Technical Report about Tiramisu: a Three-Layered Abstraction for Hiding Hardware Complexity from DSL Compilers

Riyadh Baghdadi
MIT
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

Patricia Suriana
Google
psuriana@google.com

Shoaib Kamil
Adobe
kamil@adobe.com

Saman Amarasinghe
MIT
saman@csail.mit.edu

Abstract

High-performance DSL developers work hard to take advantage of modern hardware. The DSL compilers have to build their own complex middle-ends before they can target a common back-end such as LLVM, which only handles single instruction streams with SIMD instructions. We introduce Tiramisu, a common middle-end that can generate efficient code for modern processors and accelerators such as multicores, GPUs, FPGAs and distributed clusters. Tiramisu introduces a novel three-level IR that separates the algorithm, how that algorithm is executed, and where intermediate data are stored. This separation simplifies optimization and makes targeting multiple hardware architectures from the same algorithm easier. As a result, DSL compilers can be made considerably less complex with no loss of performance while immediately targeting multiple hardware or hardware combinations such as distributed nodes with both CPUs and GPUs. We evaluated Tiramisu by creating a new middle-end for the Halide and Julia compilers. We show that Tiramisu extends Halide and Julia with many new capabilities including the ability to: express new algorithms (such as recurrent filters and non-rectangular iteration spaces), perform new complex loop nest transformations (such as wavefront parallelization, loop shifting and loop fusion) and generate efficient code for more architectures (such as combinations of distributed clusters, multicores, GPUs and FPGAs). Finally, we demonstrate that Tiramisu can generate very efficient code that matches the highly optimized Intel MKL gemm (generalized matrix multiplication) implementation. We show that Tiramisu can generate code that outperforms Intel MKL DNN convolutions and show speedups reaching 4× over the original Halide and 16× over the original Julia due to optimizations enabled by Tiramisu.

, ,
.

Keywords code optimization framework, intermediate representation, domain specific language, polyhedral compilation, iteration space transformation

1 Introduction

With a diverse set of high-performance hardware platforms available, the development of a credible high-performance Domain Specific Language (DSL) compiler requires cross platform code generation [1, 2, 7, 14, 45, 49, 50]. Many of these DSL compilers target intermediate representations (IRs) such as the LLVM IR [36], a low-level representation that models an abstract, RISC-like CPU with infinite registers. However, before generating the low-level LLVM IR, the DSL compilers must create their own middle-end passes that transform their own architecture-independent high-level IR to take advantage of target hardware platforms such as multicores, distributed clusters, GPUs, and FPGAs, each with very different requirements.

These platform specific transformations cannot be done in the LLVM IR, as it encodes memory accesses with a concrete memory layout, which reduces optimization opportunities. This also adds unnecessary dependences such as anti- and output-dependences, forcing the compiler to modify data layout in order to perform certain optimizations. For example, compilers privatize [26, 38, 56] and expand [22, 43, 44] arrays and scalars in order to parallelize loop nests; subsequently, the arrays are contracted [19, 37, 48] to minimize the memory footprint.

This paper presents Tiramisu, a compiler middle-end that hides the complexity of the large variety of execution platforms and reduces the burden on DSL creators by providing a multi-layer representation suitable for transforming from the high-level, domain-specific representation of the DSL to GPUs, multicore CPUs, distributed machines, and FPGAs.

The novel multi-layer design exposes three different representations that make it easier to perform each program transformation at the right level of abstraction.

The top layer representation (*abstract computation layer*) describes the pure algorithm using producer-consumer relationships without memory locations. The second layer (*computation placement 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. Finally, the third layer (*concrete computation layer*) specifies where to store intermediate data before they are consumed. Tiramisu then lowers the third layer to a target architecture while performing back-end-specific code optimizations.

Tiramisu does not automatically parallelize or perform other transformation decisions; the goal of the middle-end framework is to provide mechanisms and not policy, which is the responsibility of the DSL compiler or the application programmer as in Halide [49]. Instead, Tiramisu concentrates on providing scheduling primitives that can be used to transform and map the programs to the underlying hardware. This frees the DSL designers from needless bookkeeping, analysis, and redundant implementation of code transformations and provides multiple architecture-specific code generation.

Tiramisu uses a unified framework based on polyhedral sets to represent the three layers. This makes it simple for Tiramisu to reason about and implement iteration space and data layout transformations, since these are represented as transformations on polyhedral sets. It also simplifies deciding about the legality of transformations (based on dependence analysis). The use of a unified general framework relieves compiler developers from developing custom algorithms and analyses that only work in certain situations and cannot be composed with other optimizations passes.

This paper makes the following contributions:

- We show that a unified middle-end for DSL compilers can generate code for multiple high-performance architectures such as multicore CPUs, GPUs, FPGAs, distributed machines, or any combination of these, using a set of simple scheduling commands to guide the program transformation and code generation.
- We present the first compiler framework that matches the performance of the highly optimized Intel MKL gemm (comparing a single gemm kernel from MKL to a single gemm kernel generated by Tiramisu). To the best of our knowledge, this is the first compiler framework that can demonstrate such performance. Former frameworks could outperform Intel MKL but by fusing multiple kernels, since a library such as Intel MKL cannot fuse kernels. Tiramisu can also do that to get even better performance.
- We introduce a novel three-layer intermediate representation for the middle-end that separates the algorithm from the code transformations and the data layout transformations, simplifying the composition of architecture-specific lowering transformations.
- We implemented this middle-end, Tiramisu, and demonstrate its power and viability by using it as the middle-end for Halide [49, 50] and Julia [7] compilers.
- We demonstrate the expressiveness of Tiramisu by extending Halide with many new capabilities such as tiling recurrent filters, performing precise bound inference for non-rectangular iteration spaces, and performing advanced loop nest transformations such as skewing, shifting, and loop fusion.
- We evaluate Tiramisu and show that we match Intel MKL gemm, show up to 4× performance gains in Halide and 16× in Julia.

2 Middle-End Challenges

Building a compiler middle-end requires overcoming many challenges. The middle-end must support a large enough class of computations to be usable by multiple DSLs. The middle-end must also produce code for most of the popular high-performance computing systems, including multicore CPUs, clusters of compute nodes and accelerators such as GPUs and FPGAs in many combinations. Finally, the middle-end has to be able to perform heroic program and data transformations and compose them together to generate highly efficient code. These transformations must be orchestrated and tailored to the application as well as the underlying execution hardware.

2.1 The Data Dependence Challenge

Most IRs use memory as a means of communication between program statements. This creates memory-based dependences in the program. It also means that the data-layout is chosen before optimizing the code. Optimizing a program for different hardware architectures usually requires modifying the data-layout (since different data layouts are suitable for different hardware architectures) and requires eliminating memory-based dependences since they restrict optimizations [42].

Consider the simple image processing pipeline in Figure 1-(a) (left), which has not been optimized yet. In order to optimize it for a multicore machine, we can parallelize the outermost loop (over the `i` iterator). To do so, we must first expand the two-dimensional arrays `b1` and `b2` into three-dimensional arrays. In addition to the parallelization, data locality can be improved by fusing the brightening and clamp stages. After fusion, we can replace the `b1` array accesses with accesses to a scalar. After applying these optimizations, we get the code shown in Figure 1-(b) (left). Assuming now that the target architecture is multi-GPU instead of CPU, changing from an Array-of-Structures (AoS) layout to a Structure-of-Arrays

(SoA) memory layout may lead to better performance. In order to maximize parallelism on GPU, we apply loop fission to the loop over `b2` and the loop over `out`. The result of applying these optimizations is shown in Figure 1-(c) (left).

Applying the previous data-layout transformations and the elimination of memory-based dependencies is, in general, challenging [19, 22, 26, 37, 38, 43, 44, 48, 56]. However, if the program dependencies are captured using producer-consumer relationships and the data-layout is not specified early on, all program transformations can be performed without the complexity of memory layout transformations. Most modern DSL IRs capture the dependencies with producer-consumer relationships. Thus, the middle-end compiler can implement program transformations using the producer-consumer relationships first and then introduce memory layouts as a further lowering step. This requires carefully designing the middle-end IR in multiple layers. We show that a three-layer IR design can overcome these difficulties.

2.2 The Program Representation Challenge

Lowering code to complex hardware architectures requires many transformations such as loop interchange, blocking, skewing, fission, fusion, and distribution. These transformations change the loop structure in complex ways by introducing new loops, complex loop bounds, and non-trivial access functions [59]. Analyzing code generated by one of these transformations is challenging, which complicates composition with other transformations. This problem can be solved if the loops are kept within a single unified representation through all the transformations. However, many representations are inadequate or too conservative to support complex transformations. They are also inadequate for performing tasks such as dependence analysis (which is necessary for deciding the correctness of optimizations) and the computation of communication sets in the general case. For example, the interval-based representation used in Halide [49] is unable to support accurate bounds computation for non-rectangular iteration spaces. It also cannot represent programs that have cyclic dependence graphs, and does not naturally support complex transformations such as loop skewing (wavefront parallelism). We show that a polyhedral based representation is more flexible, powerful, and is capable of supporting all transformations necessary in the middle-end.

2.3 The Optimization Orchestration Challenge

Producing high-performance code for a computer system from an architecture-independent algorithm requires a series of program transformations. These transformations are non-trivial and highly dependent on the program structure, input data, and the target architecture. Parallelizing compilers have worked to fully automate this process using cost models, heuristics [28], and machine learning [54]. However, obtaining performance comparable to hand-coded implementations using such fully automated approaches is still a challenging problem. A more promising approach is to expose optimizations to the user by providing a set of optimization directives. Languages such as Halide [13, 49] have shown that this approach is a viable one. However, in order to expand this to a common middle-end, the scheduling language must encompass a much broader set of transformations and seamlessly compose these transformations. Figures 1-(b) (Schedule) and 1-(c) (Schedule) show how a simple collection of scheduling commands can map the architecture independent program into different architectural configurations.

2.4 The MPI+OpenMP+CUDA Challenge

Most high performance computer systems are complex and heterogeneous. A typical supercomputer has multiple interconnected nodes equipped with multicore CPU processors with vector units connected using NUMA shared memory. They may also have multiple GPU accelerators per node [39]. Recently, data centers have been adding FPGA accelerators to this mix [10]. Getting the best performance requires the program to take full advantage of all these different components. In the supercomputing community this is usually referred to as the MPI+OpenMP+CUDA challenge [60]. Writing code that targets such heterogeneous systems is non-trivial, as each of these require drastically different styles of code and optimizations, all using different libraries and languages. Getting all of these components to communicate and synchronize is non-trivial as well. The state-of-the-art practice is to manually write the program in separate language extensions. However, any changes to the program partitioning between these heterogeneous units will normally require a complete rewrite of the program. With Tiramisu, it is possible to generate code for these complex computer systems in a simple, flexible and malleable way. Figure 1-(c) (left) shows an example where the algorithm is mapped to a GPU cluster using simple scheduling commands and without the need to change the algorithm or to write code in different languages and libraries.

3 Tiramisu Overview

Tiramisu is a middle-end compiler framework for DSLs. Using this framework, DSLs can transform their architecture-independent IRs into architecture-specific, low-level IRs while taking advantage of modern architectural features such as multicore parallelism, non-uniform memory (NUMA) hierarchies, clusters, and accelerators like GPUs and FPGAs. Tiramisu is designed for DSLs that logically operate over dense data using loop nests and sequences of statements, which is the case of DSLs for image processing, dense linear algebra, and stencil computations, among others.

3.1 The Three-Layer Intermediate Representation

Tiramisu uses *polyhedral sets* to represent each one of the three IR layers and uses *polyhedral set* and *relation operations*

Constraints: $C_n : 0 \leq i < N$, $C_m : 0 \leq j < M$, $C_{m'} : 1 \leq j < M-1$, $C_k : 0 \leq c < 3$, $C_q : 0 \leq q < NUM_NODES$

Different Code Optimizations | **Tiramisu representation (Layer I, Layer II and Layer III)**

(a)

```
1  // Original unoptimized code
2  for (i in 0..N)
3    for (j in 0..M)
4      for (c in 0..3)
5        b1[j][c] = 1.5*img[i][j][c] // brightening
6    for (j in 0..M )
7      for (c in 0..2)
8        b2[j][c] = clamp(b1[j][c], 0, 255)
9    for (j in 1..M-1)
10     for (c in 0..3)
11       out[i][j][c] = (b2[j-1][c] + b2[j][c] +
12                       b2[j+1][c])/3
```

Layer I

The constraints C_n, C_m and C_k are defined above.
$\{b_1\ (i, j, c) : C_n \land C_m \land C_k\}$: $1.5 * img(i, j, c)$
$\{b_2\ (i, j, c) : C_n \land C_m \land C_k\}$: $clamp(b_1(i, j, c), 0, 255)$
$\{out(i, j, c) : C_n \land C_{m'} \land C_k\}$: $(b_2(i, j-1, c) + b_2(i, j, c) + b_2(i, j+1, c))/3$

(b)

```
1  // Code optimized for CPU
2  parallel for (i in 0..N)
3    for (j in 0..M)
4      for (c in 0..3)
5        float t = 1.5*img[i][j][c]
6        b2[i][j][c] = clamp(t, 0, 255)
7    for (j in 1..M-1)
8      for (c in 0..3)
9        out[i][j][c] = (b2[i][j-1][c] +
10                       b2[i][j][c] +
11                       b2[i][j+1][c])/3
```

Schedule

b_2.after(b_1, c)
b_1.parallel(i); b_2.parallel(i); out.parallel(i)
b_1.store_in(t); b_2.store_in($buf_{b2}[i, j, c]$); out.store_in($buf_{out}[i, j, c]$);

Layer II

// Layer II generated from Layer I using the schedule

$\{b_1(i(cpu), 0, j, c, 0) : C_n \land C_m \land C_k\}$: $1.5 * img(i, j, c)$
$\{b_2(i(cpu), 0, j, c, 1) : C_n \land C_m \land C_k\}$: $clamp(b_1(i, 0, j, c, 0), 0, 255)$
$\{out(i(cpu), 1, j, c, 0) : C_n \land C_{m'} \land C_k\}$:
$(b_2(i, 0, j-1, c, 1) + b_2(i, 0, j, c, 1) + b_2(i, 0, j+1, c, 1))/3$

Layer III

Layer III = Layer II representation + the following data mapping

$\{b_1\ (i(cpu), 0, j, c, 0) \rightarrow t : C_n \land C_m \land C_k\}$
$\{b_2\ (i(cpu), 0, j, c, 1) \rightarrow buf_{b2}[i, j, c] : C_n \land C_m \land C_k\}$
$\{out(i(cpu), 1, j, c, 0) \rightarrow buf_{out}[i, j, c] : C_n \land C_{m'} \land C_k\}$

(c)

```
1  // Code optimized for multi-GPU
2  distributed for (q in 0..NUM_NODES)
3    gpu for (i in 0..N/NUM_NODES)
4      gpu for (j in 0..M)
5       for (c in 0..3)
6         float t = 1.5*img[i][j][c]
7         b2[c][i][j] = clamp(t, 0, 255)
8  distributed for (q in 0..NUM_NODES)
9    gpu for (i in 0..N/NUM_NODES)
10     gpu for (j in 1..M-1)
11       for (c in 0..3)
12         out[i][j][c] = (b2[c][i][j-1] +
13                         b2[c][i][j] +
14                         b2[c][i][j+1])/3
```

Schedule

b_2.after(b_1, c); out.after(b_2, root);
b_1.split(i, N/NUM_NODES, q, i); b_2.split(i, N/NUM_NODES, q, i);
out.split(i, N/NUM_NODES, q, i);
b_1.store_in(t); b_2.store_in($buf_{b2}[c, i, j]$); out.store_in($buf_{out}[i, j, c]$);
b_1.gpu(i,j); b_2.gpu(i,j); out.gpu(i,j)
b_1.distribute(q); b_2.distribute(q); out.distribute(q);

Layer II

// Layer II generated from Layer I using the following schedule

$\{b_1(0, q(node), i(gpu), j(gpu), c, 0) : C_q \land C_n \land C_m \land C_k\}$: $1.5 * img(i, j, c)$
$\{b_2(0, q(node), i(gpu), j(gpu), c, 1) : C_q \land C_n \land C_m \land C_k\}$:
$clamp(b_1(0, q, i, j, c, 0), 0, 255)$
$\{out(1, q(node), i(gpu), j(gpu), c, 0) : C_q \land C_n \land C_{m'} \land C_k\}$:
$(b_2(0, q, i, j-1, c, 0) + b_2(0, q, i, j, c, 1) + b_2(1, q, i, j+1, c, 0))/3$

Layer III

// Same as Layer III in (b) except the mapping of b_1 and b_2 should be replaced with the following

$\{b_1\ (0, q(node), i(gpu), j(gpu), c, 0) \rightarrow t : C_q \land C_n \land C_m \land C_k\}$
$\{b_2\ (0, q(node), i(gpu), j(gpu), c, 1) \rightarrow buf_{b2}[c, i, j] : C_q \land C_n \land C_m \land C_k\}$

Figure 1. Three versions of the motivating example (left) and their equivalent Layer I, II and III representations (right)

to represent transformations on the iteration domain and data layout. Polyhedral sets and relations are described using affine (linear) constraints over loop iterators and program parameters (invariants) and are implemented in Tiramisu using ISL [58]. We use a combination of classical extensions to the polyhedral model in order to support non-affine iteration spaces; these extensions are sufficient and practical for large classes of programs [4, 5].

A typical workflow for using Tiramisu is illustrated in Figure 2. DSL compilers parse input programs and perform domain specific optimizations before translating the DSL program into Layer I of the Tiramisu IR. The first layer of the IR is then transformed to lower layers (Layer II and Layer III), and finally LLVM or other low-level IR is generated.

The three layers of the Tiramisu IR are:

Layer I (Abstract Computation Layer) which specifies the algorithm without specifying the schedule (when and where the computations occur) or how data should be stored in memory (data layout). As there is no notion of data location, values are communicated via explicit producer-consumer relationships.

Layer II (Computation Placement Layer) which 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 passes do not need to perform complicated data-layout transformations. The transformation of Layer I into Layer II is done automatically using scheduling and data layout commands. Examples of the scheduling commands supported in Tiramisu are presented in Table 1.

Layer III (Concrete Computation Layer) which makes the data layout concrete, specifying where intermediate values are stored.

The separation into levels does not force data-layout mapping to occur after scheduling; in Tiramisu, the user can still

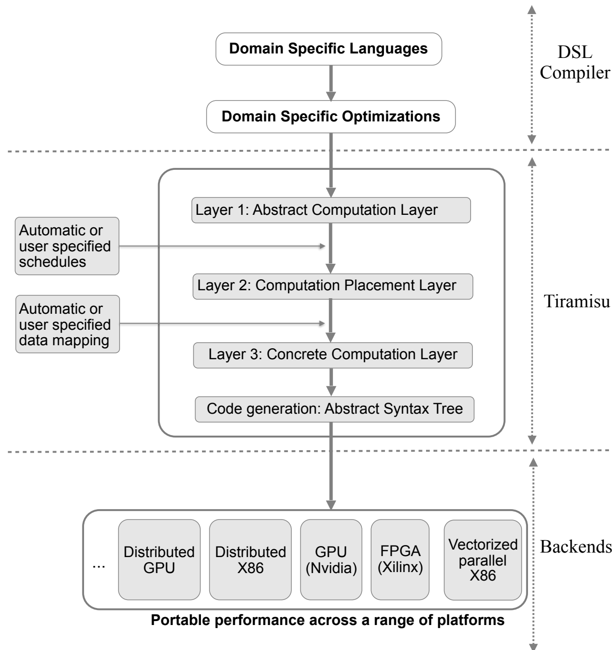

Figure 2. Tiramisu overview

specify data layout before scheduling (to constrain scheduling, for example). The separation ensures that the scheduling phase can safely assume no data-layout transformations are required, greatly simplifying scheduling transformations. If a user requests a transformation that cannot be performed with the specified data layout, Tiramisu will prevent the illegal transformation from occurring, ensuring correctness.

In the following section, we provide more details about the three layers of Tiramisu.

4 The Tiramisu IR

The input to Tiramisu is the Layer I computations and a set of scheduling and data layout commands. Layer II is generated by applying the schedule to Layer I. Commands for buffer allocation, data layout mapping, and communication (between CPU nodes for example) are then added to the Layer II representation; the result constitutes Layer III. An annotated abstract syntax tree (AST) is then generated from Layer III. This AST is traversed to generate the target code.

In this section, we describe in detail the three representations used in Tiramisu. We also describe scheduling via high-level scheduling commands as well as low level scheduling maps. We begin by showing an example.

4.1 An Example in the Three-Layer IR

We first provide an overview of the concepts of polyhedral sets and maps. More details and a formal definition of these concepts are provided in the Appendix.

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

Commands to transform Layer I into Layer II
We assume that C and P are computations

Command	Description
`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
`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).
`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.vectorize(i, v)`	Vectorize the dimension i by a vector size v
`C.unroll(i, v)`	Unroll the dimension i by a factor v
`C.parallelize(i)`	Mark the dimension i as a space dimension (cpu)
`C.distribute(i)`	Mark the dimension i as a space dimension (node)
`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()`	Set the schedule of C, i.e.,a map that transforms Layer I to Layer II
`C.gpu(i0,i1,i2)`	Mark the dimensions i0, i1 and i2 to be executed on the GPU
`C.fpga()`	Generate HLS code for the computation C
`C.pipeline(i)`	Mark the dimension i to be pipelined (FPGA)

Commands to add data mapping to Layer III

Command	Description
`Buffer b(...)`	Declare a buffer b (size, type, ...)
`C.set_access()`	Set the access relation for the computation C
`C.store_in(buff[i0,..])`	Store the result of the computation $C(i0, \ldots)$ in buff[i0, ...]
`C.auto_allocate_map()`	Allocate a buffer for C and map C to it
`C.set_access()`	Map C to a buffer access
`C.storage_fold(i, d)`	Contract the dimension i of the buffer associated to C to make its size d
`create_transfer(...)`	Create a pair of send & receive communication statements
`C.partition(b, type)`	Mark the buffer b to be partitioned in a complete, cyclic or block way (FPGA)

Table 1. Examples of Tiramisu Scheduling Commands

tuples in a set, we describe the set using affine constraints over loop iterators and symbolic constants: $\{S(i,j) : 1 \le i \le 3 \land 1 \le j \le 2\}$ where i and j are the dimensions of the set.

A map is a relation between two integer sets. For example $\{S1(i,j) \rightarrow S2(i+2, j+2) : 1 \le i \le 3 \land 1 \le j \le 2\}$ is a map between tuples in the set S1 and tuples in the set S2 (e.g. the tuple $S1(i,j)$ maps to tuple $S2(i+2, j+2)$). We use the Integer Set Library (ISL) [58] notation for sets and maps.

Figure 1 shows the code for each optimized implementation discussed in the previous section. The original, unoptimized code is shown in Figure 1-(a), with the right side showing the Layer I representation. This Layer I representation is the same for all the code variants, as this layer specifies the computation in a high-level form separate from scheduling.

Each line in Layer I of Figure 1-(a) (right side in the figure) corresponds to a statement in the algorithm (left side of the figure): for example, the first line of Layer I represents the line 5 in Figure 1-(a). The first part of that line[1], which is

```
{b1(i,j,c): 0<=i<N and 0<=j<M and 0<=c<3}
```

specifies the iteration domain of the statement, while the second part, $1.5 * img(i,j,c)$, is the computed expression. The iteration domain is the set of tuples $b_1(i,j,c)$ such that $0 \le i < N \land 0 \le j < M \land 0 \le c < 3$. Computations in Layer I are not ordered. The declaration order does not affect their order of execution, which is specified in Layer II.

[1]The constraints C_n, C_m, and C_k have been expanded inline

Figure 1-(b) shows the first optimized version of the code. The schedule on the right side is the set of scheduling and data layout commands that produce this version of the code. The scheduling commands are presented in Table 1. Layer II is generated automatically by applying these commands to Layer I. Tiramisu provides a large set of high-level scheduling and data layout transformation commands.The Layer II representation is also shown in Figure 1-(b). Computations in Layer II are ordered based on their lexicographical order[2]. The set

```
{b1(i (cpu), 0, j, c, 0):  0<=i<N and 0<=j<M and 0<=c<3}
```

in the example, is an ordered set of computations. The tag *(cpu)* for the i dimension indicates that this dimension is a space dimension and that each ith iteration is mapped to the ith CPU. In Layer II, the computation order is controlled by a total ordering of these tuples.

Layer III in Figure 1-(b) adds data layout mapping to Layer II, concretizing where each computation is stored (memory buffers and scalars are also declared in this layer). In the example, the data mapping

```
{b2(0,i(cpu),j,c,1) → buf_b2[i,j,c]:
    0<=i<N and 0<=j<M and 0<=c<3}
```

indicates that the result of the computation $b_2(0, i(cpu), j, c, 1)$ is stored in the array element $buf_{b2}[i, j, c]$. Data mapping in Tiramisu is an affine relation that maps a computation from Layer II to a buffer element; scalars are single-element buffers. Tiramisu allows the expression of any data-layout mapping that can be expressed as an affine relation (examples provided in Section 12.3). For brevity, the declaration of buffers, their types, their allocation (including when and where they are allocated), are all omitted from the example, but such information must be specified for correct code generation.

4.2 Layer I: Abstract Computation Layer

The first layer defines abstract computations, which are not yet scheduled or mapped to memory. Each *computation* represents an expression that should be computed.

As an example, the following code

```
1 for (i in 0..4)
2  for (j in 0..4)
3    if (i < j && i != 2)
4      A[i][j] = cos(i);
```

can be represented in Layer I as

```
{A(i,j): 0<=i<4 and 0<=j<4 and i<j and i!= 2}: cos(i)
```

though it is important to remember that this representation, unlike the pseudocode above, does not necessarily store results to memory locations. $A(i, j)$ is the computation, while the constraints over i and j define the iteration domain. The second part, $cos(i)$, is the computed expression.

[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 $S0(1, i, 0)$

Computations in Layer I are in Static Single Assignment (SSA) form [18]; each computation is defined only once (we use the ϕ operator to deal with branches as in classical SSA).

Reductions and Updates Reductions and updates do not fit naturally in the memory-independent model used within Tiramisu, and thus we treat them as a special case. To implement algorithms that perform a reduction or update a variable (a histogram for example), we declare a new computation for each update. These computations will all be mapped to the same buffer in Layer III. For example, a dense matrix multiplication, which has a reduction, is represented in Layer I as follows:

```
{c0(i,j):  0<=i<N and 0<=j<N}: 0
{c1(i,j,k): 0<=i<N and 0<=j<N and 1<=k<N}:
    ϕ(c0(i,j), c1(i,j,k-1)) + A(i,k) * B(k,j)
```

Since $c1(i, j, k)$ needs to read the results of the computations $c0(i, j)$ and $c1(i, j, k - 1)$, we use the ϕ node to merge them into one expression $\phi(c0(i, j), c1(i, j, k - 1))$ (although the use of ϕ nodes in this case can be avoided, in the general case the use of ϕ nodes is necessary to support cases such as the definition of computations within data-dependent conditions).

Support for Non-Static-Affine Iteration Spaces Tiramisu can represent non-static-affine code. In particular, Tiramisu can represent non-static-affine array accesses, `while` loops, non-static-affine loop bounds and non-static-affine conditionals. Tiramisu treats any non-static-affine conditional in a way similar to [4]: the conditional is represented as a single macro-statement together with its body (i.e., as a statement encapsulating both the control and the body). `while` loops and loops with non-static-affine bounds are handled in a way similar to [6].

4.3 Layer II: Computation Placement Layer

The computation placement layer describes when and where each computation is computed. Unlike computations in the first layer, computations in this layer are ordered (specifying when) and are assigned to a particular processor (specifying where). This order is dictated by *space dimensions* and *time dimensions*. Space dimensions specify on which processor computations should be executed; such dimensions are not relevant for determining the order of execution. On the other hand, time dimensions specify the order of execution relative to other computations. The order of execution of computations is determined by the lexicographical ordering of the dimensions. Space dimensions 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
`gpu_thread_X`	the dimension runs on a gpu thread (dimension X where X=0 for outermost and 2 for innermost). Similar tags are used for blocks.

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

In addition to processor type, tags can optionally include one of the following dimension properties:

`vec(s)`	vectorize the dimension (*s* is the vector length)
`unroll`	unroll the dimension
`pipeline`	pipeline the dimension (FPGA only)

Computations mapped to the same processor are ordered by projecting the computation set onto the time dimensions and comparing their lexicographical order, without considering the name of the computation, since all computations in this layer are in the same time-space domain.

4.4 Layer III: Concrete Computation Layer

The concrete computation 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 computations from Layer II to array elements read or written by those computations. Scalars are treated as single-element arrays. For each buffer, an allocation statement is created, specifying the type of the buffer (or scalar) and its size, and is scheduled by being mapped to the time-space domain. 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 $c(j, i)$.

4.5 Generating Layer II and III from Layer I

Transforming the first layer into the second layer is usually done using an affine relation called a *scheduling map*. This maps each computation in the first layer into a particular position in time-space. Composing many transformations can be done simply by composing different scheduling maps.

4.5.1 Scheduling Maps

Affine transformations including loop tiling, skewing, loop fusion, distribution, splitting, reordering, and many others can be expressed as an affine map that maps computations from Layer I into the time-space domain in Layer II. A scheduling map takes as input the iteration domain from Layer I and transforms it into a new set that represents the computation in the time-space domain. For example, suppose we want to tile the following computation (which is in Layer I) into 16×16 tiles and parallelize the outermost loop:

```
{C(i,j): 0<=i<N and 0<=j<N}: A(i,j) + B(i,j)
```

To do so, we provide the following scheduling map to Tiramisu:

```
{C(i,j)->C(i1(cpu),j1,i2,j2):i1=floor(i/16)
and i2=i%16 and j1=floor(j/16) and j2=j%16 and 0<=i<N and
     0<=j<N}
```

which will produce the following set in Layer II:

```
{C(i1(cpu),j1,i2,j2): i1=floor(i/16) and i2=i% 16
   and j1=floor(j/16) and j2=j%16 and 0<=i<N and 0<=j<N}:
   A(i1*16+i2, j1*16+j2) * B(i1*16+i2, j1*16+j2)
```

4.5.2 High Level Scheduling Commands

Tiramisu provides a set of predefined scheduling maps for common affine loop nest transformations. Table 1, presented previously, shows examples of Tiramisu high-level scheduling commands. These commands are similar to those in Halide [49] and ChiLL [13]. The high-level scheduling commands in Tiramisu provide an easy-to-use interface for advanced loop nest transformations in a composable way, while still enabling advanced users to provide their own low-level scheduling maps to modify the space-time mapping for scheduling not covered by typical compiler transformations.

4.5.3 Checking the Validity of Schedules

In order to check the validity of transformations, we first compute the dependences of the input program, then we check the validity of transformations using violated dependence analysis [57].

5 Generating Tiramisu from DSLs

To demonstrate the utility of Tiramisu, we integrated it into two DSL compilers: Halide [49] and Julia [7]. A DSL compiler that uses Tiramisu must generate three pieces of information:

- Layer I, which describes the algorithm;
- A scheduling map or a scheduling command;
- Commands declaring the buffers and mapping the computations to buffer elements.

Tiramisu takes the three inputs and generates Layer II and Layer III automatically, and then generates an Abstract Syntax Tree (AST) from Layer III. The AST is traversed to generate target code (LLVM IR, Nvidia PTX, ...).

5.1 Halide to Tiramisu

Halide [49] is an industrial-quality DSL for image processing. We generate Tiramisu IR from Halide by mapping a Halide `Func`, which is equivalent to a statement in a loop nest, directly to a Tiramisu computation (Layer I). Reductions, which update the same function, are mapped to Tiramisu computations as described in Section 4.2. Halide scheduling directives, such as tiling, splitting, reordering, parallelization, vectorization, etc., are directly mapped to the equivalent high level set of scheduling commands defined in Tiramisu. Finally, we map computations to buffer elements using the default Halide mappings, while allowing Halide scheduling commands that control data mappings to perform equivalent transformations for the Layer III representation. The rest of the code generation to low-level executable code takes place within Tiramisu.

5.2 Julia to Tiramisu

Julia is a high-level dynamically-typed programming language designed for numerical computing. However, in contrast to Halide, Julia is more general: it supports while loops and recurrent computations and is memory-based (i.e., it uses variables unlike Halide which defines pure functions mostly). We extend Julia with a set of scheduling directives and function annotations. Functions annotated with the `@acc` macro are optimized with Tiramisu.

We generate Tiramisu from the low-level Julia IR (which is in SSA form) by translating each statement in the Julia IR into a computation in Tiramisu. This Julia low-level IR does not have high level control flow (it only has `gotos`); thus we change the compilation flow of Julia to annotate the low-level IR with information about the original control flow of the program. We use the annotations to recover the control flow and generate the iteration domain of each computation. Although Julia has another high level IR that has control flow information, we cannot use that IR because it lacks the necessary data type annotations.

We transform the memory-based Julia IR into the producer-consumer Tiramisu IR using classical array expansion techniques [22, 43, 44]. The goal here is to extract the data-flow representation of the code. The user is then free to change the data layout of computations using high level data-layout directives.

6 Generating Code for Different Platforms

Generating code from Layer III (an ordered set of computations) amounts to generating nested loops (AST) that visit each computation in the set, once and only once, while following the lexicographical ordering between integer tuples. Array accesses are generated from the maps that describe the data mapping. The Tiramisu code generator (which uses the ISL [58] library for code generation) takes Layer III as input and generates an AST as output. The AST is then traversed to generate lower level code targeting different hardware architectures.

6.1 Multicore CPU

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

6.2 GPU

For GPU code generation, data copy commands are provided in Layer III of Tiramisu. These commands are translated into the equivalent data copy calls in the lowered 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.

```
1  // Layer I
2  {bx(y,x): 0<=y<rows and 0<=x<cols} :
3          (in(y,x) + in(y,x+1) + in(y,x+2))/3);
4  {by(y,x): 0<=y<rows and 0<=x<cols} :
5          (bx(y,x) + bx(y+1,x) + bx(y+2,x))/3);
6  // Layer II
7  bx.split(y,chunk_sz,y1,y2); by.split(y,chunk_sz,y1,y2);
8  // Layer III
9  comm_prop blk({BLOCK}), ablk({ASYNC,BLOCK});
10 send_recv xfer = create_transfer("{(q,y,x): 1<=q<N-1 and 0<=
        y<2 and 0<=x<cols}","{(q,y,x): 0<=q<N-2 and 0<=y<2 and
          0<=x<cols}",q-1,q,ablk,blk,bx(y,x));
11 bx.distribute(y1); by.distribute(y1);
12 xfer.s->distribute(q); xfer.r->distribute(q);
```

Figure 3. Tiramisu pseudocode for a 3x3 distributed blur

6.3 Distributed Memory Systems

Tiramisu utilizes MPI to generate code for distributed memory systems. Figure 3 shows Tiramisu pseudocode for a 3x3 distributed box blur. Lines 2 and 4 define the blur computation. This code remains the same regardless of whether we use a shared memory or distributed memory back-end.

For this example, we want to distribute the computation such that each MPI *rank* (process) operates on contiguous rows of the input data. Each rank gets `chunk_sz` rows. On line 7, the outer loop is split by `chunk_sz`. The resulting inner loop ranges over the rows in the chunk, and the outer loop ranges over the number of MPI ranks we want to use.

Lines 9 and 10 deal with communication. We assume that our image data is already distributed, thus only boundary rows need to be communicated among adjacent ranks. Line 9 defines two communication types, which will be used to select the appropriate MPI function. `blk` represents a blocking call, and `ablk` represents an asynchronous, blocking call. We use two-sided communication in Tiramisu, meaning communication is done with pairs of *send* and *receive* operations. The actual transfer is defined by `create_transfer` on line 10, which takes as input the send and receive iteration domains, the source and destination ranks, the communication types for send and receive, and the access into the producer for the send.

Line 11 tags dimension `y1` of `bx` and `by` as distributed, and line 12 tags dimension `q` of the `send` and `receive` as distributed. 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) {...}
```

All of the other scheduling commands in Tiramisu can be composed with transfers and distributed loops, as long as the composition is semantically correct. This means we can do everything from basic transformations (e.g. tiling a

```
1 input.copy_to_device();
2 bx.gpu(y2,x); by.gpu(y2,x);
3 output.copy_to_host();
```

Figure 4. Additional Tiramisu commands needed to generate a 3x3 distributed GPU box blur

transfer) to more advanced transformations (e.g. specializing a distributed computation based on rank).

GPU scheduling can also be composed with distribution, allowing programs to execute in either a multi-GPU or heterogeneous CPU-GPU environment. Only a few extra scheduling commands need to be added to distributed Tiramisu code to enable the use of GPU. Figure 4 shows the four additional scheduling commands needed to convert the distributed box blur code in Figure 3 to distributed GPU code. Lines 1 and 3 copy data from the host (CPU) to the device (GPU) and from the device to the host, respectively. Line 2 tags the computations to run on the GPU. The resulting code can be used to distribute the box blur computation across multiple GPUs that reside on different nodes. As with CPU distribution, we use MPI to control the inter-node communication.

6.4 FPGA

Tiramisu relies on FROST [20] to generate code for FPGAs. FROST is a common back-end for the hardware acceleration of DSLs on FPGA. It exposes an IR that DSLs can target, as well as a high level scheduling language to express FPGA-specific optimizations.

We integrated FROST within Tiramisu, enabling us to target FPGAs. We use Tiramisu to perform loop nest transformations that are necessary to prepare the code for lowering to FPGA, while FROST focuses on the actual lowering to the target High-Level Synthesis (HLS) toolchain. For example, in order to vectorize a loop, Tiramisu first splits the loop so that the innermost loop has a fixed number of iterations and then tags that loop for later vectorization by FROST. FROST then performs the actual vectorization of both loop and input/output buffers.

The output of FROST is a C++ implementation of the input code suitable for HLS tools, like Xilinx Vivado HLS [33]. Finally, FROST leverages Xilinx SDAccel to synthesize the resulting FPGA design and produce the bitstream file for execution on actual hardware.

7 Evaluation

We performed the evaluation on a cluster of dual-socket machines with two 24-core Intel Xeon E5-2680 v3 CPUs running at 2.50GHz running Ubuntu 14.04.5 LTS, with an Infiniband interconnect.

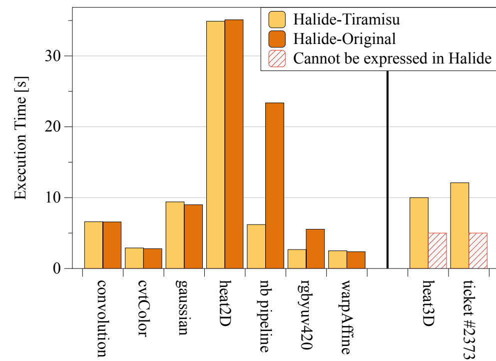

Figure 5. Execution Time for Tiramisu and Halide (s)

7.1 Halide to Tiramisu

To evaluate the integration of Tiramisu with Halide, we used the following benchmarks: `cvtColor`, which converts RGB to grayscale; `convolution`, a simple 2D convolution; `gaussian`, which performs a gaussian blur; `warpAffine`, which does affine warping on an image; `heat2D`, a simulation of the 2D heat equation; `nb pipeline`, a synthetic pipeline that computes two outputs from the same image, a negative and a brightened image; `rgbyuv420`, an image conversion from RGB to YUV420; `heat3D`, the 3D heat equation with timestepping; and `ticket #2373`, a code snippet from a bug filed against Halide where the bounds inference is over-approximated, leading the generated code to fail in execution.

Figure 5 compares the execution time of code generated by Halide and Halide-Tiramisu. In five of the benchmarks (namely `convolution`, `cvtColor`, `gaussian`, `heat2D`, and `warpAffine`), the performance of the code generated by Halide-Tiramisu matches the performance of Halide. We use the same schedule for both implementations.

Two of the other benchmarks , `heat3D` and `ticket #2373`, cannot be implemented in Halide. The following is an example of a recurrent filter extracted from [12], a compiler designed to support recurrent filters.

```
heat3d(x,y,z,0) = a*in(x, y, z) +
                  b*(in(x+1, y, z) + in(x-1, y, z)+
                     in(x, y+1, z) + in(x, y-1, z) +
                     in(x, y, z+1) + in(x, y, z-1));

heat3d(x,y,z,t) = a*heat3d(x, y, z, time.x-1) +
        b*(heat3d(x+1, y, z, t-1) + heat3d(x-1, y, z, t-1)+
           heat3d(x, y+1, z, t-1) + heat3d(x, y-1, z, t-1) +
           heat3d(x, y, z+1, t-1) + heat3d(x, y, z-1, t-1));
```

This code cannot be implemented in Halide because it contains a cyclic dependence graph due to the loop over timesteps while the Halide compiler assumes that the dependence graph is a DAG (directed acyclic graph). This limitation is mainly because it is difficult to prove the legality of optimizations in an interval-based representation in the case of a cyclic dependence graph. This is not the case for Tiramisu, which relies on a precise dependence analysis [23] and on checking the legality of transformations using the polyhedral

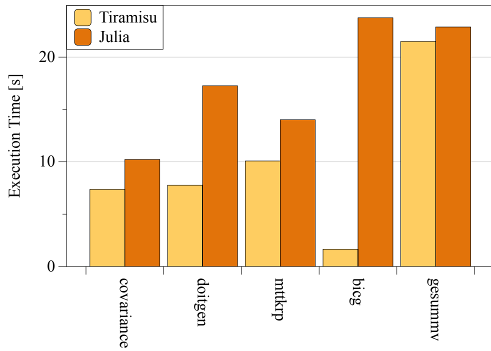

Figure 6. Execution Time for Tiramisu and Julia (s)

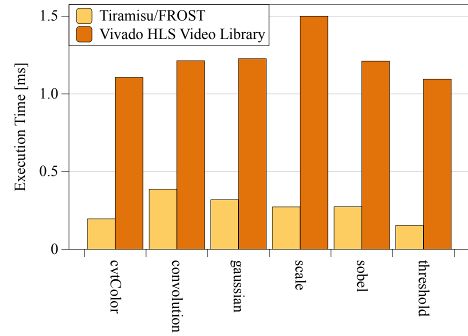

Figure 7. Execution Time for Tiramisu/FROST and Vivado HLS Video Library (ms)

model [55] to decide whether a transformation can be performed. 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 on non-rectangular iteration spaces. Tiramisu can handle this case naturally since it relies on a polyhedral based model where sets can include any affine constraint in addition to the min and max bounds. These examples show that the model exposed by Tiramisu naturally supports more complicated code patterns than an advanced, mature DSL compiler.

For `nb pipeline` and `rgbyuv420`, the code generated from Halide-Tiramisu achieves a 4× speedup over the code generated from Halide. This is primarily due to fusion. In both cases, Tiramisu can fuse multiple loops into one loop which enhances data locality; loop fusion is currently unsupported in Halide. This is another case that demonstrates that the expressiveness of the polyhedral-based representation in Tiramisu allows the framework to naturally perform certain iteration domain transformations that are difficult in other models.

7.2 Julia to Tiramisu

We used the following benchmarks to evaluate the integration of Tiramisu within Julia: `bicg`, a biconjugate gradient method; `doitgen`, a multiresolution analysis kernel; `mttkrp`, the matricized tensor times Khatri-Rao product; `covariance`, which performs a covariance computation; and `gesummv`, which is summed matrix-vector multiplications. For a fair comparison, the Julia code was tagged with the `inbounds` macro to remove any boundary checks on buffer accesses.

Figure 6 shows the execution time of code generated by Julia-Tiramisu compared to Julia without Tiramisu. The speedups of Julia-Tiramisu in `covariance`, `doitgen`, `mttkrp` and `bicg` are mainly due to the improved data locality obtained after tiling using Tiramisu, which is not possible in Julia.

7.3 Evaluating Backends

7.3.1 FPGA Backend

We evaluate the FPGA backend in Tiramisu using 6 image processing kernels: `convolution`, `cvtColor`, `gaussian`, `scale`, `sobel`, and `threshold`. We chose these kernels because they are already implemented in the Vivado HLS Video Library [32], which implements several OpenCV functions for FPGA. We compare the execution time of code generated from Tiramisu with the codes extracted from the Vivado HLS Video Library. These codes are synthesized using the Xilinx SDAccel 2016.4 toolchain at 200MHz and ran on a ADM-PCIE-7V3 board by Alpha Data (powered by a Xilinx Virtex 7 FPGA). For all the kernels, we use a 512 × 384 RGB image, except for the `threshold` kernel, which takes as input a single channel image.

The HLS Video Library kernels expect the input image to be arranged with channels as the innermost dimension of the image. The accelerator on the FPGA receives a stream of pixels from the off-chip memory, and processes each channel of the pixel completely in parallel.

While the HLS Video Library parallelizes only the channel dimension, the flexibility of the Tiramisu scheduling commands allowed us to explore other alternatives including the parallelization over the width dimension of the image leading to better performance (at the expense of more FPGA resources). Indeed, while the Video Library performs, at most, three computations in parallel (on the channels), the code generated from Tiramisu can perform, at most, sixty-four computations in parallel, in the case of a 512-bit vectorization of the input/output buffers for a 8-bit image.

Figure 7 shows the execution time of Tiramisu/FROST and the Vivado HLS Video Library. Tiramisu with FROST outperformed the Video Library implementations by at least 3×. For each kernel, we used Tiramisu to arrange the input image and split the innermost loop to prepare for vectorization (we applied vectorization to both input/output buffers). We also applied loop pipelining and array partitioning (for `convolution`, `gaussian` and `sobel`).

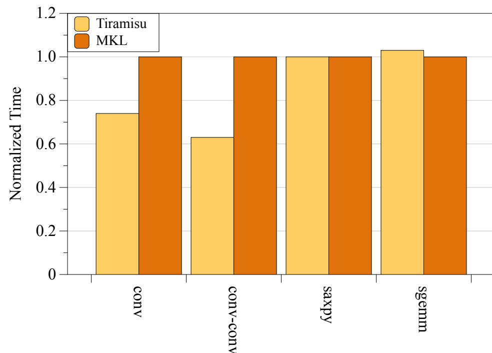

Figure 8. TIRAMISU compared to Intel MKL

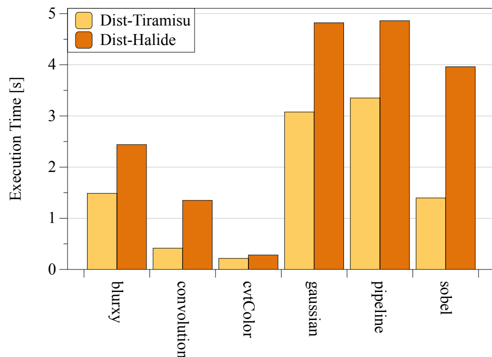

Figure 9. Execution time of distributed TIRAMISU and distributed Halide across 16 nodes (s)

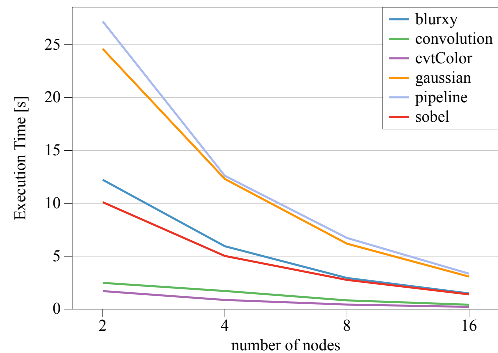

Figure 10. Execution time of distributed TIRAMISU for 2, 4, 8, and 16 nodes (s)

7.4 Generating BLAS `sgemm` using TIRAMISU

To evaluate the performance of TIRAMISU on an extreme case, we used TIRAMISU to generate the BLAS generalized matrix multiplication (`sgemm`) which computes $C = \alpha AB + \beta C$. The `sgemm` implementation in the Intel MKL library is known to be one of the most highly hand-optimized implementations for Intel CPU architectures. We used a large set of optimizations including three-dimensional L2 and L3 blocking, fusion of the computation of $T = \alpha AB$ and $C = T + \beta C$ into a single loop, vectorization, unrolling, array packing (as described in [25]), register blocking, separation of full and partial tiles (which is crucial to enable vectorization, unrolling, and reduce control overhead). We also tuned the tile size and unrolling factor for the machine on which we run our experiments. The resulting kernel matches the Intel MKL implementation as shown in 8. The TIRAMISU implementation of saxpy, convolution and two fused convolutions all outperform or match the Intel MKL implementation (lower is better).

7.4.1 Distributed and GPU Backend

For the TIRAMISU distributed backend, we used 6 kernels for evaluation: `blurxy`, `sobel`, `convolution`, `gaussian`, `pipeline`, and `cvtColor` (we chose these kernels because these are already implemented in the distributed Halide compiler [21]). We assume that the data is already distributed across the nodes by rows. Of these benchmarks, `pipeline`, and `cvtColor` do not require any communication; the other four require communication due to overlapping boundary regions in the distributed data. For the distributed CPU-only tests, we use the MVAPICH2 2.0 [31] implementation of MPI.

Figure 9 compares the execution time of distributed TIRAMISU and distributed Halide on 16 nodes for each of the kernels. TIRAMISU is faster than distributed Halide in each case. For the kernels involving communication, code generated by distributed Halide has two problems compared to TIRAMISU: (1) It overestimates the amount of data it needs to send; (2) It unnecessarily packs together contiguous data into a separate buffer before sending.

Figure 10 shows the execution time of the kernels with distributed TIRAMISU when running on 2, 4, 8, and 16 nodes. As expected, execution time decreases for these kernels as the number of nodes increases.

7.4.2 Putting it All Together

As a final experiment, we ran a modified version of the *cvtColor* kernel in a distributed GPU configuration and compared it with a distributed CPU configuration. For this experiment, we ran on a small cluster of 4 nodes, each consisting of a single Nvidia K40 GPU and a 12-core Intel Xeon E5-2695 v2 CPU clocked at 2.40GHz. We used OpenMPI 1.6.5 [46] as our MPI implementation.

Figure 11 shows the results of this experiment. The back row shows the results for running the *cvtColor* kernel on one node, using either 1 core, 10 cores, or 1 GPU. As expected, 10 cores is better than 1 core and the GPU outperforms the CPU. The front row shows the same configuration, expect distributed across 4 nodes. So, from left-to-right, the columns of the front row represent a total of 4 cores, then 40 cores, and then 4 GPUs. As with the the single node performance, 40 cores is better than 4 cores, and 4 GPUs is better than the CPUs.

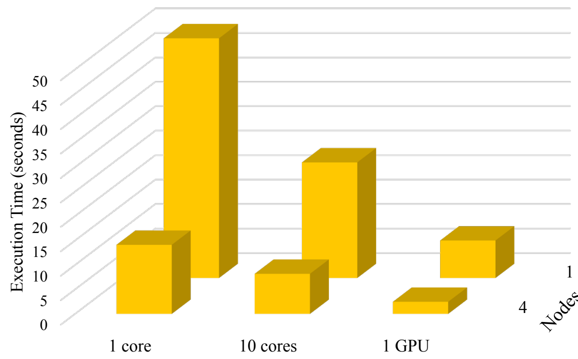

Figure 11. Results for either CPU or GPU running on a single node (back row), and distributed across 4 nodes (front row).

7.5 Evaluation Summary

Overall, the experiments demonstrated the use of Tiramisu as an IR and optimization framework for two DSLs and multiple backends. We show that Tiramisu is expressive: it allows both Halide and Julia to perform new optimizations and allows Halide to express new algorithms. The experiments also show that Tiramisu is suitable for targeting multiple hardware architectures, such as multicore, GPUs, distributed systems, and FPGA. And thanks to its flexible scheduling commands, it can generate highly optimized code for a variety of of architectures and algorithms.

8 Related Work

8.1 High Performance DSL Compilers

High performance DSL compilers such as Halide [50], Diderot [14], Simit [35], Polymage[45], OoLaLa [41] and others build custom compilation pipelines for specific domains such as image processing or linear algebra. These compilers have shown that it is possible to obtain high performance by applying domain-specific optimizations in the course of compilation. However, such compilers map DSL code directly to the target hardware, sometimes using a low-level compiler framework like LLVM. Our goal in this work is to build a more generic framework and intermediate representation that can be used by domain-specific language compilers in place of ad-hoc re-implementations of compute and data transformations.

8.2 DSL IRs and Optimization Frameworks

Delite [11] is a generic framework for building DSL compilers using Lightweight Modular Staging (LMS) [51], a technique for embedding code generators and compilers in Scala. Delite exposes several parallel computation patterns that DSLs can use to express computation; however, it has no facilities for advanced loop nest transformations. We therefore believe that generic DSL frameworks like Delite can benefit from using Tiramisu.

PENCIL [3, 4] is another generic DSL IR and automatic optimization framework which uses a polyhedral representation internally. It is a subset of C99 with additional constructs to help parallelizing compilers perform more accurate static analyses, and as a result generate more efficient code. The Pluto [9] automatic scheduling algorithm used within PENCIL can be integrated seamlessly in Tiramisu on top of the first layer. The main difference between PENCIL and Tiramisu is that Tiramisu separates computation, schedule, and data layout. In contrast, the PENCIL IR is a subset of C99 with arrays, scalar variables, etc. and thus successful parallelization and optimization sometimes requires data-layout transformations such as expansion and privatization which are not necessary in Tiramisu.

CHiLL [13, 27] is a polyhedral based compiler framework for Fortran that allows users to express a composition of high-level transformations such as tiling and unrolling, which the system performs, freeing users from having to implement them. URUK [24] is a similar framework that also uses a polyhedral representation. Other systems such as POET [61] parametrize loop transformations with a language-agnostic, purely-syntactic transformation system. These frameworks require working with concrete data layouts, in contrast to Tiramisu that does not have a concrete data layout in its first layer.

Darkroom [29] is a language and compiler for image processing pipelines. Darkoom compiles the input programs into optimized line-buffered pipelines (it relies on an ILP solver to optimally schedule the pipelines), and then synthesizes them for ASIC, FPGA, or CPU. Similarly, [47] presents an extension to Halide to hardware accelerate applications on FPGA. The authors implemented additional scheduling commands to define and control the code generated for FPGA. These works are designed for image processing applications only, while Tiramisu with FROST can support also other types of computations (e.g. linear algebra).

8.3 Data-layout Transformation

Techniques such as scalar and array expansion remove false dependencies, enabling loop nest transformation and parallelization [22, 34]. Expansion increases dimensionality to create private copies of data for each loop iteration. In Tiramisu, computations are single assignment, and thus are fully expanded, obviating the need for privatization.

A family of array contraction techniques attempts to reduce the memory footprint without constraining loop nest transformations [19, 37, 48]: the compiler performs a maximal expansion before applying loop transformations, and then attempts to contract the expanded arrays. Tiramisu simplifies this process, since maximal expansion is not needed. This is similar to Halide [49] where computations are mapped

by default to fully expanded arrays and then a compiler pass performs storage folding to contract arrays.

Several alternative approaches try to constrain expansion. Maximal static expansion (MSE) restricts the elimination of dependencies to the situations where the data flow can be captured accurately at compilation time [16]. It is important when generalizing array dependence analyses and loop transformations to dynamic control flow, and it can be combined with array contraction [15]. A priori constraints on memory footprint can also be enforced, up to linear volume approximations [53], and more generally, trade-offs between parallelism and storage allocation can be explored. These techniques can also be applied in Tiramisu to constrain the schedule.

Data layout transformations for specific dependence patterns using the polyhedral model have been used to eliminate SIMD intra-vector operations [30] and for enhancing cache locality in non-uniform cache access (NUCA) architectures [40]. These kinds of transformations can be easily implemented in Tiramisu.

8.4 Functional IRs and Data Layout Transformations

The NOVA functional language [17] was designed to be used as an IR for DSL compilers. It is a polymorphic, statically-typed functional language with a suite of higher-order functions such as map, reduce and scan that are used to express parallelism. Although the NOVA IR does not represent memory explicitly, it does not provide any framework for advanced loop nest transformations. For example, only map functions that iterate over the same ranges can be fused. Iteration space transformations such as skewing are not addressed. Transformations such as fusion are done at the function level. Tiramisu provides an IR that allows advanced iteration space transformations while still separating the algorithm from the schedule and the data layout.

Most functional languages do not expose notions of memory layout to programmers. Instead, programmers rely on profiling tools to characterize data movement [52] or design algorithms around models of memory traffic for such programming languages [8]. In contrast, Tiramisu enables writing the algorithm in a functional manner while separately dealing with data layout and computation scheduling.

9 Conclusion

In this paper we introduce Tiramisu, a middle-end compiler for domain specific languages that separates the algorithm, the schedule and the data layout in a three-layer intermediate representation. Tiramisu supports backend code generation for multicore CPUs, GPUs, FPGAs, and distributed systems, as well as machines that contain any combination of these architectures.

Tiramisu is designed so most DSLs can use high-level scheduling and data mapping constructs to control the lowering from the algorithm to the backend, cross-platform code. In addition, the underlying representations are accessible to advanced users that wish to implement new optimizations and transformations.

We evaluate Tiramisu by creating a new middle-end for the Halide and Julia compilers, targeting a variety of backends. We also demonstrate transformations made possible by Tiramisu increasing performance by up to 4× in Halide and 16× in Julia and demonstrate that Tiramisu can generate very fast code matching one of the most hand optimized kernels (Intel MKL gemm).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). http://arxiv.org/abs/1603.04467

[2] Martin Sandve Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.* 40, 2 (2014), 9.

[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] M.-W. Benabderrahmane, L.-N. Pouchet, Albert Cohen, and Cedric Bastoul. 2010. The Polyhedral Model Is More Widely Applicable Than You Think. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'10) (LNCS)*. Springer-Verlag, Paphos, Cyprus.

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

[8] Guy E. Blelloch and Robert Harper. 2013. Cache and I/O Efficent Functional Algorithms. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2429069.2429077

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

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Daniel Firestone, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, et al. 2017. Configurable Clouds. *IEEE Micro* 37, 3 (2017), 52–61.

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

[12] Gaurav Chaurasia, Jonathan Ragan-Kelley, Sylvain Paris, George Drettakis, and Fredo Durand. 2015. Compiling high performance recursive filters. In *Proceedings of the 7th Conference on High-Performance Graphics*. ACM, 85–94.

[13] 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.

[14] Charisee Chiw, Gordon Kindlmann, John Reppy, Lamont Samuels, and Nick Seltzer. 2012. Diderot: A Parallel DSL for Image Analysis and Visualization. In *PLDI*.

[15] Albert Cohen. 1999. Parallelization via constrained storage mapping optimization. In *Intl. Symp. on High Performance Computing*, Kazuki Joe Akira Fukuda, Shinji Tomita, and Constantine Polychronopoulos (Eds.). LNCS, Vol. 1615. Springer-Verlag, 83–94.

[16] Albert Cohen and Vincent Lefebvre. 1998. Optimization of Storage Mappings for Parallel Programs. *in Europar 99, number 1685 in LNCS* (1998), 375—382.

[17] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)*. ACM, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/2627373.2627375

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[19] 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

[20] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco Domenico Santambrogio. 2017. A Common Backend for Hardware Acceleration on FPGA. In *35th IEEE International Conference on Computer Design (ICCD'17)*.

[21] 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.

[22] 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

[23] 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

[24] 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.

[25] 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

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

[27] 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.

[28] Mary W Hall, Saman P Amarasinghe, Brian R Murphy, Shih-Wei Liao, and Monica S Lam. 1995. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. IEEE, 49–49.

[29] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. 2014. Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.* 33, 4, Article 144 (July 2014), 11 pages. https://doi.org/10.1145/2601097.2601174

[30] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. Data Layout Transformation for Stencil Computations on Short SIMD Architectures. In *ETAPS*

International Conference on Compiler Construction (CC'11). Springer Verlag, SaarbrÃČÂijcken, Germany, 225–245.
- [31] 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.
- [32] Xilinx Inc. 2015. HLS Video Library. http://www.wiki.xilinx.com/HLS+Video+Library. (April 2015).
- [33] Xilinx Inc. 2017. Vivado HLx Editions. https://www.xilinx.com/products/design-tools/vivado.html. (October 2017).
- [34] Ken Kennedy and John R. Allen. 2002. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. http://portal.acm.org/citation.cfm?id=502981
- [35] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (May 2016), 21 pages. https://doi.org/10.1145/2866569
- [36] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
- [37] 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
- [38] 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
- [39] Xiangke Liao, Liquan Xiao, Canqun Yang, and Yutong Lu. 2014. MilkyWay-2 supercomputer: system and application. *Frontiers of Computer Science* 8, 3 (2014), 345–356.
- [40] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. 2009. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques*. 348–357.
- [41] Mikel Luján, T. L. Freeman, and John R. Gurd. 2000. OoLALA: An Object Oriented Analysis and Design of Numerical Linear Algebra. In *OOPSLA*. 229–252.
- [42] 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.
- [43] 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
- [44] Samuel Midkiff. 2012. *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan & Claypool Publishers.
- [45] 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
- [46] MPI Open. [n. d.]. Version 1.6. 5, Open MPI Software. ([n. d.]).
- [47] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. Archit. Code Optim.* 14, 3, Article 26 (Aug. 2017), 25 pages. https://doi.org/10.1145/3107953
- [48] 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.
- [49] 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.
- [50] 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.
- [51] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314
- [52] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. *SIGPLAN Not.* 43, 9 (Sept. 2008), 253–264. https://doi.org/10.1145/1411203.1411240
- [53] 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.*
- [54] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan Notices* 44, 6 (2009), 177–187.
- [55] Konrad Trifunovic, Albert Cohen, Razya Ladelski, and Feng Li. 2011. Elimination of Memory-Based Dependences for Loop-Nest Optimization and Parallelization. In *3rd GCC Research Opportunities Workshop*. Chamonix, France.
- [56] 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.
- [57] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. 2006. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*. ACM, Cairns, Queensland, Australia, 335–344. https://doi.org/10.1145/1183401.1183448
- [58] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *ICMS*, Vol. 6327. 299–302.
- [59] 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.
- [60] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. 2011. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications* 182, 1 (2011), 266–269.
- [61] 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

10 Notation and Definitions

10.1 Presburger formula

We use an EBNF (Extended Backus-Naur Form) grammar to define Presburger formulas.

<formula> ← <formula> ∧ <formula>
| <formula> ∨ <formula>
| ¬<formula> | ∃<var>.<formula>
| ∀<var>.<formula> | <atom>
<atom> ← <term><relop><term>
<term> ← <numeral> | <term> + <term>
| − <term>
| <numeral> ∗ <term> | <var>
<relop> ← < | ≤ | = | > | ≥
<var> ← x | y | z | . . .
<numeral> ← 0 | 1 | 2 | . . .

Note that <numeral> ∗ <term> is not a general multiplication operator; it is a shortcut for <term> + · · · + <term>.

Presburger arithmetic is used mainly because it is a decidable arithmetic. That is, there exists an algorithm which decides whether an arbitrary Presburger formula is true (valid) or not, which is important for many polyhedral operations.

10.2 Quasi-Affine Constraints

A *quasi-affine constraint* is a constraint over integer values and integer variables involving only the operators `+`, `-`, `×`, `/`, `mod`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=`, and the ternary `?:` operator, where the second argument of `/` and `mod` must be a (positive) integer literal, and where at least one of the arguments of `×` must be a constant expression. An example of a quasi-affine constraint for a statement in a loop nest is $10 \times i + j + n > 0$, where i and j are loop iterators and n is a *symbolic constant* (i.e., a variable that has an unknown but fixed value for the duration of an execution). An example of a non-quasi-affine constraint is $i \times i > 0$, because we require one of the arguments be a constant.

11 Integer Sets

An *integer set* is a set of integer tuples from $\mathbb{Z}^d$ that can be specified using affine constraints. d is the dimensionality of the set (the number of integers in each tuple) and a d-tuple is represented as $(a_1, a_2, \ldots, a_d)$. 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 integer tuples of the set, we describe the set using affine constraints:

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

where i and j are the dimensions of the set. The tuples of a set can optionally have a common name, such as S in this example. Figure 12 shows a graphical representation of the map S.

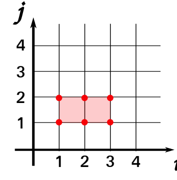

Figure 12. Graphical representation of a set

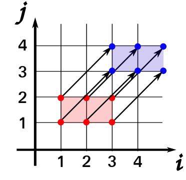

Figure 13. Graphical representation of a map

In general, an integer set has the form

$$S = \{N(\vec{s}) | f(\vec{s}, \vec{p})\}$$

with $\vec{s}$ representing the integer tuples of the integer set ($\vec{s} \in \mathbb{Z}^d$), N, a common name for all the tuples $\vec{s}$ usually used as the name of computations, d the dimensionality of the set, $\vec{p} \in \mathbb{Z}^e$ a vector of e parameters and $f(\vec{s}, \vec{p})$ a Presburger formula that evaluates to true, if and only if $\vec{s}$ is an element of S for the given parameters $\vec{p}$.

11.1 Relations (maps)

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

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

represents a relation between two sets. The first set is called the *domain* or the *source* and the second is called the *range* or the *sink*. Figure 13 shows a graphical representation of the map M.

In general, a map has the form

$$M = \{A(\vec{s}) \rightarrow B(\vec{o}), (\vec{s}, \vec{o}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} | f(\vec{s}, \vec{o}, \vec{p})\}$$

where $A(\vec{s})$ represents the domain or the source and $B(\vec{o})$ represents the range or the sink. d_1 and d_2 are the dimensionalities of $\vec{s}$ and $\vec{o}$, $\vec{p} \in \mathbb{Z}^e$ is a vector of e parameters and $f(\vec{s}, \vec{o}, \vec{p})$ is a Presburger formula that evaluates to true if and only if there is a relation from $\vec{s}$ to $\vec{o}$ in M for the given parameters $\vec{p}$.

12 Three-Layer IR

12.1 Layer I: Abstract Computation Layer

The first layer is a union of *computation sets* such that each computation set describes one statement in the program. Each computation set is defined as follows:

$$\{N1(\vec{s}) | f(\vec{s}, \vec{p})\} : g(N2(\vec{s}), N3(\vec{s}), ..., N4(\vec{s}))$$

where $N1(\vec{s})$ is a computation that has the name $N1$, and where $g(N2(\vec{s}), N3(\vec{s}), ..., N4(\vec{s}))$ is the expression that the computation computes and $f(\vec{s}, \vec{p})$ is a Presburger formula that evaluates to true, if and only if $\vec{s}$ is an element of S for the given parameters $\vec{p}$.

```
for (i in 0..N)
  for (j in 0..M)
    S1
    S2
```

(a) Original computation expressed as an imperative program

S1: $(i, j, 0)$ S2: $(i, j, 1)$ (b) Sequential	S1: $(j, i, 0)$ S2: $(j, i, 1)$ (c) Transposed
S1: $(i, 0, j)$ S2: $(i, 1, j)$ (d) Inner loop fission	S1: $(0, i, j)$ S2: $(1, i, j)$ (e) Outer loop fission
S1: $(i/N, i\%N, j, 0)$ S2: $(i/N, i\%N, j, 1)$ (f) Loop split	S1: $(i/N, j, i\%N, 0)$ S2: $(i/N, j, i\%N, 1)$ (g) ... & permuted
S1: $(i\%P\ (cpu), j, i/P, 0)$ S2: $(i\%P\ (cpu), j, i/P, 1)$ (h) Outer parallel	S1: $(i, j/4, 0, j\%4\ (vec))$ S2: $(i, j/4, 1, j\%4\ (vec))$ (i) Inner vectorized

Figure 14. For a simple loop next with two statements, examples of different time-processor vectors leading to many possible execution arrangements.

12.2 Layer II: Computation Placement Layer

The second layer is identical to the first layer except that computations in this layer are ordered based on their lexicographical order.

12.3 Layer III: Concrete Computation Layer

The third layer is a union of computation sets and a set of access relations. The computation sets are identical to the Layer II computation sets except that new allocation/deallocation statements are added. The set of access relations is described as follows:

$$\{N1(\vec{s}) \to B(\vec{o}), (\vec{s}, \vec{o}) \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} | f(\vec{s}, \vec{o}, \vec{p})\}$$

where $N1(\vec{s})$ is a computation mapped to the buffer element $B[\vec{o}]$ and $f(\vec{s}, \vec{o}, \vec{p})$ is a Presburger formula that evaluates to true if and only if there is a relation from $\vec{s}$ to $\vec{o}$ for the given parameters $\vec{p}$.

12.4 Time-Processor Vectors

The *time-space vector* in Layer II is a vector indicates the logical time of execution of computations and the processor on which they should be executed. Each one of those vectors has a name associated to it (the name of the computation). $S1(0, 0, 0)$, $S2(0, 0, 1)$, $S1(i, j, 0)$ and $S2(i(cpu), j, 1)$ are examples of time-space vectors representing computations in Layer II. In general, the time-space vector has two types of dimensions: time dimensions and space dimensions. The time dimensions provide the logical order of execution of the computations while the space dimensions indicate on which processor the computations should be executed. In the previous example, the first three vectors have time dimensions only, while the last vector has one space dimension. We use a tag to indicate that a given dimension is a space dimension; this tag indicates mainly the type of processor to which the computations are mapped.

Assuming that we have two time-space vectors we want to know which vector among the two executes first, then all we need to do is to compare the two vectors lexicographically [3]. In the example, $S1(0, 0, 0]$ precedes $S2(0, 0, 1)$ lexicographically, so $S1(0, 0, 0)$ is scheduled to be executed before $S2(0, 0, 1)$. The ability to add dimensions and reorder them freely enables the expression of multiple possible mappings from the original iteration space of the computations to complex execution scenarios. Figure 14 provides examples of different optimizations for a simple algorithm and shows the time-space vectors used to express those optimizations. Each of the dimensions of the vector can be an indexed variable, distributing the computation over that dimension or a constant providing a lexical ordering between statements. The algorithms will be using a custom intermediate representation within each DSL, however, we use a classical imperative language representation to describe them in this paper. A value can be annotated by a processor type, indicating where that computation will be placed, and indicating that dimension will be run in parallel.

[3] A *time-space vector* $(i_1, \ldots, i_k, \ldots, i_n)$ lexicographically precedes another time-space vector $(i'_1, \ldots, i'_k, \ldots, i'_n)$ if and only if $\exists k \in \mathbb{N}$ such that $i_1 = i'_1 \land i_2 = i'_2 \land \cdots \land i_k < i'_k$