manage the complexities that arise when targeting these systems. The framework is designed for the areas of image processing, stencils, linear algebra and deep learning. TIRAMISU has two main features: it relies on a flexible representation based on the polyhedral model and it has a rich scheduling language allowing fine-grained control of optimizations. TIRAMISU uses a four-level intermediate representation 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, deep learning, and linear algebra benchmarks and compare them with state-of-the-art compilers and hand-tuned libraries. We show that TIRAMISU matches or outperforms existing compilers and libraries on different hardware architectures, including multicore CPUs, GPUs, and distributed machines.**

***Index Terms*—Code Optimization, Code Generation, Polyhedral Model, Deep Learning, Tensors, GPUs, Distributed Systems**

I. 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 [20], 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, including coalescing memory accesses, managing data movement between global, shared, and register memory, and inserting synchronization primitives. Automatically generating such complex code is still beyond the capabilities of state-of-the-art compilers. The importance of kernels such as `gemm` motivates vendors to release immensely complex hand-optimized libraries for these 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 complicated code transformations.

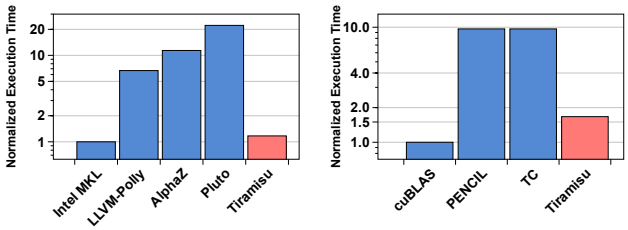

Fig. 1: Normalized execution times of code generated for `sgemm` on CPU (left) and GPU (right).

Previous work using the polyhedral model has shown success in implementing complex iteration space transformations [49], [8], [44], [22], [46], [37], data locality optimizations [27], [21], and memory management optimizations [17], [43], [29], [38], [13]. Although polyhedral compilers can represent these program and data transformations, they still do not successfully select transformations that result in the best performance. Currently, these compilers do not match the performance of hand-optimized kernels for algorithms such as `gemm`. The blue bars in Figure 1 show the performance of state-of-the-art polyhedral compilers for `gemm` compared to the Intel MKL [26] and Nvidia cuBLAS [35] libraries. Fully-automatic polyhedral compilers such as Polly [22] and Pluto [8] improve productivity, but do not obtain the desired level of performance since their search techniques consider only a subset of the necessary optimizations and rely on less accurate machine models, leading the compiler to make suboptimal decisions. Other polyhedral frameworks, such as AlphaZ [51] and CHiLL [10], 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[1], a polyhedral compiler with a scheduling language featuring *novel commands 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 (a 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 code transformations, data layout, and communication. TIRAMISU targets multicore CPUs, CUDA GPUs, distributed architectures, and FPGA. This paper presents the first three backends while Del Sozzo et al. [14] describe an FPGA backend.

The use of a scheduling language has been shown effective for generating efficient code by multiple compilers including CHiLL, AlphaZ, and Halide [39], [40]. In comparison with Halide in particular, not only does TIRAMISU introduce novel scheduling extensions, TIRAMISU fundamentally differs in that it relies on the expressive polyhedral representation instead of the interval-based representation used by Halide. This allows TIRAMISU to naturally 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 naturally expressible in Halide.

This paper makes the following contributions:

- We introduce a polyhedral compiler with a scheduling language that features *novel commands 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 deep learning and linear algebra kernels and show that TIRAMISU can generate efficient code that outperforms Intel MKL by up to $2.3\times$. We also evaluate TIRAMISU on a set of image processing benchmarks and show that TIRAMISU matches or outperforms state-of-the-art compilers on different hardware architectures, including multicore CPUs, GPUs, and distributed machines.

[1]http://tiramisu-compiler.org/

II. RELATED WORK

a) Polyhedral compilers with automatic scheduling: Polyhedral compilers such as PENCIL [4], [3], Pluto [8], Polly [22], Tensor Comprehensions [46], and PolyMage [34] 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 fully automatic compilers provide productivity, they may not always obtain the best performance. This suboptimal performance is due to several reasons: first, these compilers do not implement some key optimizations such as array packing [20], 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 (used in Pluto, PENCIL and Polly) tries to minimize the distance between producer and consumer statements while maximizing outermost parallelism, but it does not consider data layout, redundant computations, or the complexity of the control of the generated code. Instead of fully automatic scheduling, TIRAMISU relies on a set of scheduling commands, giving the user full control over scheduling.

Polyhedral frameworks proposed by Amarasinghe et al. [1] and Bondhugula et al. [7] address the problem of automatic code generation for distributed systems. Instead of being fully automatic, TIRAMISU relies 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, cost of communication versus re-computation, etc.).

b) Polyhedral compilers with a scheduling language: AlphaZ [51], CHiLL [10], [24] and URUK [19] are polyhedral frameworks developed to allow users to express high-level transformations using scheduling commands. Since these frameworks are polyhedral, they can express any affine transformation. However, their scheduling languages do not target distributed architectures nor GPUs. In contrast, 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 I compare between TIRAMISU and three representative polyhedral frameworks.

c) Non-polyhedral compilers with a scheduling language: Halide [39] is an image processing DSL with a scheduling language that 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, and this is the reason

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
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
Non-rectangular iteration spaces	Yes	Yes	Yes	Yes	Limited
Exact dependence analysis	Yes	Yes	Yes	Yes	No
Compile-time set emptiness check	Yes	Yes	Yes	Yes	No
Implement parametric tiling	No	Yes	No	No	Yes

TABLE I: Comparison between different frameworks.

why distributed Halide [15] over-approximates the amount of data to communicate (send and receive) when generating distributed code. This also makes some 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). The use of intervals also prevents Halide from performing many complex affine transformations, such as iteration space skewing.

Halide does not have dependence analysis and thus 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 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 work-around 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 I summarizes the comparison between TIRAMISU and Halide.

Vocke et al. [48] extend Halide to target DSPs, and add scheduling commands such as `store_in` to specify in which memory hierarchy data should be stored. TVM [11] is another system that shares many similarities with Halide. It uses a modified form of the Halide IR internally. Since TVM is also a non-polyhedral compiler, the differences between Halide and TIRAMISU that are due to the use of polyhedral model also apply to TVM.

POET [50] is a system that uses an XML-based description of code and transformation behavior to parametrize loop transformations. It uses syntactic transformations, which are less general than the polyhedral transformations used in TIRAMISU. GraphIt [52] is another compiler that has a scheduling language but that is mainly designed for the area of graph applications.

d) Other Compilers: Delite [9] is a generic framework for building DSL compilers. It exposes several parallel computation patterns that DSLs can use to express parallelism. NOVA [12] and Lift [42] are IRs for DSL compilers. They are functional languages that rely on a suite of higher-order functions such as map, reduce, and scan to express parallelism. TIRAMISU is complementary to these frameworks as TIRAMISU allows complex affine transformations that are easier to express in the polyhedral model.

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

Fig. 2: `Blur` algorithm without scheduling commands.

III. THE TIRAMISU EMBEDDED DSL

TIRAMISU is a domain-specific language (DSL) embedded in C++. It provides a C++ API that allows users to write a high level, architecture-independent algorithm and a set of scheduling commands that guide code generation. 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).

A. Scope of TIRAMISU

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 image processing, deep learning, dense linear algebra, tensor operations and stencil computations.

B. 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 is composed of a sequence of computations. A computation is used to represent a statement in TIRAMISU. Flow-control around 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. This algorithm declares two computations, `bx` and `by`. 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.

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

C. Scheduling Commands

TIRAMISU provides a set of high-level scheduling commands for common optimizations; Table II shows some examples. There are four types of scheduling commands:

- Commands for loop nest transformations: these commands include common affine transformations such as loop tiling, splitting, shifting, etc. For example, applying 32×32 loop tiling to 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 GPU block or thread dimensions. For example, calling `C.vectorize(j, 4)` splits the `j` loop by a factor of 4 and maps the inner loop to vector lanes.
- 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.

Novel commands introduced by TIRAMISU are highlighted in **bold** in Table II. 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 needing 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

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 loop levels (i, j) of the computation C by $t1 \times t2$. The names of the new loop levels are (i0, j0, i1, j1) where i0 is the outermost loop level and j1 is the innermost.
`C.interchange(i, j)`	Interchange the i and j loop levels of C.
`C.shift(i, s)`	Loop shifting (shift the loop level i by s iterations).
`C.split(i, s, i0, i1)`	Split the loop level i by s. (i0, i1) are the new loop levels.
`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 loop level 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)`	Parallelize the i loop level for execution on a shared memory system.
`C.vectorize(i, v)`	Vectorize the loop level i by a vector size v.
`C.gpu(i0, i1, i2, i3)`	Mark the loop levels i0, i1, i2 and i3 to be executed on GPU. (i0, i1) are mapped to block IDs and (i2, i3) to thread IDs.
`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)`	Parallelize the loop level i for execution on a distributed memory system.
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(P,i)`	Cache (copy) the buffer of C in shared memory. Copying from global to shared GPU memory will be done at loop level i of the computation P. The amount of data to copy, the access functions, and synchronization are computed automatically.
`C.cache_local_at(P, 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 of the send; `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 allocates b at the loop i of p. An operation can be scheduled like any 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 buffer to be stored in global GPU memory.
`b.tag_gpu_shared()`	Tag buffer to be stored in shared GPU memory.
`b.tag_gpu_local()`	Tag buffer to be stored in local GPU memory.
`b.tag_gpu_constant()`	Tag 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 II: Examples of TIRAMISU Scheduling Commands

TIRAMISU Scheduling Commands | **Pseudocode Representing Code Generated by TIRAMISU**

(a)

```
// Scheduling commands for targeting
// a multicore architecture.

// Tiling and parallelization.
Var i0, j0, i1, j1;
by.tile(i, j, 32, 32, i0, j0, i1, j1);
by.parallelize(i0);
bx.compute_at(by, j0);
```

```

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

   for(i1 in 0..min(N-2,32))
    for(j1 in 0..min(M-2,32))
    int i = i0*32+i1
    int j = j0*32+j1
    for (c in 0..3)
     by[i][j][c]=
        (bx[i][j][c] + bx[i+1][j][c]
                     + bx[i+2][j][c])/3
```

(b)

```
// 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);
```

```

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];
  // Tiling with redundancy
  GPUThread for(i1 in 0..min((N-2)%32,32)+2)
   GPUThread for(j1 in 0..min((M-2)%32,32)+2)
    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[i][j+2][c])/3

  GPUThread for(i1 in 0..min(N-2,32))
   GPUThread for(j1 in 0..min(M-2,32))
    int i = i0*32+i1
    int j = j0*32+j1
    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);
```

(c)

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

// Declare additional iterators
Var is(1, Nodes), ir(0,Nodes-1), i0, i1;

// Split loop i into loops i0 and i1 and
// parallelize i1
bx.split(i,N/Ranks,i0,i1); bx.parallelize(i1);
by.split(i,N/Ranks,i0,i1); by.parallelize(i1);

// Communicate the border rows where necessary
send s =
  send({is}, lin(0,0,0), M*2*3, is-1, {ASYNC});
recv r =
  receive({ir}, lin(N,0,0), M*2*3, ir+1,{SYNC},s);

// Order execution
s.before(r,root);
r.before(bx,root)

// Distribute the outermost loops
bx.distribute(i0); by.distribute(i0);
s.distribute(is); r.distribute(ir);
```

```
// We assume that in[][][] is initially
// distributed across nodes.  Each node
// has a chunk of the original
// in[][][] that we call lin[][][].

// Start by exchanging border rows of
// lin[][][]
distributed for (is in 1..Nodes)
  send(lin(0,0,0), M*2*3, is-1,{ASYNC})
distributed for (ir in 0..Nodes-1)
  recv(lin(N,0,0), M*2*3, ir+1, {SYNC})

distributed for (i0 in 0..Nodes)
  parallel for (i1 in 0..(N-2)/Nodes)
   int i = i0*((N-2)/Nodes) + i1
   for (j in 0..M-2)
     for (c in 0..3)
       bx[i][j][c] =
          (lin[i][j][c] + lin[i][j+1][c]
                        + lin[i][j+2][c])/3

distributed for (i0 in 0..Nodes)
  parallel for (i1 in 0..(N-2)/Nodes)
   int i = q*((N-2)/Nodes) + i1
   for (j in 0..M-2)
     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
```

Fig. 3: Three examples illustrating TIRAMISU scheduling commands (left) and the corresponding generated code (right). (a) shows scheduling commands for mapping to a multicore architecture; (b) shows scheduling commands for mapping to GPU; (c) uses commands to map to a distributed CPU machine.

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 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. TIRAMISU simplifies this step by automatically 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 it for execution on a multicore architecture. The necessary scheduling commands are shown in Figure 3-(a) (left). The `tile()` command tiles the computation `by`. The `compute_at()` command computes the tile of `bx` that needs to be consumed by `by` at the loop level `j0`. This transformation introduces redundant computations (in this case) and is known as overlapped tiling [28]. The `parallelize()` command parallelizes the `i0` loop.

Now let us take the same example but map the two outermost loops of `bx` and `by` to GPU. The necessary scheduling commands are shown in Figure 3-(b) (left). The `tile_gpu()` command tiles the computation `by` then maps the new loops to GPU block and thread dimensions. The `compute_at()` command computes the tile of `bx` needed by `by` at the loop level `j0` (this introduces redundant computations). `cache_shared_at()` instructs TIRAMISU to store the results of the `bx` computation in shared memory. Copying from global to shared memory will be done at the loop level `j0` of `by`. The subsequent `store_in()` command specifies the access functions for `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 we want to run the `blur` example on a distributed system with a number of multicore CPU nodes equal to `Nodes`. Figure 3-(c) (left) shows the scheduling commands to use in this case. We assume that the array `in[][][]` is initially distributed across nodes such that node `n` has the chunk of data represented by `in[n*((N-2)/Nodes)..(n+1)*((N-2)/Nodes),*,*]`. In other words, this corresponds to row `n*(N-2)/Nodes` through row `(n+1)*((N-2)/Nodes)`. This chunk is stored in the local array `lin[][][]`.

`send()` and `recv()` define communication for the border regions. Assuming that each node has a chunk of `in`. The `blur` computation for a chunk stored in node `n` requires the first two rows of data from the chunk stored in node `n+1`. These two rows are referred to as the border region. The `send()` will send 2 rows ($M \times 2 \times 3$ contiguous data elements) from node `is` to node `is-1` starting from `lin(0,0,0)`, which corresponds to the first two rows of the chunk on node `is`. In response, the `recv` for node `ir` will receive 2 rows ($M \times 2 \times 3$ contiguous data elements) from node `ir+1`, which corresponds to `ir` receiving the first two rows from node `ir+1`. The receive for node `ir` places these elements starting at the end of its local chunk by starting at `lin(N,0,0)`. Additionally, {`ASYNC`} defines an asynchronous send and {`SYNC`} defines a synchronous receive. 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 run on a different node).

All other scheduling commands in TIRAMISU can be composed with `send`s, `recv`s, and distributed loops, as long as the composition is semantically correct.

IV. 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.

A. Rationale for a Multi-layer IR

In this section we provide examples showing why current intermediate representations are not adequate for TIRAMISU and why we need a multi-layer IR.

Most current intermediate representations use memory to communicate between program statements. This creates memory-based dependencies in the program, and forces compilers to choose data layout before deciding on optimizations and mapping to hardware. Optimizing a program for different hardware architectures usually requires modifying the data layout and eliminating memory-based dependencies since they restrict optimizations [31]. 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 [23], [45], [30], [17], [33], [32], [29], [38], [13].

Another example that demonstrates the complexity of code generation 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 both 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 occur before scheduling.

TIRAMISU addresses these complexities in code generation by using a multi-layer IR that fully separates the architecture-independent algorithm from loop transformations, data layout and communication. The *first layer* representation describes the pure algorithm using producer-consumer relationships without memory locations. The *second layer* specifies the order of computation, 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 for applying optimizations and ensures that compiler passes 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. 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 [23], [45], [30], [17], [33], [32], [29], [38], [13].

B. 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 to transform iteration domains and memory accesses (apply loop nest and memory access transformations). More details and formal definitions for these concepts are provided in [47], [2], [36].

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 the tuples as we do in the previous set, we can describe the set using affine constraints over loop iterators and symbolic constants as follows:

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

where i and j are the dimensions of the 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) [47]. We also use the ISL library notation for sets and maps throughout the paper.

C. The Multi-Layer IR

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

1) Layer I (Abstract Algorithm): Layer I of TIRAMISU specifies the algorithm without specifying when and where computations occur, how data should be stored in memory, or communication. Values are communicated via explicit producer-consumer relationships.

For example, the Layer I representation of the code in Figure 2 for the computation `by` is as follows:

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

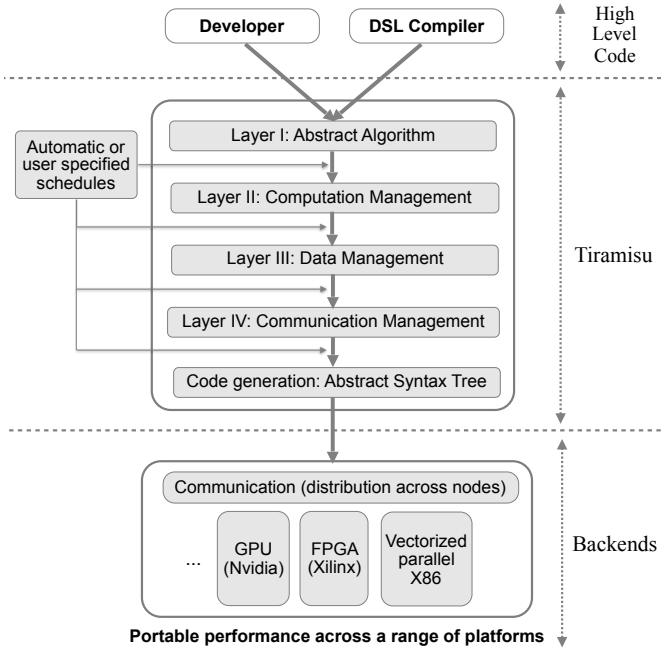

Fig. 4: TIRAMISU overview

The first part, $\{by(i,j,c) : 0 \leq i < N-2 \wedge 0 \leq j < M-2 \wedge 0 \leq c < 3\}$, specifies the iteration domain of the computation `by`, 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.

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-(b) (right) shows the GPU-optimized version of the code, produced by the 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[2]. The set before the colon in the representation is an ordered set of computations. The tag *gpuB* for the dimension $i0$ and $j0$ indicates that each iteration $(i0, j0)$ is

[2]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 $S0(1,i,0)$

mapped to the GPU block $(i0, j0)$. In Layer II, the total ordering of these tuples determines execution order.

Computations in this layer are ordered and assigned to a particular processor; the 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. Space dimensions are distinguished from time dimensions using tags, which consist of a processor type. 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 transform 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.

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. TIRAMISU generates this layer 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 of Figure 3-(b) (left), 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 each computation to a buffer element. TIRAMISU allows any data-layout mapping expressible as an affine relation.

4) Layer IV (Communication Management): Layer IV adds synchronization and communication operations to the representation, mapping them to the time-space domain, 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.

V. COMPILER IMPLEMENTATION

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 appropriate literature for more details.

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.

a) 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. 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 which loop levels to parallelize, vectorize, map to GPU blocks, and so on.

b) 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 loop level `i`.

c) 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 into MPI calls during code generation.

A. 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], [27], [38]. TIRAMISU relies on an implementation of the Cloog [5] code generation algorithm provided by the ISL library [47]. 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 (depending on the target backend).

The multicore CPU code generator generates LLVM IR from the AST. In order to generate LLVM IR, we use Halide as a library: we first generate the Halide IR then we lower the Halide IR to LLVM IR using Halide. We do not use Halide to perform any high level code optimization. All the code optimizations are performed by TIRAMISU before generating the Halide IR. The Halide compiler then lowers the Halide IR loops into LLVM IR.

The GPU code generator generates LLVM IR for the host code and CUDA 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 CUDA data copy calls 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 tiles (the remaining part of a loop).

The code generator for distributed memory systems utilizes MPI. During code generation, all the function calls for data copying are translated to the equivalent MPI function calls. The generated code is postprocessed and each distributed loop is converted into a conditional based on the MPI 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) {...}
```

B. 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 demonstrated by Benabderrahmane et al. [6] and 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.

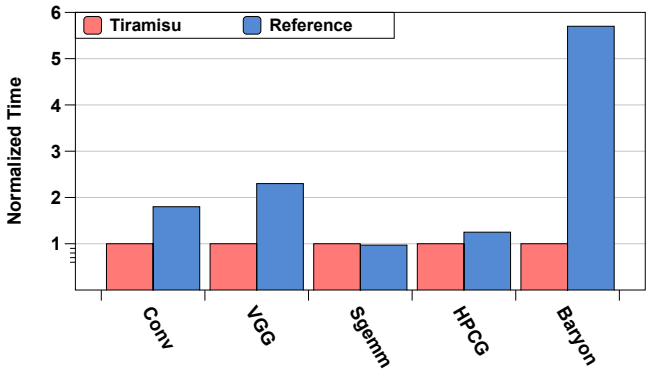

Fig. 5: Normalized Execution Times for Deep Learning, Linear and Tensor Algebra Benchmarks.

VI. EVALUATION

We evaluate TIRAMISU on two sets of benchmarks. The first is a set of deep learning and linear algebra benchmarks. The second is a set of image processing benchmarks.

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 [25] 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 with 12 GB of RAM. Each experiment is repeated $30\times$ and the median time is reported.

A. Deep Learning and Linear Algebra Benchmarks

We evaluated TIRAMISU by implementing a set of deep learning and linear algebra benchmarks, including `Conv` (a direct implementation of a neural network convolution layer), `VGG` (a block of a VGG neural network), and `sgemm` (matrix multiplication used to implement convolutions), `HPCG` (a benchmark for multigrid preconditioned conjugate gradient, CG)[3], and `Baryon` (a dense tensor contraction code for constructing Baryon Building Blocks [16]). For all of these benchmarks, we compare the TIRAMISU implementation with Intel MKL, except for `HPCG` and `Baryon`, where we compare TIRAMISU with reference implementations. Figure 5 shows a comparison between the performance of CPU code generated by Tiramisu and reference code. For `sgemm` and `HPCG` we use matrices of size 1060×1060 and vectors of size 1060 while for `Conv` and `VGG` we use 512×512 as the data input size, 16 as the number of input/output features and a batch size of 32. For `Baryon`, we use the same tensor sizes as in the reference code.

[3]http://www.hpcg-benchmark.org/

For sgemm, TIRAMISU matches the performance of Intel MKL. sgemm is interesting in particular because the Intel MKL implementation of this kernel is well-known for its hand-optimized performance. We used a large set of optimizations to match Intel MKL. These optimizations include two-level blocking of the three-dimensional sgemm loop, vectorization, unrolling, array packing, register blocking, and separation of full and partial tiles (which is crucial to enable vectorization, unrolling, and reducing control overhead). We also used auto-tuning to find the best tile size and unrolling factor for the machine on which we run our experiments.

For the Conv kernel, TIRAMISU outperforms the Intel MKL implementation because the TIRAMISU-generated code uses a fixed size for the convolution filter. We generate specialized versions for common convolution filter sizes (3×3, 5×5, 7×7, 9×9 and 11×11). This allows the TIRAMISU compiler to apply optimizations that Intel MKL does not perform; for example this allows TIRAMISU to unroll the innermost (convolution filter) loops since their size is known at compile time. In VGG, TIRAMISU fuses the two convolution loops of the VGG block, which improves data locality. In addition, we generate code with fixed sizes for convolution filters (as we did in Conv). This provides $2.3\times$ speedup over Intel MKL. The TIRAMISU speedup over the Baryon reference code is achieved through vectorization, but this vectorization is not trivial since it requires the application of array expansion and then the use of scatter/gather operations, which are both not implemented in the reference Baryon code.

B. Image Processing Benchmarks

We used the following image processing benchmarks in our evaluation: edgeDetector, a ring blur followed by Roberts edge detection [41]; cvtColor, which converts an RGB image to grayscale; conv2D, 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. This code simply has a loop that assigns a value to an array but the iteration space is not rectangular (it tests if x >= r where x and r are loop iterators). The inferred bounds in this code 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, conv2D, warpAffine and gaussian. We used a 2112×3520 RGB input image for the experiments.

We compare TIRAMISU with two other compilers: Halide [39], an industrial-quality DSL for image processing that has a scheduling language, and PENCIL [3], a state-of-the-art fully automatic polyhedral compiler.

Figure 6 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 PENCIL. For the distributed architecture, we compare to distributed Halide [15].

a) 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, conv2D, warpAffine and gaussian, which have non-affine array accesses and conditionals, show that TIRAMISU handles such access patterns 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 creates a cyclic dependence graph with a cycle length ≥ 1 (R is written in the first statement and read in the second while Img is written in the second and read in the first), but 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 [18].

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 $3.77\times$ 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.

The slowdown of the PENCIL compiler in gaussian is due to a suboptimal decision made by PENCIL. The gaussian kernel is composed of two successive loop nests (each of them contains 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

Architectures	Frameworks	Benchmarks						
		edge Detector	cvtColor	Conv2D	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	-

Fig. 6: 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.

the distance between producer and 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 case 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 multicore 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, and 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.

b) GPU: For the GPU backend, the reported times are the total execution times (data copy and kernel execution). Code generated by TIRAMISU for `conv2D` 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.

Compared to PENCIL, the speedup in `conv2D` and `gaussian` is due to the fact that PENCIL generates unnecessarily complicated control flow within the CUDA kernel, which leads to thread divergence.

c) 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.

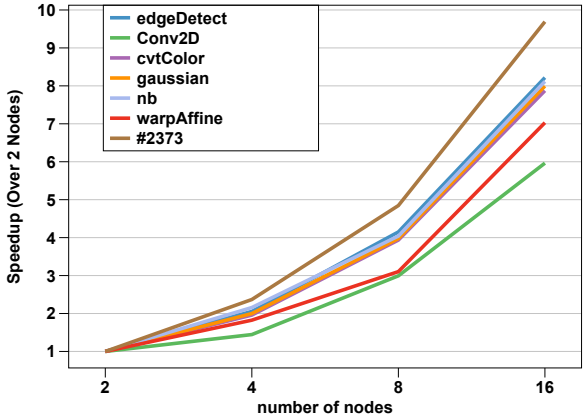

Fig. 7: Speedup of code generated by distributed TIRAMISU for 2, 4, 8, and 16 nodes. The baseline is the execution time on 2 nodes.

Figure 6 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 `conv2D`. 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.

Distributed 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[4] to handle boundary cases), therefore distributed 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 the amount of data to send and also allow the compiler

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

to avoid unnecessary packing.

Figure 7 shows the speedup 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).

VII. CONCLUSION

This paper introduces TIRAMISU, a polyhedral compiler framework that features a scheduling language with 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 compiler. We evaluate TIRAMISU by targeting a variety of backends and demonstrate that it generates code matching or outperforming state-of-the-art frameworks and hand-tuned code.

ACKNOWLEDGEMENTS

This work was supported by the ADA Research Center, a JUMP Center co-sponsored by SRC and DARPA.

REFERENCES

[1] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. *SIGPLAN Not.*, 28(6):126–138, June 1993.

[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. Pencil: a platform-neutral compute intermediate language for accelerator programming. In *under review*, 2015.

[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. 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, pages 138–149, Washington, DC, USA, 2015. IEEE Computer Society.

[4] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven van Haastregt, Alexey Kravets, and Alastair F. Donaldson. PENCIL language specification. Research Rep. RR-8706, INRIA, 2015.

[5] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT–13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004. Classement CORE : A, nombre de papiers acceptés : 23, soumis : 122, student award.

[6] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 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, 2010.

[7] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

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

[9] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *PPoPP*, pages 35–46, 2011.

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

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[12] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM.

[13] Alain Darte and Guillaume Huard. New complexity results on array contraction and related problems. *J. VLSI Signal Process. Syst.*, 40(1):35–55, May 2005.

[14] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco Domenico Santambrogio. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018.

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

[16] William Detmold and Kostas Orginos. Nuclear correlation functions in lattice qcd. *Physical Review D*, 87(11):114512, 2013.

[17] P. Feautrier. Array expansion. In *Proceedings of the 2nd international conference on Supercomputing*, pages 429–441, St. Malo, France, 1988. ACM.

[18] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, February 1991.

[19] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[20] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[21] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[22] Tobias Grosser, Armin Groslinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012.

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

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

[25] Wei Huang, Gopalakrishnan Santhanaraman, H-W Jin, Qi Gao, and Dhabaleswar K Panda. Design of high performance mvapich2: Mpi2 over infiniband. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 43–48. IEEE, 2006.

[26] Intel, Inc. Intel math kernel library. https://software.intel.com/en-us/mkl, April 2018.

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

[28] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *SIGPLAN Not.*, 42(6):235–244, June 2007.

[29] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[30] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proceedings of the 6th international conference on Supercomputing*, pages 313–322, Washington, D. C., United States, 1992. ACM.

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

[32] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. 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*, pages 2–15, Charleston, South Carolina, United States, 1993.

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

[34] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, March 2015.

[35] Nvidia. *cuBLAS Library User Guide*, 2012.

[36] Feautrier Paul and Lengauer Christian. The polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581, 1592. Springer, 2011.

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

[38] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. on Programming Languages and Systems*, 22(5):773–815, September 2000.

[39] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

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

[41] Lawrence G. Roberts. *Machine perception of three-dimensional solids*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering, 1963.

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

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

[44] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjodin, and Ramakrishna Upadrasta. GRAPHITE two years after: First lessons learned from Real-World polyhedral compilation, January 2010.

[45] Peng Tu and David Padua. Automatic array privatization. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 500–521. Springer Berlin / Heidelberg, 1994.

[46] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zach DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[47] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *ICMS*, volume 6327, pages 299–302, 2010.

[48] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. Extending halide to improve software development for imaging dsps. *ACM Trans. Archit. Code Optim.*, 14(3):21:1–21:25, August 2017.

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

[50] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. 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)*, pages 1–8, Long Beach, CA, USA, March 2007.

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

[52] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA):121:1–121:30, October 2018.