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. 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.

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 [19]. 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 \( b_1 \) and \( b_2 \) 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 \( b_1 \) 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 \( b_2 \) and the loop over \( \text{out} \). 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 [23], 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 3-(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.
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 [35]. 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][3].

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 I.

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

A map is a relation between two integer sets. For example, \{ i, j \} → \{ S1(i, j) \} 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) 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 of 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 \( b(i, j, c) \) in Figure 1(a). The first part of that line, which is \( \{ b(i, j, c) \} \{ 0 = i, j \} \{ 0 = i, j \} \{ 0 = c, 3 \} \) specifies the iteration domain of the statement, while the second part, \( 1.5 * \text{img}(i, j, c) \), is the computed expression. The iteration domain is the set of tuples \( b(i, j, c) \) such that \( 0 ≤ i ≤ N ∧ 0 ≤ j ≤ M ∧ 0 ≤ c ≤ 3 \). Computations in Layer I are not ordered. The declaration order does not affect their order of execution, which is specified in Layer II.

---

1The constraints \( C_n \), \( C_m \), and \( C_k \) have been expanded inline.
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.

The set

\{b_1(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

\{b_2(0,i,cpu,j,c,1) \rightarrow buf_{g_{i,j}}[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 \(buf_{g_{i,j}}[i,j,c]\). 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.

\[\{A(i,j,k): 0<=i<N and 0<=j<N and 1<=k<N\}: \cos(i) + \cos(j) + \cos(k) + \cos(0)\]

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:

\[
\begin{align*}
&\{c0(i,j) : 0<=i<N and 0<=j<N\}: 0 \\
&\{c1(i,j,k) : 0<=i<N and 0<=j<N and 1<=k<N\}:
\end{align*}
\]

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 \(\phi\) node to merge them into one expression \(\phi(c0(i,j), c1(i,j,k-1)) + A(i,k) \ast B(k,j)\)

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.

Computations in Layer I are in Static Single Assignment (SSA) form [13]; each computation is defined only once (we use the \(\phi\) operator to deal with branches as in classical SSA).

For example the computation \(S0(0,0,0)\) is lexically before the computation \(S0(0,0,1)\) and the computations \(S0(i,0,0)\) are lexically before the computations \(S0(i,0,1)\).
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 x 16 tiles and parallelize the outermost loop:

\[
\{C(i,j): 0\leq i < N \text{ and } 0\leq j < N\}: A(i,j) + B(i,j)
\]

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

\[
\{C(i,j)\rightarrow C(i1(cpu),j1,i2,j2): i1=\text{floor}(i/16) \text{ and } i2=i\%16 \text{ and } j1=\text{floor}(j/16) \text{ and } j2=j\%16 \text{ and } 0\leq i < N \text{ and } 0\leq j < N\}
\]

which will produce the following set in Layer II:

\[
\{C(i1(cpu),j1,i2,j2): i1=\text{floor}(i/16) \text{ and } i2=i\%16 \text{ and } j1=\text{floor}(j/16) \text{ and } j2=j\%16 \text{ and } 0\leq i < N \text{ and } 0\leq j < N\}: A(16i+12, 16j+12) \times B(16i+12, 16j+12)
\]

#### 4.5.2 High Level Scheduling Commands

Tiramisu provides a set of predefined scheduling maps for common affine loop nest transformations. Table I, 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 [71]. 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 \texttt{@acc} macro are optimized with \texttt{Tiramisu}.

We generate \texttt{Tiramisu} from the low-level Julia IR (which is in SSA form) by translating each statement in the Julia IR into a computation in \texttt{Tiramisu}. This Julia low-level IR does not have high level control flow (it only has \texttt{gosub}); 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 \texttt{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 \texttt{Tiramisu} code generator (which uses the \texttt{SSL} 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 \texttt{cpu} dimensions are translated into parallel loops in the generated code, using OpenMP-style parallelism. Loops that are tagged with the \texttt{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 \texttt{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.

6.3 Distributed Memory Systems

\texttt{Tiramisu} utilizes MPI to generate code for distributed memory systems. Figure 3 shows \texttt{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 \texttt{chunk_z} rows. On line 7 the outer loop is split by \texttt{chunk_z}. 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. \texttt{blk} represents a blocking call, and \texttt{ablk} represents an asynchronous, blocking call. We use two-sided communication in \texttt{Tiramisu}, meaning communication is done with pairs of \texttt{send} and \texttt{receive} operations. The actual transfer is defined by \texttt{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 \texttt{y1} of \texttt{bx} and \texttt{by} as distributed, and line 12 tags dimension \texttt{q} of the \texttt{send} and \texttt{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:

\begin{verbatim}
for(q in 1..N-1) (...) // distribute on q
\end{verbatim}

becomes:

\begin{verbatim}
q = get_rank(); if (q>1 and q<N-1) (...)
\end{verbatim}

All of the other scheduling commands in \texttt{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

---

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

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; rgb420, 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 Tiramisu, a compiler designed to support recurrent filters.

heat3d(x,y,z,t) = a*heat3d(x, y, z, t-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 and on checking the legality of transformations using the polyhedral...
We used the following benchmarks to evaluate the integration of Tiramisu within Julia: \texttt{bicg}, a biconjugate gradient method; \texttt{doitgen}, a multiresolution analysis kernel; \texttt{mttkrp}, the matricized tensor times Khatri-Rao product; \texttt{covariance}, which performs a covariance computation; and \texttt{gesummv}, which is summed matrix-vector multiplications. For a fair comparison, the Julia code was tagged with the \texttt{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 \texttt{covariance}, \texttt{doitgen}, \texttt{mttkrp} and \texttt{bicg} are mainly due to the improved data locality obtained after tiling using Tiramisu, which is not possible in Julia.

Figure 7 shows the execution time of Julia-Tiramisu/FROST and Vivado HLS Video Library (ms)

### 7.3 Evaluating Backends

#### 7.3.1 FPGA Backend

We evaluate the FPGA backend in Tiramisu using 6 image processing kernels: convolution, \texttt{cvtColor}, \texttt{gaussian}, \texttt{scale}, \texttt{sobel}, and \texttt{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 run on an 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 \texttt{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, \texttt{gaussian} and \texttt{sobel}).
7.4 Generating BLAS \texttt{sgemm} using \textsc{Tiramisu}

To evaluate the performance of \textsc{Tiramisu} on an extreme case, we used \textsc{Tiramisu} to generate the BLAS generalized matrix multiplication (\texttt{sgemm}) which computes $C = \alpha AB + \beta C$. The \texttt{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 \textsc{Tiramisu} implementation of \texttt{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 \textsc{Tiramisu} distributed backend, we used 6 kernels for evaluation: \texttt{blurxy}, \texttt{sobel}, \texttt{convolution}, \texttt{gaussian}, \texttt{pipeline}, and \texttt{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, \texttt{pipeline}, and \texttt{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 \textsc{Tiramisu} and distributed Halide on 16 nodes for each of the kernels. \textsc{Tiramisu} is faster than distributed Halide in each case. For the kernels involving communication, code generated by distributed Halide has two problems compared to \textsc{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 \textsc{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 \texttt{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 \texttt{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.
that generic DSL frameworks like Delite can benefit from using Tiramisu.

PENCIL [34] 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. Darkroom 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 [19,37,49]: 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.
References


10 Notation and Definitions

10.1 Presburger formula

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

\[
\begin{align*}
<\text{formula}> & \leftarrow <\text{formula}> \land <\text{formula}> \\
& \leftarrow <\text{formula}> \lor <\text{formula}> \\
& \leftarrow <\text{formula}> \equiv <\text{formula}> \\
& \leftarrow \exists \text{-var}. <\text{formula}> \\
& \leftarrow <\text{var}> <\text{atom}> \\
& \leftarrow <\text{numeral}> <\text{term}> \\
& \leftarrow <\text{term}> \neg <\text{term}> \\
& \leftarrow <\text{term}> \ast <\text{numeral}> <\text{term}> \\
& \leftarrow |<\text{term}>| <\text{term}> \\
& \leftarrow |<\text{term}>| |<\text{term}>|
\end{align*}
\]

Note that \(<\text{numeral}> \ast <\text{term}>\) is not a general multiplication operator; it is a shortcut for \(<\text{term}> \ast \ast \ast <\text{term}>\).

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 \(+, -, \times, /, \mod, \&\&, |, <, <, >, >, =, ==, !=, \&\&\&\&\&\&\&\&\&,\) 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 \(x\) 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 to 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:

\[\{S(i, j) : 1 \leq i \leq 3 \land 1 \leq j \leq 2\}\]

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 \(S\).
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 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:

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

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

12.4 Time-Processor Vectors

The *time-space vector* in Layer II is a vector indicating 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. 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.

![Figure 14](image)

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

(a) Original computation expressed as an imperative program

<table>
<thead>
<tr>
<th>Layer I: Iteration Space</th>
<th>Layer II: Computation Placement Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S1: (i, j, 0)$</td>
<td>$S1: (i, j, 0)$</td>
</tr>
<tr>
<td>$S2: (i, j, 1)$</td>
<td>$S2: (i, j, 1)$</td>
</tr>
<tr>
<td>(b) Sequential</td>
<td>(c) Transposed</td>
</tr>
<tr>
<td>$S1: (i, 0, j)$</td>
<td>$S1: (0, i, 0)$</td>
</tr>
<tr>
<td>$S2: (i, 1, j)$</td>
<td>$S2: (1, i, j)$</td>
</tr>
<tr>
<td>(d) Inner loop fission</td>
<td>(e) Outer loop fission</td>
</tr>
<tr>
<td>$S1: (i/N, %N, j, 0)$</td>
<td>$S1: (i/N, j, %N, 0)$</td>
</tr>
<tr>
<td>$S2: (i/N, %N, j, 1)$</td>
<td>$S2: (i/N, j, %N, 1)$</td>
</tr>
<tr>
<td>(f) Loop split</td>
<td>(g) ... &amp; permuted</td>
</tr>
<tr>
<td>$S1: (%P(cpu), j, i/P, 0)$</td>
<td>$S1: (i, j/4, 0, j/4)$ (vec)</td>
</tr>
<tr>
<td>$S2: (%P(cpu), j, i/P, 1)$</td>
<td>$S2: (i, j/4, 1, j/4)$ (vec)</td>
</tr>
<tr>
<td>(h) Outer parallel</td>
<td>(i) Inner vectorized</td>
</tr>
</tbody>
</table>


1 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 $i_1 = i'_1 \land i_2 = i'_2 \land \cdots \land i_k < i'_k$