Tiramisu: A Code Optimization Framework for High Performance Systems

Presented by Nicholas Christensen
Motivation

Performance of different frameworks on generalized matrix multiplication $C = \alpha A B + \beta C$
Overview

• Polyhedral compiler with scheduling language
• Evolves IR over four layers
• Matches or outperforms existing compilers on different architectures
## Framework comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>Tiramisu</th>
<th>AlphaZ</th>
<th>PENCIL</th>
<th>Pluto</th>
<th>Halide</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>GPU code generation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Distributed CPU code generation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Distributed GPU code generation</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Support all affine loop transformations</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Optimize data accesses</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Commands for loop transformations</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Commands for optimizing data accesses</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Commands for communication</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Commands for memory hierarchies</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Limited</td>
</tr>
<tr>
<td>Expressing cyclic data-flow graphs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Support non-rectangular iteration spaces</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Limited</td>
</tr>
<tr>
<td>Instance-wise, exact dependence analysis</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Compile-time affine-set emptiness check</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Implement support for parametric tiling</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The Tiramisu embedded DSL

- Embedded in C++
- Express high level algorithm
- Scope
  - Data parallel algorithms
  - Operations on dense arrays
Specifying the algorithm

- Pure function with inputs, outputs, sequence of statements
- Flow-control limited to for-loops and conditionals
- User provides iteration domain and expression to compute

Blur algorithm in Tiramisu DSL (left) and code equivalent (right)
Scheduling commands

- Loop nest transformations (Layer II)
  - \( C.tile(i, j, 32, 32, i0, j0, i1, j1) \)
- Map loop levels to hardware (Layer II)
  - \( C.vectorize(j, 4) \)
- Data manipulation (Layer III)
  - Allocation, setting properties, copying, accessing
    - \( b.allocate_at(P, i) \)
- Synchronization and communication (Layer IV)
  - \( barrier_at(P, i) \)

* C and P are computations, b is a buffer, i and j are loop iterators
Scheduling commands example

... 

tiramisu::init("function0");  // Initialize compiler and declare a function

// Layer I: provide the algorithm.  // Equivalent to
var i("i", 0, 10);  // for (i = 0; i < 10; i++)
computation S0("S0", {i}, 3 + 4);  // S0(i) = 3 + 4;

// Layer II: specify how the algorithm is optimized.
S0.parallelize(i);

// Layer III: allocate buffers and specify how computations stored
buffer buf0("buf0", {10}, p_uint8, a_output);
S0.store_in(&buf0);

// Code Generation
tiramisu::codegen({&buf0}, "build/generated_fct_developers_tutorial_01.o");
...

The Tiramisu IR

- Optimizations restricted by data layout, memory-based dependencies
- Data layouts specified before scheduling end up undone for more scheduling freedom (difficult)
- Idea: Separate algorithm from architecture details
  - Then apply scheduling commands in four discrete phases ("layers")
  - Don’t need to worry about undoing work of previous layers
- Integer sets represent IR layers
- Maps represent transformations on iteration domain and data layout
Layer I – Abstract Algorithm

- Unordered computation
- No notion of data location
  - Simple producer-consumer communication

\[
\{ \text{by}(i, j, c) : 0 \leq i < N - 2 \land 0 \leq j < M - 2 \land 0 \leq c < 3 \} : \\
(bx(i, j, c) + bx(i + 1, j, c) + bx(i + 2, j, c))/3
\]

Layer I representation of blur
Layer II – Computation Management

- Specifies execution order and location
  - Tags designate execution location
  - Lexicographical ordering of computations

{ by(1, i0(gpuB), j0(gpuB), i1(gpuT ), j1(gpuT ), c) :
  i0 = floor (i / 32) ∧ j0 = floor (j / 32) ∧
  i1 = i % 32 ∧ j1 = j % 32 ∧ 0 ≤ i < N − 2 ∧ 0 ≤ j < M − 2 ∧ 0 ≤ c < 3
}:
// Computation
(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

Layer II blur, specifies execution location on GPU and tiling
Layer III – Data Management

- Specifies location of intermediate values
- Adds buffer allocation/deallocation commands
- Affine mapping of Layer II computations to buffer elements

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

Layer III mapping generated from by.store_in(c, i, j)
Layer IV – Communication Mgmt.

- Maps synchronization, communication to time-space domain
- Determines buffer allocation/deallocation placement
Generating code with Tiramisu

- Transforms code through seven stages
  - Algorithm → Layer I
  - Layer I + loop nest/loop tagging commands → Layer II
  - Layer II + data layout mapping → Layer III
  - Layer III + memory hierarchy mapping → Layer IV
  - Layer IV → Abstract syntax tree
  - AST → Generated code (target dependent)
Layer I to Layer II

• Loop nest transformations
  - e.g. tile(), split(), shift(), interchange()
  - Applies map to Layer I that transforms iteration domain

\[ \{ \text{by}(i, j, c) \rightarrow \text{by}(i_0, j_0, i_1, j_1, c) : i_0 = \text{floor}(i / 32) \land i_1 = i \% 32 \land j_0 = \text{floor}(j / 32) \land j_1 = j \% 32 \land 0 \leq i < N \land 0 \leq j < N \} \]

• Mapping loop levels to hardware
  - e.g. parallelize(), vectorize(), gpuT()
  - Adds space tags to dimensions
Layer II to Layer III

- Augments Layer II with access relations
  - Typically identity relations (computation $C(i,j) \rightarrow \text{buffer } C[i, j]$)

- Adds buffer allocation statements
  - e.g. $b.$allocate_at$(C, i)$
  - Automatically deduces iteration domains
Layer III to Layer IV

- Translates communication, synchronization, memory mapping commands into statements
- e.g. `tag_gpu_global()`, `host_to_device()`, `send()`, `receive()`
Code Generation

• Relies on Cloog code generation algorithm
• Generates AST from Layer IV IR
• Traverses AST to generate code for specific HW
  - Multicore CPU → LLVM IR
  - GPU → LLVM IR (host), CUDA (device)
  - Distributed memory systems → MPI
### Evaluation

<table>
<thead>
<tr>
<th>Architectures</th>
<th>Frameworks</th>
<th>edgeDetector</th>
<th>cvtColor</th>
<th>convolution</th>
<th>warpAffine</th>
<th>gaussian</th>
<th>nb</th>
<th>ticket #2373</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-node multicore</td>
<td>Tiramisu</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3.77</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>PENCIL</td>
<td>2.43</td>
<td>2.39</td>
<td>11.82</td>
<td>10.2</td>
<td>5.82</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GPU</td>
<td>Tiramisu</td>
<td>1.05</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Halide</td>
<td>-</td>
<td>1</td>
<td>1.3</td>
<td>1</td>
<td>1.3</td>
<td>1.7</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>PENCIL</td>
<td>1</td>
<td>1</td>
<td>1.33</td>
<td>1</td>
<td>1.2</td>
<td>1.02</td>
<td>1</td>
</tr>
<tr>
<td>Distributed (16 Nodes)</td>
<td>Tiramisu</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Dist-Halide</td>
<td>-</td>
<td>1.31</td>
<td>3.25</td>
<td>2.54</td>
<td>1.57</td>
<td>1.45</td>
<td>-</td>
</tr>
</tbody>
</table>

Execution times normalized by the fastest framework
Strong scaling
Conclusion

- Dependency analysis gives Tiramisu advantages over Halide
- Equivalent performance to hand-tuned Halide in many cases
- Does heavy lifting, but programmer expertise still required
- Would like to see strong scaling at high node count, performance on harder problems
- Further development – FPGA backend (Del Sozzo et. al., 2018)
// Scheduling commands for targeting GPU.
// Tile i and j and map the resulting dimensions to GPU
var i0, j0, i1, j1;
by.tile_gpu(i, j, 32, 32, i0, j0, i1, j1);
by.compute_at(by, j0);
by.cache_shared_at(by, j0);

// Use struct-of-array data layout for bx and by.
bx.store_in({c, i, j}); by.store_in({c, i, j});

// Create data copy operations
operation cp1 = in.host_to_device();
operation cp2 = by.device_to_host();

// Specify the order of execution of copies
cp1.before(bx, root); cp2.after(by, root);

host_to_device_copy(in_host, in);
GPUBlock for(i0 in 0..floor((N-2)/32))
GPUBlock for(j0 in 0..floor((M-2)/32))
shared bx[3, 32, 34];
GPUThread for(i1 in 0..min(N-2, 32+2)) // Tiling with redundancy
GPUThread for(j1 in 0..min(M-2, 32+2)) // Tiling with redundancy
int i = i0*32+i1
int j = j0*32+j1
for (c in 0..3)
  bx[c][i][j] =
    (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
GPUThread for(i1 in 0..min(N-2, 32+i0+31))
GPUThread for(j1 in 0..min(M-2, 32+j0+31))
for (c in 0..3)
  by[c][i][j]=(bx[c][i][j]+bx[c][i+1][j]+bx[c][i+2][j])/3
device_to_host_copy(by, bv_host):
Distributed scheduling example

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

// Declare additional iterators
var qs(1,Ranks), qr(0,Ranks-1), q, z;

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

// Create communication and order execution
send(s = send({qs}, in(0,0,0), N×2×3, qs-1, {ASYNC}));
recv r = receive({qr}, in(0,N,0), N×2×3, qr+1, {SYNC});
s.before(r,root); r.before(bx,root)

// Distribute the outermost loops
bx.distribute(q); by.distribute(q);
s.distribute(qs); r.distribute(qr);
```

// The array in[][][] is initially distributed across nodes
// Exchange ghost zones of in[][][]
distributed for (qs in 1..Ranks)
  send(in(0,0,0), N×2×3, qs-1, {ASYNC})
distributed for (qr in 0..Ranks-1)
  recv(in(0,N,0), N×2×3, qr+1, {SYNC})

distributed for (q in 0..Ranks)
  parallel for (z in 0..N/Ranks)
    int i = q*(N/Ranks) + z
    for (j in 0..M)
      for (c in 0..3)
        bx[i][j][c] = (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
distributed for (q in 0..Ranks)
  parallel for (z in 0..N/Ranks)
    int i = q*(N/Ranks) + z
    for (j in 0..M)
      for (c in 0..3)
        by[i][j][c] = (bx[i][j][c]+bx[i+1][j][c]+bx[i+2][j][c])/3

// We assume that no gather operation on by[][][] is needed
```