TIRAMISU: A POLYHEDRAL COMPILER FOR EXPRESSING FAST AND PORTABLE CODE PAPER ANALYSIS

전기정보공학부 김희원

Overview

- **Tiramisu is a polyhedral compiler for high-performance code generation for multiple platforms including multicore CPUs, GPUs, and distributed machines**
- Tiramisu uses 4-level intermediate representation that allows full seperation between the algorithms, loop transformations, data layouts, and communication.
- Evaluation is done by: writing a set of image processing, deep learning, linear algebra benchmarks.
  Compared with state-of-the-art compilers and hand-tuned libraries.
  Tiramisu outperformed/matched existing compilers/libraries on different hardware architectures(multicore CPUs, GPU, distributed machines).

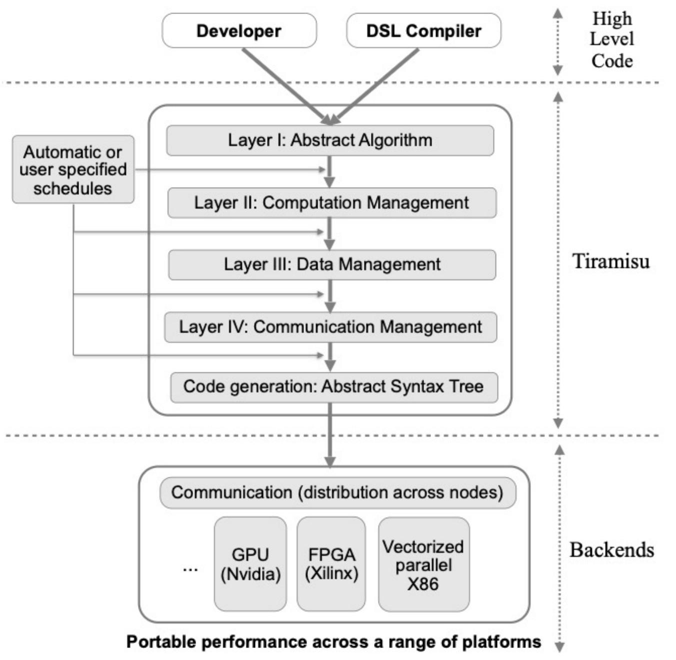
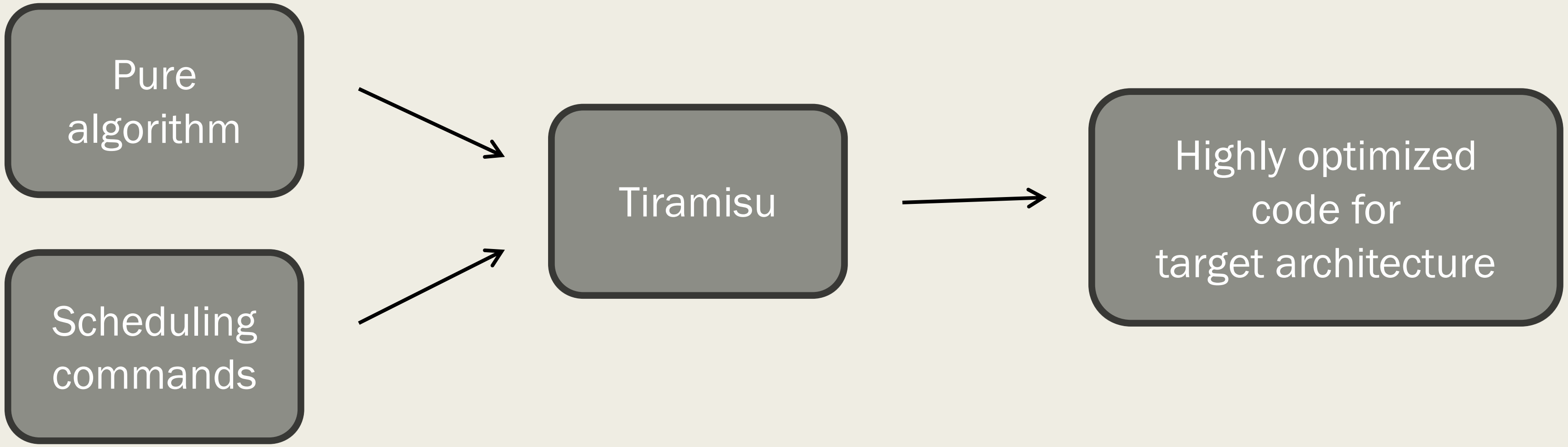


Fig. 4: TIRAMISU overview

How Tiramisu works

- Tiramisu is a Domain-Specific Language(DSL) embedded in C++, providing C++ APIs that allows us to write a high-level and architecture-independent algorithm + a set of scheduling commands that guide code generation.
- Given the algorithm and scheduling commands, Tiramisu generates optimized backend codes.

Tiramisu is designed for expressing data parallel algorithms (especially those that operate over dense arrays using loop nests and sequence of statements)



4 Types of Scheduling Commands

- <u>Type 1) Commands for loop nest transformations</u>

Changes the loop structure to optimize access to data. Only about "how the loop is executed".

Ex) **Tiling**- divides the loop into block units for locality and cache efficiency

`C.tile(i,j,t1,t2, i0,j0,i1,j1)`	Tile the loop levels (i, j) of the computation C by $t1 \times t2$. The names of the new loop levels are (i0, j0, i1, j1) where i0 is the outermost loop level and j1 is the innermost.

```
// Loop without tiling
for (int i = 0; i < 1000; i++)
  for (int j = 0; j < 1000; j++)
    A[i][j] += 1;
```

→

```
// Loop with tiling
for (ii = 0; ii < 1000; ii += 32)
  for (jj = 0; jj < 1000; jj += 32)
    for (i = ii; i < ii+32; i++)
      for (j = jj; j < jj+32; j++)
        A[i][j] += 1;
```

Loop without tiling is slow because all the memory is split for a million iterations. (Cache miss)

Loop with tiling is faster since it moves a chunk to a cache and executes using that chunk. So, there is no need to go to the memory a million times (Cache hit)

4 Types of Scheduling Commands

- <u>**Type 2) Commands for mapping loop levels to hardware**</u>

Decides what loop to map to which physical execution unit (CPU, multithread, GPU, SIMD, …).

Ex) **Parallelizing**

`C.parallelize(i)`	Parallelize the i loop level for execution on a shared memory system.

```
for (int i = 0; i < 1000; i++) {
    A[i] += 1;
}
```

For the above code, since all A[i]'s are independent,
parallelize(i) makes multiple CPU cores to compute different A[i]'s at the same time.

This is possible since A[i]'s are independent to each other.
(It is not parallelizable if A[i] is dependent to A[i-1], etc.)

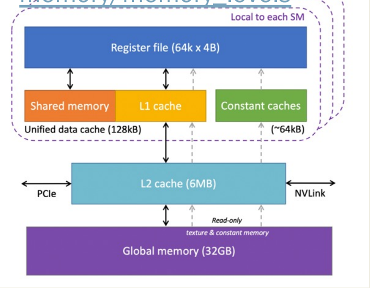
4 Types of Scheduling Commands

- <u>Type 3) Commands for manipulating data</u>

Controls allocation, storage location, movement, and properties of buffers.

Ex) **cache_shared_at**: caches some data at GPU's shared memory so that the time to fetch that data gets faster.

https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_levels



`C.cache_shared_at(P,i)`	Cache (copy) the buffer of C in shared memory. Copying from global to shared GPU memory will be done at loop level i of the computation P. The amount of data to copy, the access functions, and synchronization are computed automatically.

- <u>Type 4) Commands for adding synchronization operations</u>

Ex) **allocate_at:** Allocates buffer 'b' at 'i'-th iteration of 'p' loop.

`b.allocate_at(p, i)`	Return an operation that allocates b at the loop i of p. An operation can be scheduled like any computation.

Specifies when to get the memory. If we do not use "allocate_at", then the memory remains allocated for the entire loop.
If we use "allocate_at", we only allocate the memory at the start of the i-th iteration,
and deallocate it at the end of the i-th iteration.

This prevents memory waste.

Structure of the Tiramisu compiler

- **Why multi-layer intermediate representation (IR) is needed:** Most current IRs use memory to share data between different statements of the program. Therefore, programs need to first decide data layout(how data will be allocated in the memory) before deciding on optimizations and mapping to hardware. Since the layout is already fixed, it is hard to change the layout for optimization.
- Tiramisu's multi-layer structure ensures the compiler need not to worry about the earlier layer.

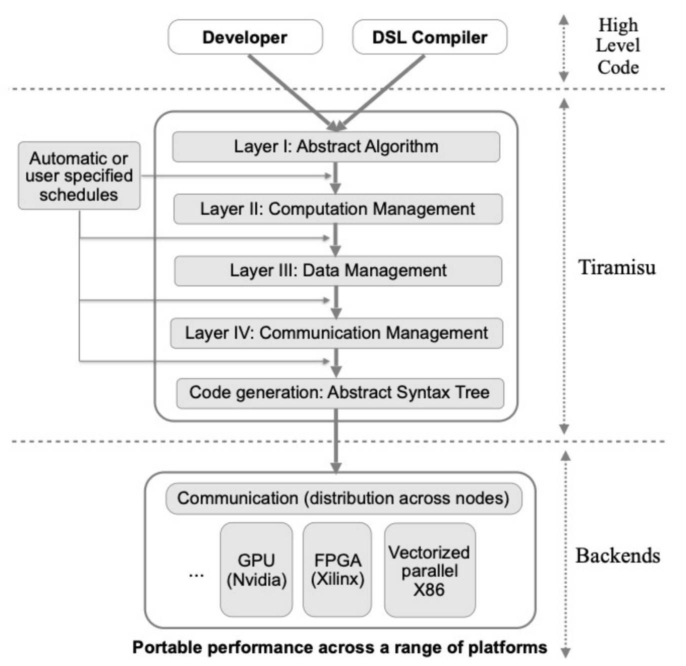


Fig. 4: TIRAMISU overview

Structure of the Tiramisu compiler

```
// Declare the iterators i, j and c.
Var i(0, N-2), j(0, M-2), c(0, 3);

Computation bx(i, j, c), by(i, j, c);

// Algorithm.
bx(i,j,c) = (in(i,j,c)+in(i,j+1,c)+in(i,j+2,c))/3;
by(i,j,c) = (bx(i,j,c)+bx(i+1,j,c)+bx(i+2,j,c))/3);
```

- **Layer 1- Abstract Algorithm**

Specifies the algorithm only.

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

- **Layer 2- Computation Management**

Specifies the order of execution of computations, and the processor on which they execute.
(Doesn’t consider how to allocate to memory.)
Processor type: CPU / node (in a distributed system) / GPU thread dimension / GPU block dimension

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

Structure of the Tiramisu compiler

- **Layer 3- Data Management**

Maps each computation to a buffer element (access relation: maps from iteration domain to memory index domain), and allocate or deallocate buffers.

Access relations are usually represented in affine relations.

Left side (Iteration Domain):
scheduled iterators in ‘by’ computation of 1-th iteration / i0, j0 of GPU block index / i1, j1 of GPU thread index

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

Right side (Memory index domain): 3D array that stores the results-
Indices: i0 block * tile size 32 + i1 thread, j0 block * tile size 32 + j1 thread
+ Constraints after “:”

- **Layer 4- Communication Management**

Adds synchronization and communication operations. Decides <u>exactly when </u>to do synchronization / communication / memory allocation / memory deallocation

Compiler Implementation

- **Layer 1 to Layer 2 transformation:** done using two types of scheduling commands

1) Commands for loop nest transformations: transforms the iteration domain

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

2) Commands for mapping loop levels to hardware: adds space tags to dimensions to indicate which loop levels to parallelize/ vectorize/ map to GPU blocks/ so on.

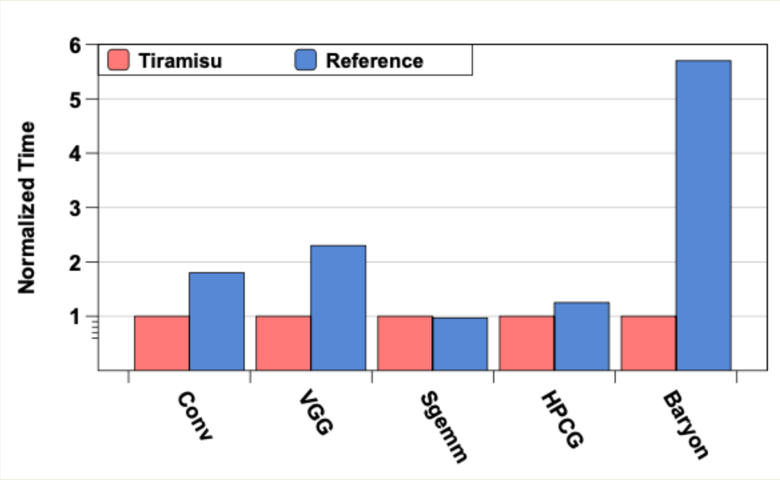
- **Layer 2 to Layer 3 transformation:** done by augmenting layer 2 with access relations

Ex) buffer allocations b.allocate_at(C, i): At the i-th iteration, computation C is allocated at buffer b.

- **Layer 3 to Layer 4 transformation:** Scheduling commands for data communication, synchronization, and for copying data are all translated into statements.
- **Layer 4 to an AST (Abstract Syntax Tree)**
- **AST to a lower level code for specific hardware architectures:**
  - *If the target is a multicore CPU: AST → LLVM IR (Halide used)*
  - *If the target is a GPU: AST→ LLVM IR (for the host code) + CUDA (for the kernel code)*
  - *If the target is a distributed memory system: uses MPI*

Evaluation

- Two criteria:
  1) deep learning and linear algebra benchmarks, 2) image processing benchmarks
- **Deep learning and linear algebra benchmarks:** matches or outperforms Intel MKL.
- **Image processing benchmarks:** matches or outperforms Halide and Pencil.
- These outstanding results are due to the architecture that separates the four IR layers.



1) Deep learning and linear algebra

Architectures	Frameworks	Benchmarks						
		edge Detector	cvtColor	Conv2D	warp Affine	gaussian	nb	ticket #2373
Single-node multicore	Tiramisu	1	1	1	1	1	1	1
	Halide	-	1	1	1	1	3.77	-
	PENCIL	2.43	2.39	11.82	10.2	5.82	1	1
GPU	Tiramisu	1.05	1	1	1	1	1	1
	Halide	-	1	1.3	1	1.3	1.7	-
	PENCIL	1	1	1.33	1	1.2	1.02	1
Distributed (16 Nodes)	Tiramisu	1	1	1	1	1	1	1
	Dist-Halide	-	1.31	3.25	2.54	1.57	1.45	-

2) Image processing

Conclusion

- Tiramisu can generate **high-performance code** for **multiple platforms** using 4-layer IRs, for **various areas** of image processing, stencils, linear algebra, and deep learning.
- The **4-layer IRs** of Tiramisu consists of the following:
  Layer 1: Abstract Algorithm
  Layer 2: Computation Management
  Layer 3: Data Management
  Layer 4: Communication Management
  All layers do not influence other layers.
- Most **current IRs** use memory to communicate between program statements. This **forces** compilers to **choose data layout before** decideing optimizations and mappings. Therefore, **this data layout must be undone before scheduling** to allow more freedom for scheduling, which is a challenging procedure. Tiramisu's fully separated multi-layer IR **solves this issue.**
- While **targeting a variety of backends**, Tiramisu **outperforms or matches** existing state-of-the-art frameworks and hand-tuned code on two criteria:
  1) Deep learning and linear algebra benchmarks
  2) Image processing benchmarks
- Tiramisu is the only **open source** DNN compiler that **optimizes sparse DNNs.** (https://tiramisu-compiler.org/)