Tiramisu: A Code Optimization Framework for High Performance Systems

Riyadh Baghdadi
MIT
baghdadi@mit.edu

Jessica Ray
MIT
jray@mit.edu

Malek Ben Romdhane
MIT
malek@mit.edu

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

Patricia Suriana
Google
psuriana@google.com

Shoaib Kamil
Adobe
kamil@adobe.com

Saman Amarasinghe
MIT
saman@mit.edu

ABSTRACT

This paper introduces Tiramisu, an optimization framework designed to generate efficient code for high-performance systems such as multicores, GPUs, FPGAs, distributed machines, or any combination of these. Tiramisu relies on a flexible representation based on the polyhedral model and introduces a novel four-level IR that allows full separation between algorithms, schedules, data-layouts and communication. This separation simplifies targeting multiple hardware architectures from the same algorithm. We evaluate Tiramisu by writing a set of linear algebra and DNN kernels and by integrating it as a pass in the Halide compiler. We show that Tiramisu extends Halide with many new capabilities, and that Tiramisu can generate efficient code for multicores, GPUs, FPGAs and distributed heterogeneous systems. The performance of code generated by the Tiramisu backends matches or exceeds hand-optimized reference implementations. For example, the multicore backend matches the highly optimized Intel MKL library on many kernels and shows speedups reaching 4× over the original Halide.

KEYWORDS

Code Optimization, Generation, Polyhedral Model, CPU, GPU, FPGA, Distributed System, Heterogeneous System

1 INTRODUCTION

High performance systems today range from multicore mobile phones to large scale heterogeneous supercomputers and cloud infrastructures equipped with GPUs and FPGAs. Building a code optimization framework for these high performance systems requires solving many challenges. In this paper we present an optimization framework that addresses four of these challenges.

The first is the *multi-language* or the *MPI+OpenMP +CUDA+HLS* challenge. Most high performance computer systems are complex and increasingly heterogeneous; they may be single-node or distributed and may have GPUs [32] and FPGAs [10]. Achieving best performance requires taking full advantage of all these different architectures [53]. Writing code for such heterogeneous systems is difficult as each hardware architecture requires drastically different styles of code and optimization, all using different libraries and languages. In addition, partitioning the program between heterogeneous components that correctly communicate and synchronize is difficult. The current practice is to manually write and optimize the program in separate languages and libraries for each component. However, even a small change to the partitioning among heterogeneous units will often require a complete rewrite of the program.

The second challenge is that of *memory dependence*. Most intermediate representations use memory to communicate between program statements. This creates memory-based dependences in the program and also means that the data-layout is chosen before deciding how the code should be scheduled (i.e., how it should be optimized and mapped to hardware). Optimizing a program for different hardware architectures usually requires modifying the data-layout and eliminating memory-based dependences since they restrict optimization [33]. Thus, any data-layout specified before scheduling must be undone to allow more freedom for scheduling, and the code must be adapted to use the data-layout best-suited for the target hardware. Applying these data-layout transformations and the elimination of memory-based dependences is challenging [16, 19, 23, 30, 31, 34, 35, 40, 48].

PACT'18, November 2018, Limassol, Cyprus
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The third challenge is the ability to *optimize and generate efficient code*. In many performance critical domains, users need code that achieves performance comparable to hand-optimized code. Generating such code requires combinations of non-trivial program transformations that optimization frameworks try to fully automate using cost models, heuristics [25], and machine learning [47]. While these automatic optimization frameworks provide productivity, they may not always achieve the desired level of performance. Some frameworks also impose restrictions on the type of programs they support, since they cannot decide the correctness of schedules otherwise. While these language restrictions guarantee correctness, they may prevent users from applying valid combinations of optimizations. A more flexible way to guarantee the correctness of optimizations is needed.

The fourth challenge is that of *representation*. Lowering code to execute on complex hardware architectures requires numerous transformations that change program structure by introducing new loops, complex loop bounds, and non-trivial array accesses [52]. Analyzing code generated by one of these transformations is challenging, which complicates composition with other transformations. This problem can be mitigated by keeping loops within a single unified representation through all transformations. However, many representations are inadequate or too conservative to support complex transformations or for tasks such as dependence analysis (necessary for deciding about the correctness of optimization) and tasks such as the computation of communication sets (data to send/receive in a distributed system).

This paper addresses these challenges by introducing Tiramisu, a compiler optimization framework designed for targeting high performance systems. Tiramisu takes a high level representation of the program (pure algorithm and a set of commands specifying the schedule and data-layout), applies transformations on the representation and generates highly optimized code for the target architectures. Tiramisu is well suited for the implementation of data parallel algorithms (loop nests manipulating arrays). It is designed to hide the complexity and large variety of execution platforms by providing a multi-layer representation suitable for transforming from high-level languages to multicore CPUs, GPUs, distributed machines, and FPGAs.

Tiramisu addresses the first challenge by allowing users to partition their program and specify communication from the same source code using a simple set of scheduling commands. This simplifies programming distributed and heterogeneous systems: the algorithm does not change and only commands that control its execution and communication mapping require modification. Tiramisu also addresses the first challenge by using a novel multi-layer IR that fully separates the architecture-independent algorithm from the schedule, data-layout and communication. The multi-layer design makes the algorithm portable and makes it easier to perform each program transformation at the right layer of abstraction. This multi-layer IR also helps Tiramisu address the memory dependence challenge since this design separates data-layout from other transformations.

Tiramisu addresses the challenge of optimization by separating mechanism from policy in scheduling and by removing heuristics and automatic decision-making. This way, Tiramisu allows full control over scheduling while still enabling integration with higher level frameworks for policy-making (deciding which optimization should be applied). Tiramisu guarantees correctness using dependence analysis and thus does not need to impose undue restrictions on its input language to guarantee correctness.

The challenge of representation is addressed by using a unified framework based on polyhedral sets to represent the four 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 polyhedral framework also enables the application of a large set of complex optimizations. Tiramisu does not extend the core of the polyhedral model, but rather it leverages the power of polyhedral compilation to target heterogeneous systems and to generate efficient code that matches kernels from highly optimized libraries such as the Intel MKL library. To the best of our knowledge, Tiramisu is the first framework that uses polyhedral techniques and matches the performance of a single *sgemm* kernel from Intel MKL.

In this paper we make the following contributions:

- We introduce a unified framework that generates code for multiple high-performance architectures including multicore CPUs, GPUs, FPGAs, distributed machines, or any combination of these, using a set of simple scheduling commands to guide program transformations.
- We introduce a novel four-layer representation that separates the algorithm from code transformations and data-layout transformations, allowing for portability and simplifying the composition of architecture-specific lowering transformations.
- We demonstrate the first polyhedral framework that can generate code that matches a single *sgemm* kernel from the highly optimized Intel MKL library.
- We demonstrate the power and viability of Tiramisu by using it to write multiple linear algebra and neural network kernels and by using it as an optimization framework for the Halide [41, 42] image processing domain-specific language.
- We demonstrate the expressiveness of Tiramisu by extending Halide with new capabilities such as expressing

code with cyclic dataflow, performing precise bounds inference for non-rectangular iteration spaces, and performing advanced loop transformations such as skewing.
- We evaluate TIRAMISU and show that it matches or outperforms reference implementations on different hardware architecture backends (multicore CPUs, GPUs, FPGAs and distributed machines).

2 TIRAMISU OVERVIEW

TIRAMISU is an optimization framework that takes as input a high level, architecture-independent representation of code and a set of scheduling and data mapping commands that guide code transformation. The input can either be generated by a domain-specific language (DSL) compiler or directly written by a programmer. TIRAMISU then applies the user-specified code and data-layout transformations and generates an architecture-specific, low-level IR that takes advantage of modern architectural features such as multicore parallelism, non-uniform memory (NUMA) hierarchies, clusters, and accelerators like GPUs and FPGAs.

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

Approach. Obtaining high-performance requires a holistic approach. Getting the best performance involves choosing the right algorithm, mapping the algorithm to hardware, and choosing the best distribution of data and communication. It is necessary to think about all of these issues at a high level and to make global decisions. However, the vast majority of the time of a programmer is spent implementing low-level details of these global decisions. TIRAMISU simplifies the process of implementing these details by breaking this process into four layers; instead of worrying about all the details at the same time, the programmer can focus on one type of detail at each layer. TIRAMISU's novel layer abstraction not only simplifies the programmer's task but also simplifies compiler design. The separation of layers ensures that compiler passes in a given layer should not worry about modifying or undoing a decision made in an earlier layer. For example, the phase that specifies the order of computations and where they occur can safely assume that no data-layout transformations are required, greatly simplifying the phase. This simple assumption allows TIRAMISU to avoid the need to rely on a large body of research that focuses on data-layout transformations to allow scheduling [16, 19, 23, 30, 31, 34, 35, 40, 48].

2.1 The Four-Layer IR

TIRAMISU uses *polyhedral sets* to represent each of the four 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 [51]. We use a combination of classical extensions to the polyhedral model in order to support non-affine iteration spaces; these extensions are sufficient for large classes of programs in practice [5, 6], and in particular to our areas of interest: dense linear algebra and tensor algebra, stencils, image processing and deep neural networks.

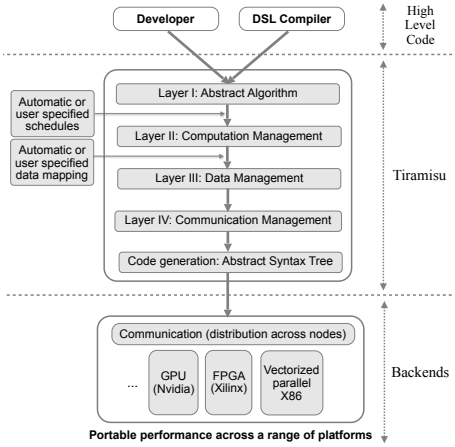

Figure 1: TIRAMISU overview

A typical workflow of using TIRAMISU is illustrated in Figure 1. The first layer of TIRAMISU can be written directly by a developer or generated from a DSL compiler. The first layer of the IR is then transformed to lower layers, and finally TIRAMISU generates LLVM or other appropriate low-level IR.

The four layers of the TIRAMISU IR are:

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

Layer II (Computation Management) specifies the order of execution of computations and the processor on which they execute. This layer does not specify how intermediate values are stored in memory; this simplifies optimization passes since these transformations do not need to perform complicated data-layout transformations. The transformation of Layer I into Layer II is done automatically using scheduling commands. Examples of scheduling commands supported in TIRAMISU are shown in Table 1.

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

Layer IV (Communication Management) adds synchronization and communication operations to the representation, as well as scheduling when statements for buffer allocation/deallocation occur. This layer is generated automatically from Layer III by applying user-specified commands.

3 RELATED WORK

The design of Tiramisu inherits from the design of two systems: Halide [41] and PENCIL [4]. It takes the best aspects of the two systems to build an optimization framework targeting high performance systems including distributed heterogeneous systems.

Halide [41] is an image processing DSL that relies on conservative rules to determine whether a schedule is legal; for example, Halide does not allow fusion of two loops (using the `compute_with` command) if the second loop reads a value produced by the first loop. While this rule avoids illegal fusion, it prevents fusing many legal common cases which may lead to suboptimal performance. Halide also assumes the program has an acyclic dataflow graph in order to simplify checking the legality of scheduling commands. This prevents users from expressing many programs with cyclic dataflow. It is possible in some cases to work around the above restrictions, but these methods are not general. Tiramisu avoids over-conservative constraints by relying on dependence analysis to check for the correctness of code transformations, enabling more possible schedules.

Since Halide uses intervals to represent iteration spaces, it cannot naturally represent non-rectangular iteration spaces. This makes certain Halide passes over-approximate non-rectangular iteration spaces which leads to less efficient code generation. Tiramisu in contrast can express non-rectangular iteration spaces naturally since it uses a polyhedral representation. Relying on a polyhedral representation also enables Tiramisu to perform precise bounds inference for non-rectangular iteration spaces as well as performing many complex affine transformations such as iteration space skewing which Halide cannot perform.

PENCIL [4, 5] is a generic DSL intermediate representation (IR) based on the polyhedral representation. PENCIL uses the Pluto [9] algorithm for automatic scheduling. The Tensor Comprehensions [50] compiler uses the PENCIL compiler as a backend thus it has characteristics that are similar to PENCIL. PolyMage [36] is a polyhedral compiler similar to Halide that uses fully automatic scheduling heuristics. While such fully automatic approaches provide productivity, they may not always provide the best performance. Thus, instead of fully automatic scheduling, Tiramisu uses a set of scheduling commands, giving the user full control over scheduling. CHiLL [13, 24], AlphaZ [54] and URUK [21] are polyhedral frameworks that allow users to express high-level transformations using scheduling commands, freeing users from having to implement them. The main difference between Tiramisu and these frameworks is that Tiramisu is designed to target distributed heterogeneous systems in addition to single-node architectures.

Polyhedral frameworks proposed by Amarasinghe and Lam [1] and Bondhugula [8] address the problem of fully automatic code generation for distributed systems. Tiramisu makes a different design choice, relying on the user to provide scheduling commands to control choices in the generated code (synchronous/asynchronous communication, the granularity of communication, buffer sizes, when to send and when to receive, explore communication versus recomputation, etc.). Even though Tiramisu provides a mechanism for code optimization (i.e., it provides scheduling commands for controlling how the program should be optimized), it is still possible to build a framework that provides policy on top of Tiramisu (i.e., a framework that automates scheduling). The separation between mechanism and policy allows users to choose between using automatic scheduling or manual scheduling which provides more flexibility.

In general, the goal of Tiramisu is not to extend the core of the polyhedral model itself, but rather to leverage the power of the polyhedral model to build a framework that targets high performance systems (multicores, GPUs, FPGA, distributed and any combination of these). Its goal is to also demonstrate that a polyhedral framework can generate efficient code that matches the performance of highly optimized libraries such as Intel MKL. To the best of our knowledge, Tiramisu is the first to demonstrate that a polyhedral compiler can generate code that matches a single *sgemm* kernel from the Intel MKL library.

The Cyclops Tensor Framework (CTF) [45] is a library for performing tensor contractions, primarily in the field of quantum chemistry. CTF automatically decomposes tensors using a communication-optimal tensor contraction algorithm and maps the computations to the underlying architecture. The framework targets distributed architectures, and can provide hybrid execution through the use of MPI and OpenMP. Unlike Tiramisu which is designed to be more general, CTF is designed mainly for tensor contractions. Chapel [12] is a parallel programming language that supports a Partitioned Global Address Space (PGAS) memory model [29]. In this model, code can refer to variables and arrays regardless of whether they are stored in a local or remote memory. Any necessary communication is automatically inserted by the compiler and executed at runtime. Similarly, computations in Layer I of Tiramisu refer to other computations in the same way regardless of whether they are stored or computed in local or remote memory (or in shared or global memory on GPU). Specifying whether a computation is local or remote, how it is accessed and whether communication is needed are all done using scheduling commands at Layers II, III and

Commands to transform Layer I into Layer II, III and IV 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_ts_map()`	Set the time-space map for C (to transform Layer I to 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	
`Buffer b(...)`	Declare a buffer b (size, type, ...)
`C(`i_0`,..) .store_in(b[`i_0`,..])`	Store the result of the computation $C(i0; ::)$ in b[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_send(...)`	Create a send communication statement
`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

IV. TIRAMISU provides a set of fine-grain commands that can implement any data-mapping and communication that a language like Chapel provides, yet TIRAMISU can complement Chapel by providing new capabilities such as advanced loop nest transformations and checking schedule validity.

Delite [11] is a generic framework for building DSL compilers using Lightweight Modular Staging (LMS) [44]. It exposes several parallel computation patterns that DSLs can use to express parallelism. NOVA [14] and Lift [46] are other IRs for DSL compilers. They are functional languages that rely on a suite of higher-order functions such as map, reduce, and scan to express parallelism. Weld [38] is another IR designed for the area of databases and data analytics and can be used to implement libraries, such as Numpy, and enable optimizations across different library calls. TIRAMISU is complementary to these frameworks as TIRAMISU would allow them to perform and compose a large set of complex affine transformations.

Most functional languages do not expose notions of memory layout to programmers. Compared to those languages, TIRAMISU enables writing algorithms in a functional manner while separately dealing with data-layout and computation scheduling using a fine-grained scheduling language.

4 THE TIRAMISU IR

In order to generate code, a TIRAMISU user provides Layer I computations and a set of scheduling commands. Layer II is generated automatically by applying the schedule to Layer I. The user then specifies commands for buffer allocation and data-layout mapping. These commands augment the Layer II representation; the result constitutes Layer III. The newly added buffer allocation statements are not yet scheduled. Finally, the user provides commands specifying how communication is performed, and also provides commands to schedule any communication and buffer allocation statements. These commands are applied on the Layer III representation, resulting in the Layer IV representation. An annotated abstract syntax tree (AST) is then generated from Layer IV; this AST is traversed to generate the target code.

4.1 An Example in the Four-Layer IR

We first provide an overview of polyhedral sets and maps. More details and formal definitions for these concepts are provided in [3, 39, 51].

An *integer set* is a set of integer tuples described using affine constraints. An example of a set of integer tuples is $\{(1; 1); (2; 1); (3; 1); (1; 2); (2; 2); (3; 2)\}$. Instead of listing all tuples in a set, we describe the set using affine constraints over loop iterators and symbolic constants: $\{S(i; j) : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2\}$ where i and j are the dimensions of tuples in the set.

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

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

is a map between tuples in the set S1 and tuples in the set S2 (e.g. the tuple $S1(i; j)$ maps to the tuple $S2(i + 2; j + 2)$). We use the Integer Set Library (ISL) [51] notation for sets and maps.

Figure 2 shows a code snippet and two optimized versions of that code: a version for a multicore machine and one for a distributed GPU system. The original, unoptimized code is shown in the left side of Figure 2-(a), while the right side shows the Layer I representation of this code. The Layer I representation remains 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 2-(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 statement in line 5 in Figure 2-(a). The first part of that line[1], which is $\{b_1(i; j; c) : 0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq c < 3\}$ specifies the iteration domain of the statement, while the second part, $1:5 * imd(i; j; c)$, is the computed expression. The iteration domain is the set of tuples $b_1(i; j; c)$ such that $0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq c < 3$. Computations in Layer I are not ordered; declaration order does not affect the order of execution, which is specified in Layer II.

Figure 2-(b) shows the first optimized version of the code, produced by the set of scheduling and data-layout commands on the right side. Examples of scheduling commands are shown in Table 1. The generated Layer II representation is

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

Constraints: $C_n : 0 \leq i < N$, $C_m : 0 \leq j < M$, $C_{m'} : 1 \leq j < M-1$, $C_k : 0 \leq c < 3$, $C_q : 0 \leq q < NODES$, $Sp : q = i\%(N/NODES) \wedge i' = i/(N/NODES)$

Different Code Optimizations	TIRAMISU representation (Layer I, Layer II, Layer III and Layer IV)	
(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] 6 for (j in 1..M-1) 7 for (c in 0..3) 8 out[i][j][c] = (b1[j-1][c] + b1[j+1][c])/2	**Layer I**	// The constraints C_n, C_m and C_k are defined above. $\{b_1(i;j;c) : C_n \wedge C_m \wedge C_k\}$: 1:5 $* imd(i;j;c)$ $\{out(i;j;c) : C_n \wedge C_{m'} \wedge C_k\}$: $(b_1(i;j-1;c) + b_1(i;j+1;c))$=2

(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 b1[i][j][c] = 1.5*img[i][j][c]
6 for (j in 1..M-1)
7 for (c in 0..3)
8 out[i][j][c] = (b1[i][j-1][c]+b1[i][j+1][c])/2
	Layer II	// Scheduling commands to generate Layer II:
		b_1.parallel(i); out.parallel(i); out.after(b_1, i);
		
		// Layer II generated from Layer I
		$\{b_1(i(cpu);0;j;c) : C_n \wedge C_m \wedge C_k\}$: 1:5 $* imd(i;j;c)$
		$\{out(i(cpu);1;j;c) : C_n \wedge C_{m'} \wedge C_k\}$:$(b_1(i;0;j-1;c) + b_1(i;0;j+1;c))$=2

	Layer III	// Scheduling commands to generate Layer III:
		$b_1(i;j;c)$.set_access($buf_{b1}[i;j;c]$); $out(i;j;c)$.set_access($buf_{out}[i;j;c]$);
		
		// Generated Layer III: Same as Layer II + the following data mapping
		$\{b_1(i(cpu);0;j;c) \rightarrow buf_{b1}[i;j;c] : C_n \wedge C_m \wedge C_k\}$
		$\{out(i(cpu);1;j;c) \rightarrow buf_{out}[i;j;c] : C_n \wedge C_{m'} \wedge C_k\}$

	Layer IV	Same a Layer III (since no communication is needed)

(c)
1 // Code optimized for multi-GPU
2 // Communication to GPU and buffer allocation ommitted
3 p = current_node()
4 if (p = 0)
5 for (q in 1..NODES)
6 send a chunk of img to processor q
7 if (p != 0)
8 receive a chunk of img from master
9 distributed for (q in 0..NODES)
10 gpu for (i in 0..N/NODES)
11 gpu for (j in 0..M)
12 for (c in 0..3)
13 b1[i][c][j] = 1.5*img[i][j][c]
14 distributed for (q in 0..NODES)
15 gpu for (i in 0..N/NODES)
16 gpu for (j in 1..M-1)
17 for (c in 0..3)
18 out[i][j][c] = (b1[i][c][j-1]+b1[i][c][j+1])/2
	Layer II	// Scheduling commands to generate Layer II:
		b_1.split(i, N/NODES, q, i); out.split(i, N/NODES, q, i);
		b_1.distribute(q); out.distribute(q); b_1.gpu(i,j); out.gpu(i,j); out.after(b_1, root);
		
		// Layer II generated from Layer I
		$\{b_1(2;q(node);i'(\partial pu);j(\partial pu);c) : Sp \wedge C_n \wedge C_m \wedge C_k\}$: 1:5 $* imd(i';j;c)$
		$\{out(3;q(node);i'(\partial pu);j(\partial pu);c) : Sp \wedge C_n \wedge C_{m'} \wedge C_k\}$:
		$(b_1(2;q;i';j-1;c) + b_1(2;q;i';j+1;c))$=2

	Layer III	// Scheduling commands to generate Layer III:
		$b_1(i;j;c)$.set_access($buf_{b1}[i;c;j]$); $out(i;j;c)$.set_access($buf_{out}[i;j;c]$);
		
		// Generated Layer III: Layer II + the following data mapping
		$\{b_1(2;q(node);i'(\partial pu);j(\partial pu);c) \rightarrow buf_{b1}[i';c;j] : Sp \wedge C_n \wedge C_m \wedge C_k\}$
		$\{out(3;q(node);i'(\partial pu);j(\partial pu);c) \rightarrow buf_{out}[i';j;c] : Sp \wedge C_n \wedge C'_m \wedge C_k\}$

	Layer IV	// Scheduling commands to generate Layer VI (create sends and receives):
		s = create_send(img, q, $\{send[p;q] : p = 0 \wedge 1 \leq q < NODES\}$, ...);
		r = create_receive(img, 0, $\{receive[p] : 1 < p < NODES\}$, ...);
		s.distribute(p); r.distribute(p); r.after(s, root); b_1.after(r, root);
		
		Generated Layer IV: Same a Layer III + the following communication statements
		$\{send(0;p(node);q) : p = 0 \wedge 1 \leq q < NODES\}$:$send(:::)$
		$\{receive(1;p(node)) : 1 < p < NODES\}$:$receive(:::)$

Figure 2: Three versions of the motivating example (left) and their equivalent Layer I, II, III and IV (right)

shown in Figure 2-(b), right side. Computations in Layer II are ordered based on their lexicographical order[2]. The set

$$\{b_1(i(cpu);0;j;c) : 0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq c < 3\}$$

in the example, is an ordered set of computations. The tag *(cpu)* for the i dimension indicates that each i-th iteration is mapped to the i-th CPU. In Layer II, the total ordering of these tuples determines the execution order.

Layer III in Figure 2-(b) adds data-layout mapping to Layer II, concretizing where each computation is stored (declarations of memory buffers and scalars are also introduced in this layer, based on the schedule). In the example, the data mapping

$$\{b_1(i(cpu);0;j;c) \rightarrow buf_{b1}[i;j;c] : 0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq c < 3\}$$

indicates that the result of the computation $b_1(i(cpu);0;j;c)$ is stored in the array element $buf_{b1}[i;j;c]$. 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 any data-layout mapping that can be expressed as an affine relation.

Figure 2-(c) shows a second version of the code optimized for a distributed system with GPUs. The outermost loop is first split and then the resulting outermost loop is mapped to different nodes. The two inner loops execute on the GPU. The data-layout of the computation b1 is modified to improve GPU memory access coalescing.

For brevity, the declaration of buffers, their types, their allocations (including when and where they are allocated), as well as host-to-gpu communication are all omitted from

[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 $S0(1;i;0)$

the examples, but such information must be specified by the user for correct code generation.

4.2 Layer I: Abstract Algorithm

The first layer defines abstract computations, which are not yet scheduled or mapped to memory. Each computation represents an expression to compute.

As an example, the following code

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

can be represented as

$$\{A(i;j) : 0 \leq i < 4 \wedge 0 \leq j < 4 \wedge i < j \wedge i \neq 2\} : cos(i)$$

though it is important to remember that this representation, unlike the pseudocode above, does not 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.

Computations in Layer I are in Static Single Assignment (SSA) form [15]; each computation is defined only once. Like classical SSA, we use the ϕ operator for branches; for example if a computation is defined in the two branches of a conditional and then this computation is read afterwards, we use a ϕ node to aggregate the two definitions.

Support for Non-Affine Iteration Spaces. TIRAMISU represents non-affine array accesses, non-affine loop bounds, and non-affine conditionals in a way similar to Benabderrahmane et al. [7]. For example, a conditional is transformed into a predicate and is attached to the computation. The list of accesses of the computation is the union of the accesses of the computation in the two branches of the conditional; this is an over-approximation. During code generation, a preprocessing step inserts the conditional back into the generated code. The efficiency of these techniques was confirmed in the PENCIL compiler [5]. Our experiences in general, as well as the experiments in this paper, show that these approximations do not hamper performance.

4.3 Layer II: Computation Management

The computation management layer describes when and where each computation is computed. Unlike the first layer, computations in this layer are ordered and assigned to a particular processor (i.e., we know when and where they will run). This order is dictated by *time dimensions* and *space dimensions*. Time dimensions specify the order of execution relative to other computations while space dimensions specify on which processor each computation executes. The ordering of the time dimensions determines the execution order of each computation. Space dimensions only indicate where computations run. 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 the dimension X of gpu threads (0 is the outermost dimension).
`gpu_block_X`	the dimension runs on the dimension X of gpu blocks.

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

Other tags that TIRAMISU supports and that can be used to describe how a dimension should be optimized include:

`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.

4.4 Layer III: Data Management

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

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

4.5 Layer IV: Communication Management

In this layer, communication statements (including synchronization) are added and scheduled (i.e., mapped to the time-space domain). Any allocation or deallocation operation that was added in Layer III is also scheduled in this layer.

5 LOWERING TIRAMISU LAYERS

Generating code using TIRAMISU requires sequentially lowering from each upper layer to the layer below. In this section we describe how TIRAMISU generates Layer II, III and IV. Lowering is done through two means: *time-space mapping* and *adding new statements*. *Time-space mapping* maps computations to the time-space domain. This mapping is done by applying an affine relation, called a *time-space map*. *Adding new statements* happens between Layer II, III and IV. These statements are added by the TIRAMISU user by issuing commands. Examples of statements added include

buffer allocation/deallocation needed during the lowering of Layer II to Layer III and communication statements needed while lowering Layer III into Layer IV. Since adding new statements by the user is trivial, the rest of this section focuses on time-space mapping. We present time-space maps and the high level scheduling commands (which specify both data-mapping and time-space mapping and also can be used to add buffer allocation/deallocation and communication). We also describe how Tiramisu ensures the correctness of a given schedule.

Time-space 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. We call this map a *time-space map*. A time-space map transforms the iteration domain from Layer I into a new set that represents the computation in the time-space domain. For example, suppose we want to tile the following computation in Layer I into 16×16 tiles:

$$\{C(i;j) : 0 \leq i < N \wedge 0 \leq j < N\} : A(i;j) + B(i;j)$$

To do so, we provide the following time-space map to Tiramisu:

$$\{C(i;j) \rightarrow C(i1;j1;i2;j2) : i1 = floor(i/16) \wedge i2 = i\%16 \wedge j1 = floor(j/16) \wedge j2 = j\%16 \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

which produces the following set in Layer II:

$$\{C(i1;j1;i2;j2) : i1 = floor(i/16) \wedge i2 = i\%16 \wedge j1 = floor(j/16) \wedge j2 = j\%16 \wedge 0 \leq i < N \wedge 0 \leq j < N\} : A(i1*16+i2;j1*16+j2) + B(i1*16+i2;j1*16+j2)$$

High Level Scheduling Commands. Tiramisu provides a set of high-level time-space maps for common affine loop nest transformations. Table 1 shows examples of these commands. Each command generates a time-space map that is applied to the Layer I representation during lowering. Composing many transformations can be done simply by composing different time-space maps, since the composition of two affine maps is an affine map.

Checking the Validity of Schedules. To check the validity of transformations, we first compute the dependences of the input program using array data-flow analysis [20]. The original dependences (order between producers and consumers) represent the semantics of the program. After computing dependences, we check the validity of transformations using violated dependence analysis [49]: a schedule is valid if it preserves the original semantics of the program (i.e., if it preserves the order between producers and consumers).

6 CODE GENERATION

Generating code from the set of computations in Layer IV amounts to generating nested loops that visit each computation in the set, once and only once, while following the lexicographical ordering between the computations. The Tiramisu code generator (which uses the ISL [51] library) takes Layer IV as input and generates an abstract syntax tree (AST). The AST is then traversed to generate lower level code targeting specific hardware architectures.

```
// Layer I
{bx(y,x): 0<=y<N ∧ 0<=x<M}:(in(y,x)+in(y,x+1)+in(y,x+2))/3);
{by(y,x): 0<=y<N ∧ 0<=x<M}:(bx(y,x)+bx(y+1,x)+bx(y+2,x))/3);
// Layer II
bx.split(y, chunk_sz, y1, y2); by.split(y, chunk_sz,y1,y2);
// Layer III
send s = create_send("{(q,y,x): 1<=q<N-1 ∧ 0<=y<2 ∧
     0<=x<M}", q-1 /*dest*/, {ASYNC,BLOCK}, bx(y,x));
recv r = create_receive("{(q,y,x): 0<=q<N-2 ∧ 0<=y<2 ∧
     0<=x<M}", q /*src*/, {SYNC,BLOCK}, bx(y,x));
bx.distribute(y1); by.distribute(y1);
s.distribute(q); r.distribute(q);
```

Figure 3: Tiramisu pseudocode for a 3x3 distributed blur

6.1 Multicore CPU

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

6.2 GPU (CUDA)

For GPU code generation, data copy commands and information about where to store buffers (shared, constant, or global memory) are all provided in Layer IV. Tiramisu translates these into the equivalent data copies and buffer allocations in the 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. The Tiramisu code generator can generate coalesced array accesses and can use shared and constant memories. It can also avoid thread divergence by separating full loop nests (loop nests with a size that is multiple of the tile size) from partial tile (the remaining part of a loop). The final output of the GPU code generator is an optimized CUDA code.

6.3 FPGA

Tiramisu relies on FROST [17] to generate code for FPGAs. FROST is a common back-end for accelerating DSLs using FPGAs. It exposes an IR for DSLs to target, as well as a high level scheduling language to express FPGA-specific optimizations. We use Tiramisu to perform loop nest transformations necessary for preparing code for lowering to FPGA, while FROST focuses on the actual lowering to the target High-Level Synthesis (HLS) toolchain. The output of FROST is C++ code suitable for HLS tools like Xilinx Vivado HLS [28].

6.4 Distributed Memory Systems

Tiramisu utilizes MPI to generate code for distributed memory systems. Figure 3 shows Tiramisu pseudocode for a 3x3

```
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

distributed box blur. Lines 2 and 3 define the blur computation. For this example, we want to distribute the computation such that each MPI *rank* (process) operates on contiguous rows of input data. Each rank gets `chunk_sz` rows. On line 5, 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.

Line 7 and 8 deal with communication. We assume that our image data is already distributed, thus only boundary rows need to be communicated among adjacent ranks. We use two-sided communication in TIRAMISU, meaning communication is done with pairs of *send* and *receive* statements. Line 7 defines an asynchronous blocking send operation to processor q-1. `create_send` takes as input the send iteration domain (which also defines the size of data to send), destination rank, communication type, and access into the producer. Line 8 defines the receive operation.

Line 9 tags dimension `y1` of `bx` and `by` as distributed, and line 10 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 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 such as tiling a transfer to more advanced transformations including specializing a distributed computation based on rank.

GPU and FPGA scheduling can also be composed with distribution, allowing programs to execute in a heterogeneous environment. For example, adding only a few extra scheduling commands to distributed TIRAMISU code enables 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 which computations run on the GPU. The resulting code can be used to distribute the box blur computation across multiple GPUs residing on different nodes. As with CPU distribution, we use MPI to control inter-node communication.

7 EVALUATION

We performed the evaluation on a cluster of dual-socket machines with two 24-core Intel Xeon E5-2680v3 CPUs, 128 GB RAM, Ubuntu 14.04, and an Infiniband interconnect. Each experiment is repeated 30× and the median time is reported.

7.1 Halide to TIRAMISU

To demonstrate the utility of TIRAMISU, we create a version of Halide[41], an industrial-quality DSL for image processing, that uses TIRAMISU for transformations. 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 Layer I computation. Similarly, we map Halide scheduling directives, such as tiling, splitting, and reordering to the equivalent high level scheduling commands in TIRAMISU. Finally, we map computations to buffer elements using default Halide mappings. For CPU code generation, we convert Layer IV TIRAMISU IR into low-level transformed Halide IR (bypassing all lowering in the original Halide compiler) and feed it into the Halide LLVM code generator. In contrast, for GPU code generation, we generate CUDA source directly from TIRAMISU IR.

We used the following benchmarks in our evaluation: `cvtColor`, which converts an RGB image 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`, a synthetic pipeline composed of 4 stages and that computes a negative and a brightened image from the same input image; `rgbyuv420`, an image conversion from RGB to YUV420; `edgeDetector`, a ring blur followed by Roberts edge detection [43]; and `ticket #2373`, a code snippet from a bug filed against Halide where the inferred bounds are over-approximated, causing the generated code to fail due to an assertion during execution. Some of these kernels, such as `warpAffine` and `resize`, contain non-affine array accesses and non-affine conditionals for clamping. We used a 2112×3520 RGB input image for the experiments.

Figure 5 compares the execution time of code generated by Halide and Halide-TIRAMISU. In six of the benchmarks the performance of the code generated by Halide-TIRAMISU matches the performance of Halide. We use the same schedule for both implementations; these schedules were handwritten by Halide experts. The results for `warpAffine` and `resize` show that TIRAMISU handles non-affine array accesses and conditionals efficiently.

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

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

`edgeDetector` cannot be implemented in Halide because it creates a cyclic dependence graph with a cycle length

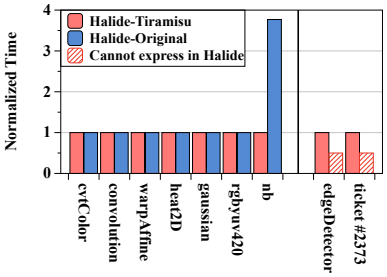

Figure 5: CPU execution time comparison between Halide-Original and Halide-Tiramisu

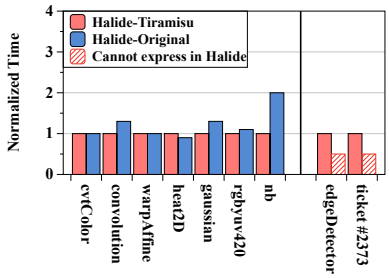

Figure 6: GPU kernel execution times for Halide-Original and Halide-Tiramisu

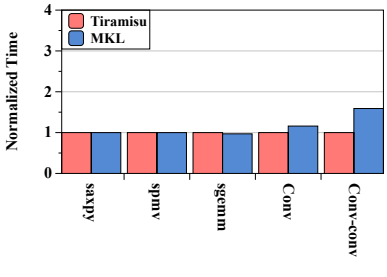

Figure 7: Comparing CPU code generated from Tiramisu with Intel MKL

≥ 1. Halide can only express programs with an acyclic dependence graph, with some exceptions; this restriction is imposed by the Halide language and compiler to avoid the need to prove the legality of some optimizations (since proving the legality of certain optimizations is difficult in the Halide interval-based representation). Tiramisu does not have this restriction since it checks transformation legality using dependence analysis [20].

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

For nb, the code generated from Halide-Tiramisu achieves almost 4× speedup over the Halide-generated code. This is primarily due to loop fusion. In this code, Tiramisu enhances data locality by fusing loops into one loop; this is not possible in Halide, which cannot fuse loops if they update the same buffer. Halide makes this conservative assumption because otherwise it cannot prove the fusion is correct. This is not the case for Tiramisu which uses dependence analysis to prove correctness.

We generated GPU code (CUDA) from all the filters and compared our code with the original Halide compiler, which generates PTX to target Nvidia GPUs. We compared both the data copy times and kernel execution times. Figure 6 shows a comparison between kernel execution times. Tiramisu-generated kernel execution for `convolution` and `gaussian` is faster because code generated by Tiramisu uses constant memory to store the weights array, while the current version of Halide does not use constant memory for its PTX backend. Data copy times (which we elide for brevity), for all the filters, are the same for Halide-Tiramisu and Halide.

7.2 Linear Algebra and DNN Kernels

We also evaluated Tiramisu by implementing a set of linear algebra and neural network kernels, including `saxpy` ($Y = \alpha X + Y$), `spmv` ($y = Ax$ with A sparse), `sgemm` ($C = \alpha AB + \beta C$), `conv` (a neural network convolution layer), and `conv-conv` (two conv layers fused together). Figure 7 shows a comparison between the performance of CPU code generated by Tiramisu and the Intel MKL library. For linear algebra we use matrices of size 1060×1060 and vectors of size 1060 while for DNN we use 512×512 as the data input size, 16 as the number of input/output features and a batch size of 32.

For `saxpy`, `spmv` and `sgemm`, Tiramisu matches the performance of Intel MKL. The comparison between the Tiramisu implementation of `sgemm` and Intel MKL is interesting in particular because the Intel MKL implementation of this kernel is well-known for its hand-optimized performance. We used a large set of optimizations to match Intel MKL. These optimizations include two-level blocking of the three-dimensional `sgemm` loop, fusing the computation of $T = \alpha AB$ and $C = T + \beta C$ into a single loop, vectorization, unrolling, array packing (as described in [22]), register blocking, and separation of full and partial tiles (which is crucial to enable vectorization, unrolling, and reduce control overhead). We also used auto-tuning [2] to find the best tile size, unrolling factor and vector length for the machine on which we run our experiments. For the `conv` kernel, Tiramisu outperforms the Intel MKL implementation due to the tuning of vector size, unrolling factor and tile size. In `conv-conv`, Tiramisu fuses the two convolution loops which improves data locality.

7.3 Evaluating the FPGA Backend

We evaluated the FPGA backend in Tiramisu using six 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 [27], a library that implements several OpenCV functions for FPGA. We compare the execution time of hardware designs generated from Tiramisu with those extracted from the Vivado HLS Video Library. We synthesized these designs using the Xilinx SDAccel 2016.4 toolchain at 200MHz and ran on an ADM-PCIE-7V3 board by Alpha Data (powered by

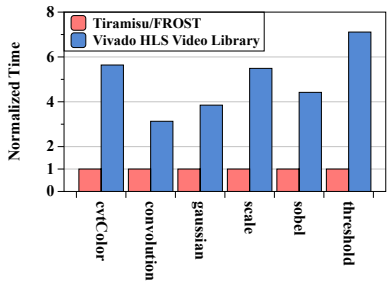

Figure 8: Execution time comparison for TIRAMISU/FROST and Vivado HLS Video Library

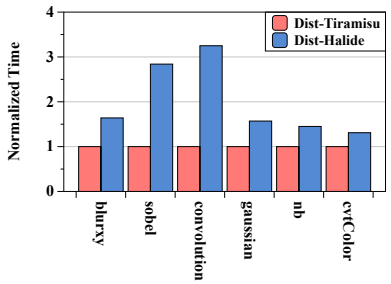

Figure 9: Comparing distributed TIRAMISU and distributed Halide (16 nodes)

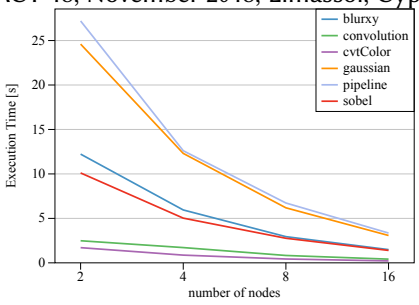

Figure 10: Execution time of distributed TIRAMISU for 2, 4, 8, and 16 nodes

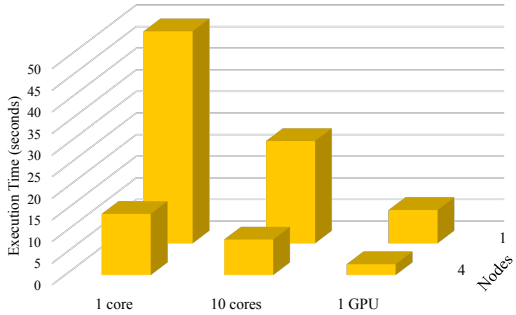

Figure 11: Results for either CPU or GPU running on a single node (back row), and distributed across 4 nodes (front row).

a Xilinx Virtex 7 FPGA). For all kernels, we use a 512×384 RGB image, except for the `threshold` kernel, which uses a single channel image as input.

The HLS Video Library kernels are parallelized on the channel dimension. This is done to maintain consistency between functions of the OpenCV library. When we parallelized our kernels on the channel dimension, in a way similar to the HLS video library, TIRAMISU was able to match HLS video library performance. While the HLS Video Library provides only one version of the code parallelized on the channel dimension, the flexibility of TIRAMISU scheduling commands allowed us to explore other alternatives, including parallelization over the width dimension of the image, and this led to better performance (at the expense of more FPGA resources). Figure 8 shows the results of the evaluation. For each kernel, we used TIRAMISU to arrange the input image in a plane interleaved manner (i.e. width as innermost dimension of the image) and split the innermost loop to prepare for vectorized computation. Then, we relied on scheduling commands to generate pipelined FPGA designs able to perform both vectorized load/store from the off-chip memory (we applied a 512-bit vectorization), and vectorized computations (each design performs 64 parallel computations per clock cycle). While the difference in performance is not due to a fundamental limitation of the HLS video library, this experiment shows that TIRAMISU is (1) suited to target FPGAs; (2) flexible enough to allow the exploration of different optimizations; and (3) can match the performance of a hand-written library.

7.4 Evaluating the Distributed Backend

For the TIRAMISU distributed backend, we used 6 kernels for evaluation: `blurxy`, `sobel`, `convolution`, `gaussian`, `nb`, and `cvtColor` (we chose these kernels because we wanted to compare with the distributed Halide compiler [18] and these kernels are already implemented in that compiler). We assume the data are already distributed across the nodes by rows. Of these benchmarks, `nb`, and `cvtColor` do not require any communication; the other four require communication due to overlapping boundary regions in the distributed data. For these distributed CPU-only tests, we use the MVAPICH2 2.0 [26] 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: it overestimates the amount of data it needs to send, and 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. This graph shows that distributed code generated from TIRAMISU scales well as the number of nodes increases (strong scaling).

7.5 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. We ran on a cluster of 4 nodes, each consisting of an Nvidia K40 GPU, and a 12-core Intel Xeon E5-2695 v2 with OpenMPI 1.6.5 [37].

Figure 11 shows the results of this experiment. The back row shows the results for running the *cvtColor* kernel on one node, using 1 core, 10 cores, or 1 GPU. As expected, 10 cores is better than 1 core. It also shows that the GPU outperforms 10 CPU cores. The front row shows the same configuration, except that it is 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 40 CPU cores.

7.6 Evaluation Summary

Overall, the experiments demonstrated the use of Tiramisu as an optimization framework for DSLs and for implementing a set of linear algebra and DNN kernels, all for multiple backends. We show that Tiramisu is expressive: it allows Halide to implement new optimizations and algorithms. The experiments also show that Tiramisu is suitable for targeting multiple hardware architectures, such as multicore CPUs, GPUs, distributed systems, and FPGAs. Thanks to its flexible scheduling commands, it generates highly optimized code for a variety of of architectures and algorithms.

8 CONCLUSION

In this paper we introduce Tiramisu, an optimization framework that separates the algorithm, the schedule, the data layout and the communication using a four-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.

We evaluate Tiramisu by integrating it as a pass in Halide and by implementing linear algebra and DNN kernels and by targeting a variety of backends. We demonstrate that Tiramisu allows Halide to implement new optimizations and algorithms and that Tiramisu can generate efficient code for multiple hardware architectures.

REFERENCES

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

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada.

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

[4] 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

[5] 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

[6] 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.

[7] 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.

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

[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] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[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] 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

[15] 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

[16] 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

[17] 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)*.

[18] 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.

[19] 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

[20] 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

[21] 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.

[22] 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

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

[24] 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.

[25] 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.

[26] 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.

[27] Xilinx Inc. 2015. HLS Video Library. http://www.wiki.xilinx.com/HLS+Video+Library. (April 2015).

[28] Xilinx Inc. 2017. Vivado HLx Editions. https://www.xilinx.com/products/design-tools/vivado.html. (October 2017).

[29] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. 1993. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing (Supercomputing '93)*. ACM, New York, NY, USA, 262–273. https://doi.org/10.1145/169627.169724

[30] 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

[31] 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

[32] 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.

[33] 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.

[34] 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

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

[36] 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

[37] MPI Open. [n. d.]. Version 1.6. 5, Open MPI Software. ([n. d.]).

[38] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford Inf oLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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

[45] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 813–824.

[46] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17)*. IEEE Press, Piscataway, NJ, USA, 74–85. http://dl.acm.org/citation.cfm?id=3049832.3049841

[47] 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.

[48] 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.

[49] 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

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

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

[52] 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.

[53] 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.

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