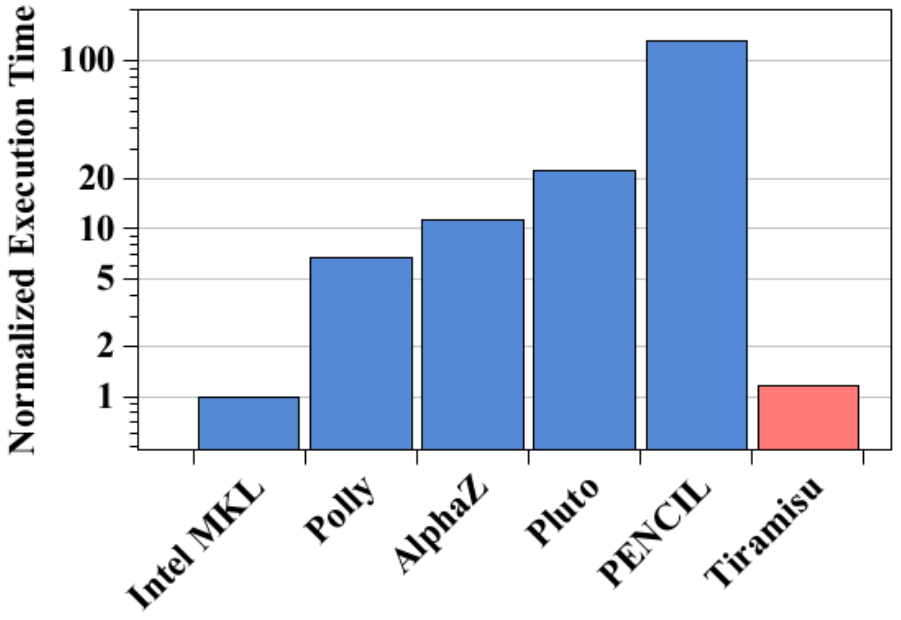
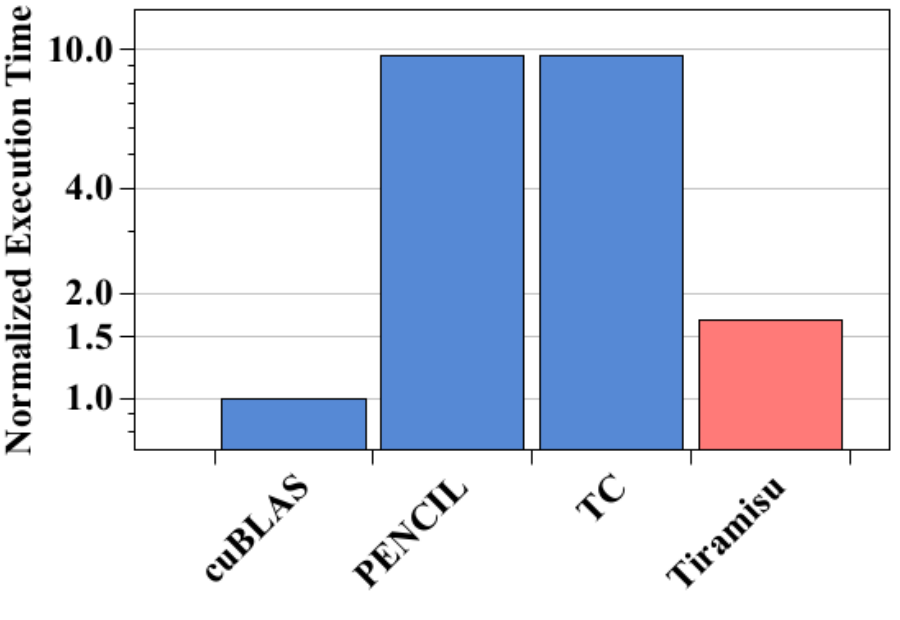
Riyadh Baghdadi et. al.

Tiramisu: A Code Optimization Framework for High Performance Systems

Presented by Nicholas Christensen

Motivation

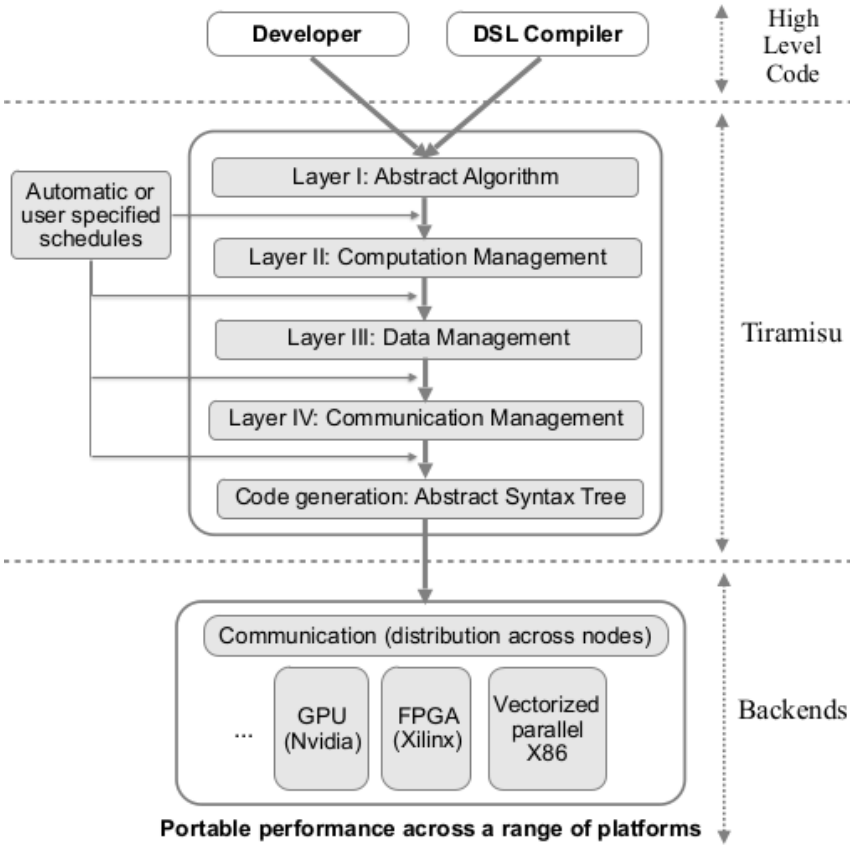




Performance of different frameworks on generalized matrix multiplication $C = \alpha AB + \beta C$

Overview

- Polyhedral compiler with scheduling language
- Evolves IR over four layers
- Matches or outperforms existing compilers on different architectures



Framework comparison

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

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

```
// Declare the iterators i, j and c.

var i(0, N-2), j(0, M-2), c(0, 3);

// 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);
```

```
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      bx[i][j][c] = (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      by[i][j][c] = (bx[i][j][c]+bx[i+1][j][c]+bx[i+2][j][c])/3
```

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)*
- Sychronization 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

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

Layer I representation of blur

Layer II – Computation Management

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

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

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

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

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

$$\{ by(i, j, c) \rightarrow by(i0, j0, i1, j1, c) : i0 = floor\ (i / 32) \wedge i1 = i\%32 \wedge j0 = floor\ (j/32) \wedge j1 = j\%32 \wedge 0 \leq i < N \wedge 0 \leq j < N \}$$

- 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) → 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

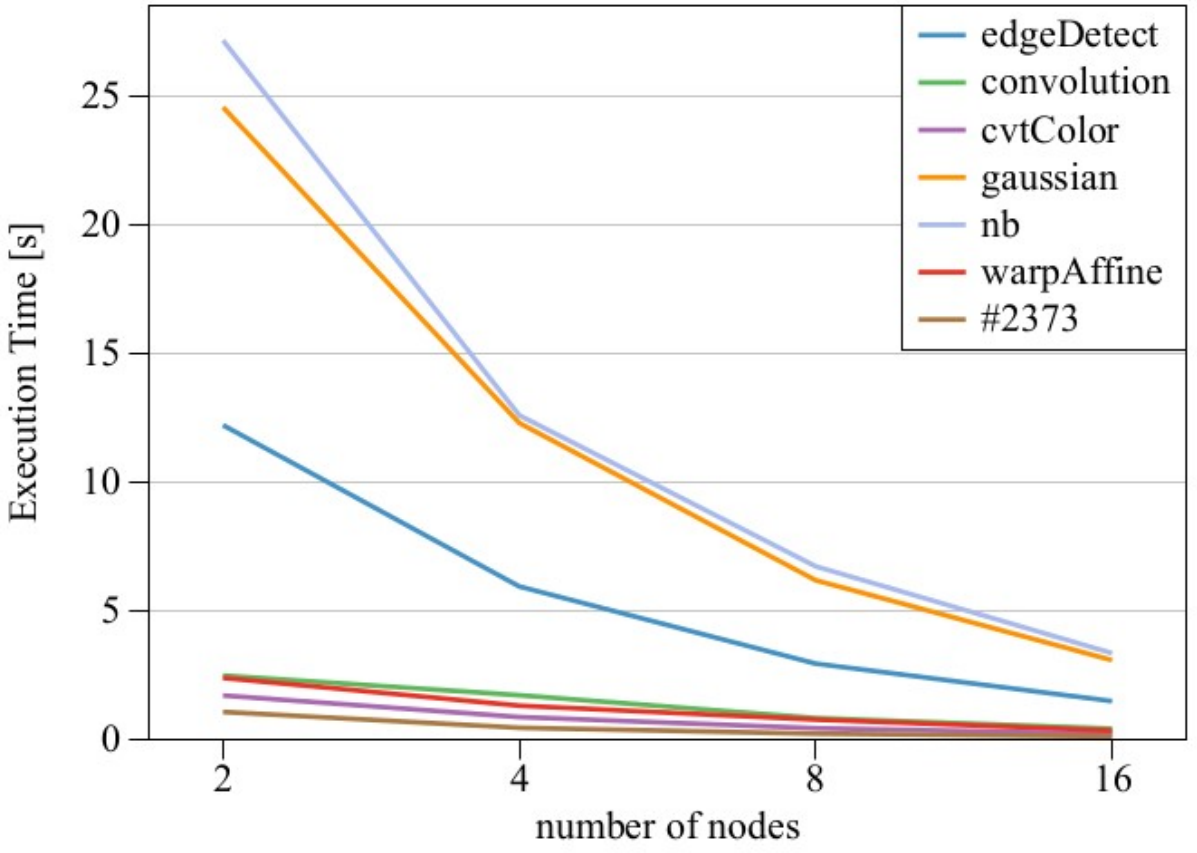
- Relies on Cloog code generation algorithm
- Generates AST from Layer IV IR
- Traverses AST to generate code for specific HW
  - Multicore CPU → LLVM IR
  - GPU → LLVM IR (host), CUDA (device)
  - Distributed memory systems → MPI

Evaluation

Architectures	Frameworks	Benchmarks						
		edge Detector	cvtColor	convolution	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	-

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)

Markus Mitterauer (CC-BY-SA-2.5)

GPU scheduling example

(a)

```
// 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);
bx.compute_at(by, j0);
bx.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][i1][j1]=
       (in[i][j][c]+in[i][j+1][c]+in[i1][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, by_host);
```

Distributed scheduling example

```
// 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);
s.before(r,root); r.before(bx,root)

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

(b)

```
// 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
```