Compiling Dense and Sparse Neural Networks using Tiramisu

Riyadh Baghdadi, J. Ray, E. Del Sozzo, Y. Zhang, P. Suriana, S. Kamil, M. Ben Romdhane, A. Akkas, A. Renda, J. Elliott Frankle, K. Abdous, A. N. Debbagh, F. Z. Benhamida, T. Arbaoui, B. Karima, Michael Carbin, **Saman Amarasinghe**

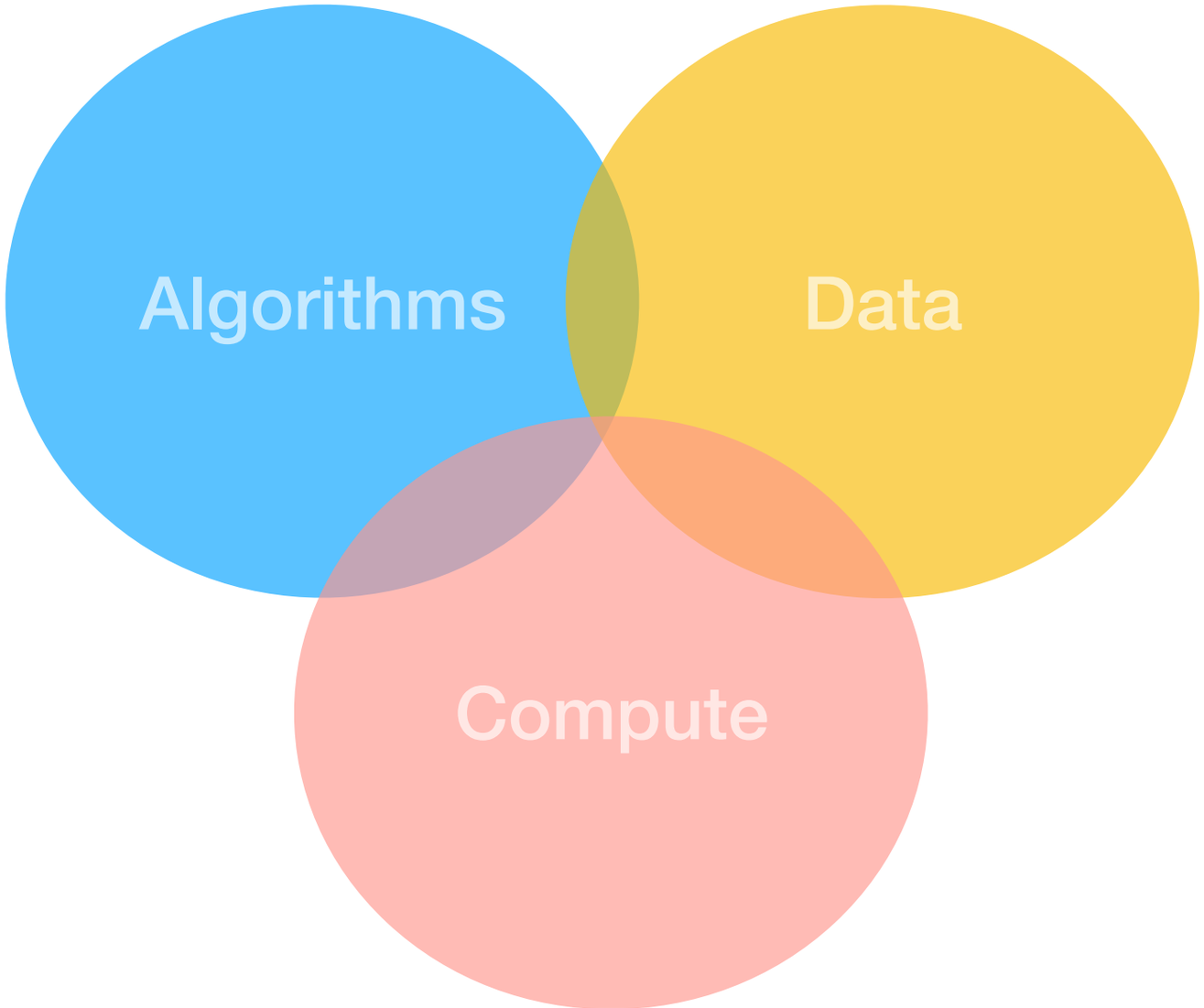
Massachusetts Institute of Technology

Google

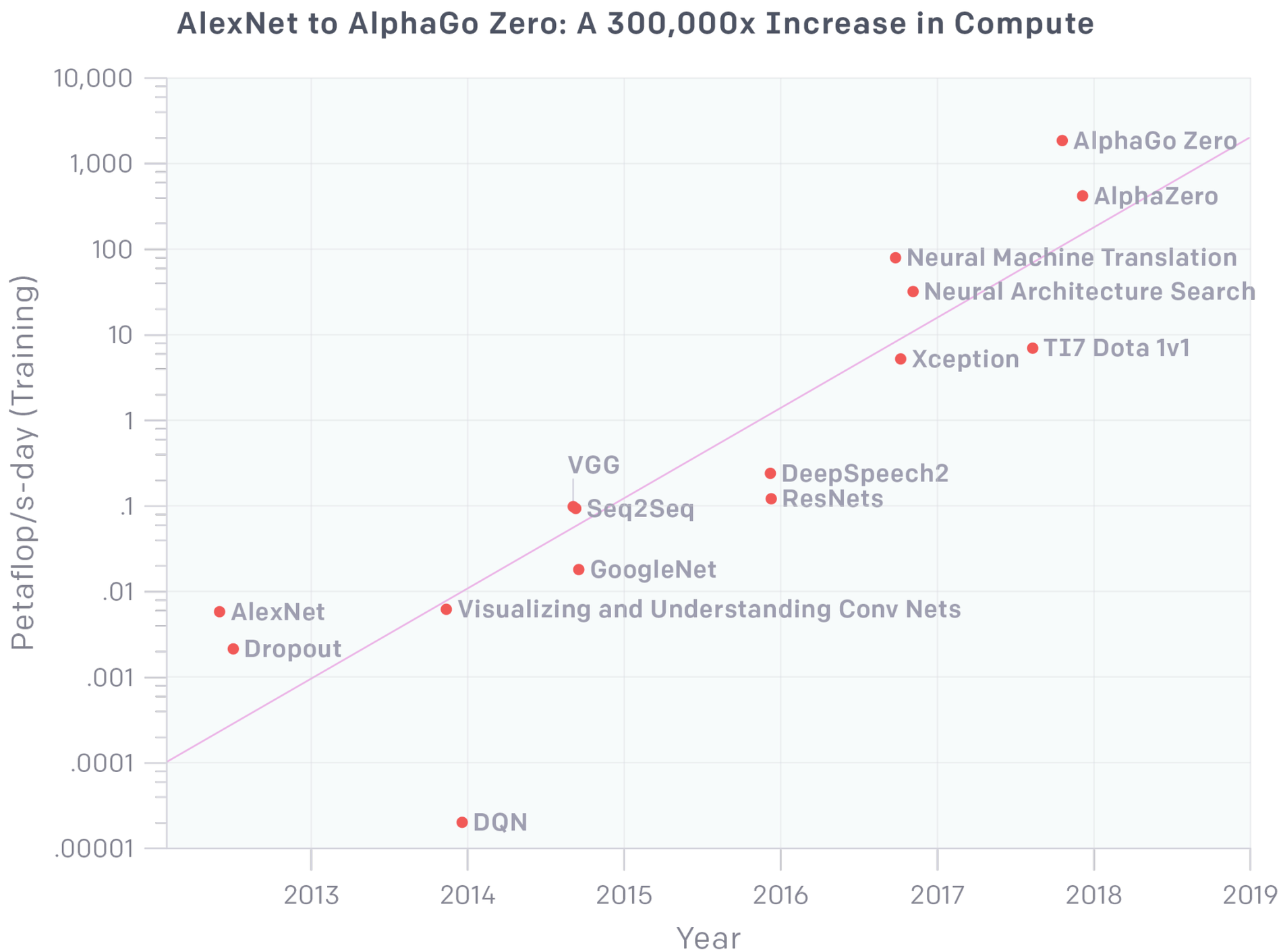
Adobe

POLITECNICO MILANO 1863

Deep Learning Advancements

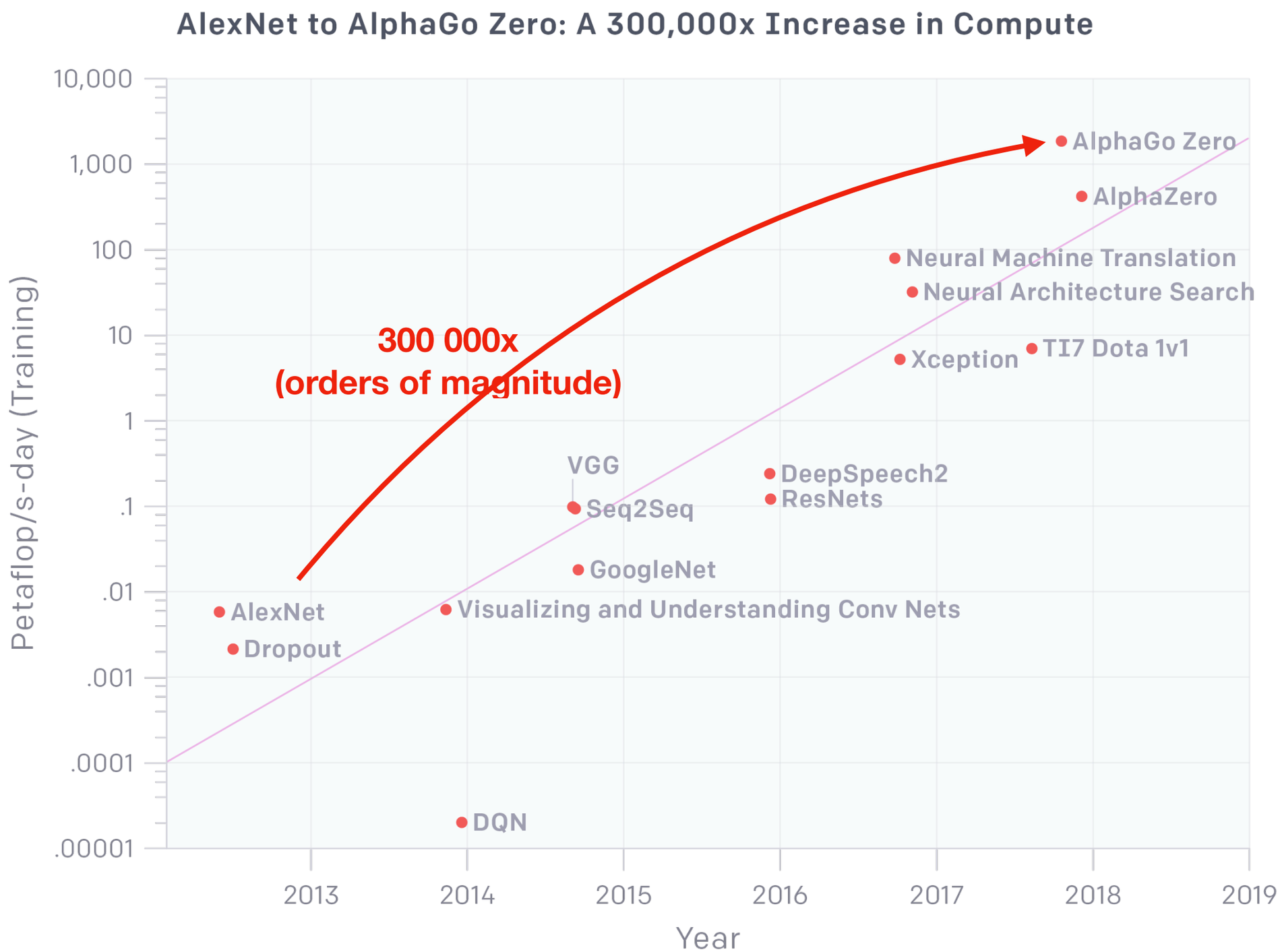


Increasing Need for Computing Power



[source: https://openai.com/blog/ai-and-compute/]

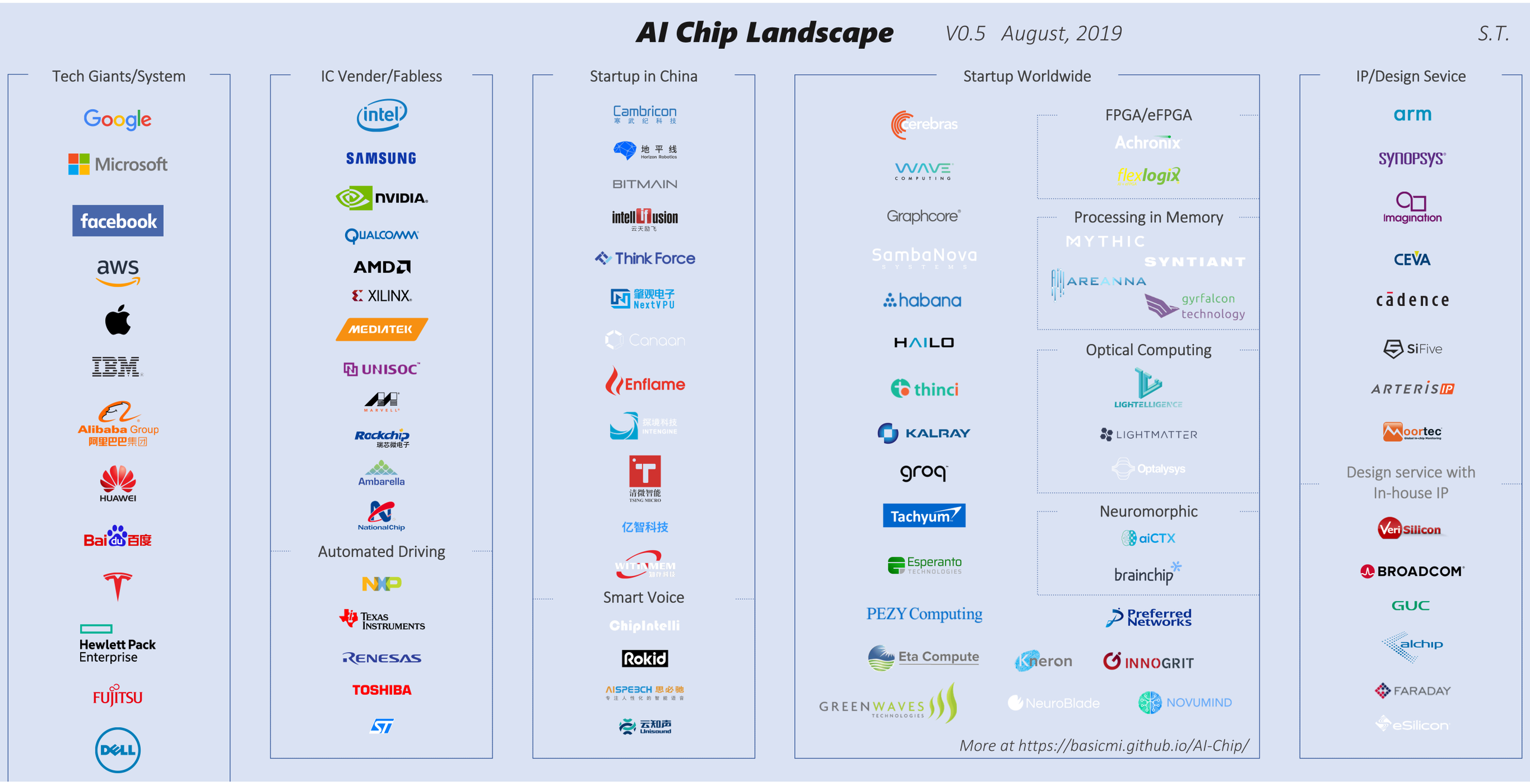
Increasing Need for Computing Power



[source: https://openai.com/blog/ai-and-compute/]

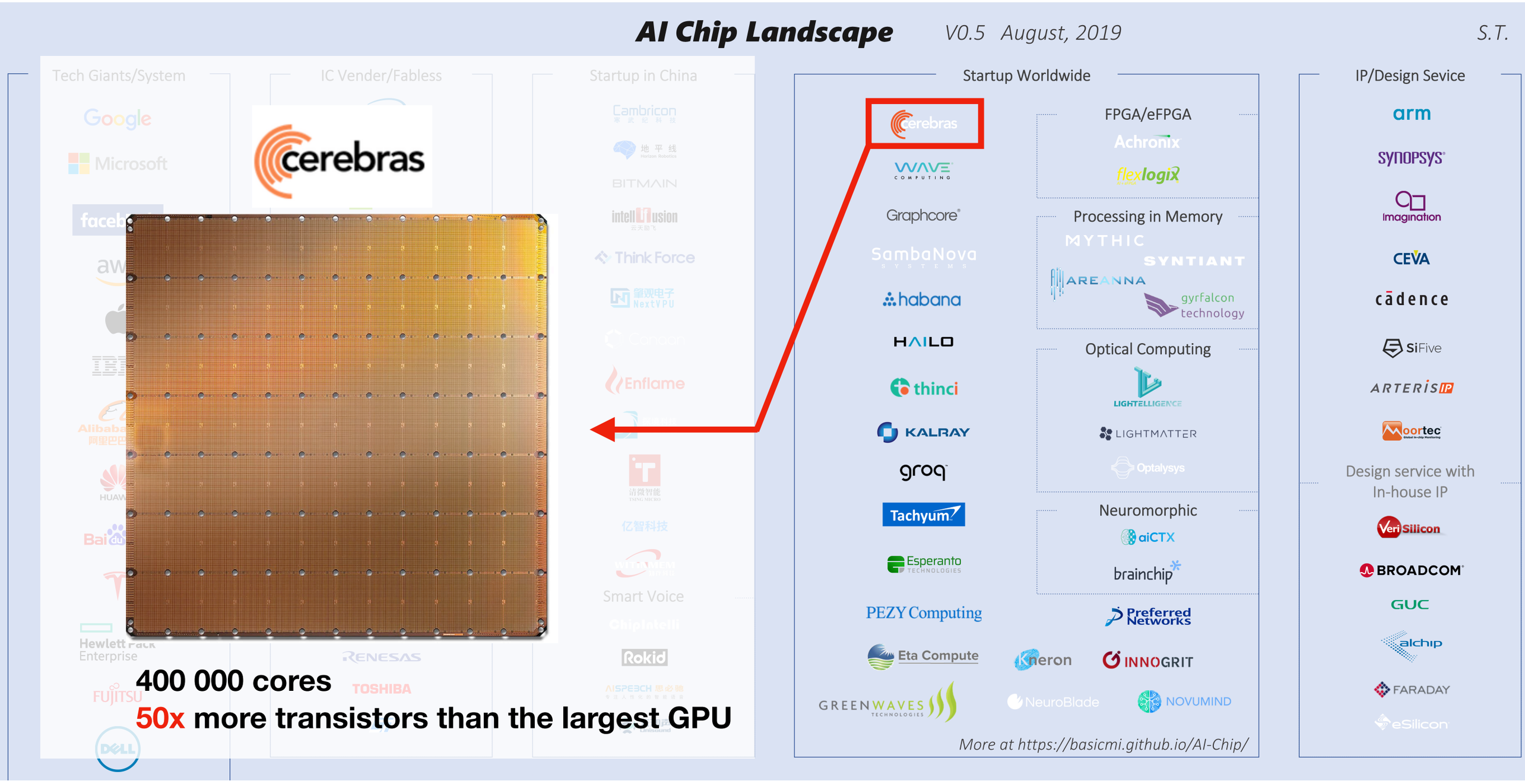
Diverse Hardware Architectures

Diverse Hardware Architectures



[source: http://github.com/basicmi/AI-Chip]

Diverse Hardware Architectures



400 000 cores

50x more transistors than the largest GPU

[source: http://github.com/basicmi/AI-Chip]

We need compilers to target these hardware architectures

State-of-the-art DNN Compilers

<u>Support</u>

- Feed forward NN
- Multiple hardware architectures

State-of-the-art DNN Compilers

<u>Support</u>

- Feed forward NN
- Multiple hardware architectures

<u>Limitations in supporting</u>

- Sparse NN
- Optimizing RNNs
- Distributed architectures

State-of-the-art DNN Compilers

<u>Support</u>

- Feed forward NN
- Multiple hardware architectures

<u>Limitations in supporting</u>

- Sparse NN
- Optimizing RNNs
- Distributed architectures

} **Tiramisu**

State-of-the-art DNN Compilers

<u>Support</u>

- Feed forward NN
- Multiple hardware architectures

<u>Limitations in supporting</u>

- Sparse NN
- Optimizing RNNs
- Distributed architectures

5x

over industrial libraries!

Tiramisu

State-of-the-art DNN Compilers

<u>Support</u>

- Feed forward NN
- Multiple hardware architectures

<u>Limitations in supporting</u>

- Sparse NN
- Optimizing RNNs
- Distributed architectures

Tiramisu

5x

over industrial libraries!

2x

over state-of-the-art compilers!

Tiramisu

Tiramisu

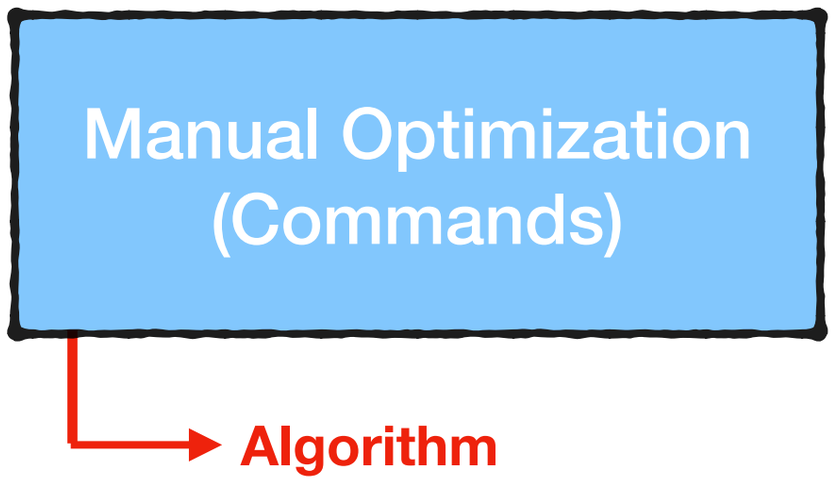
<u>Phase I:</u>

Tiramisu

<u>Phase I:</u>

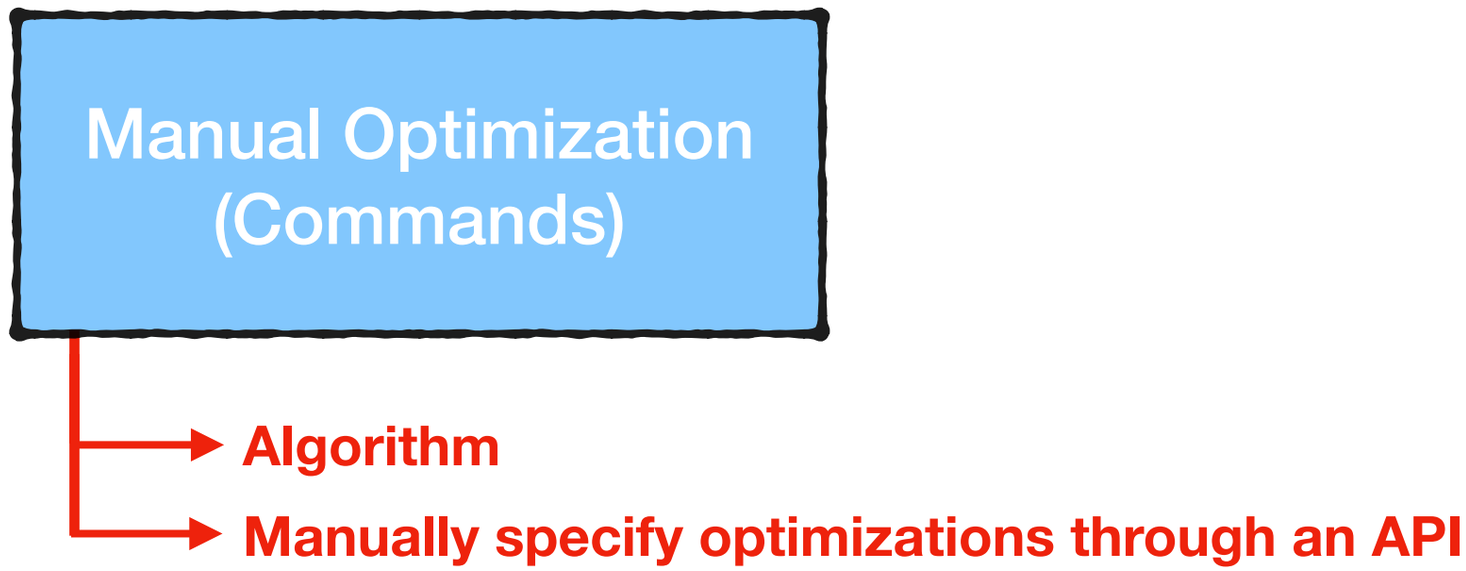
Tiramisu

Phase I:



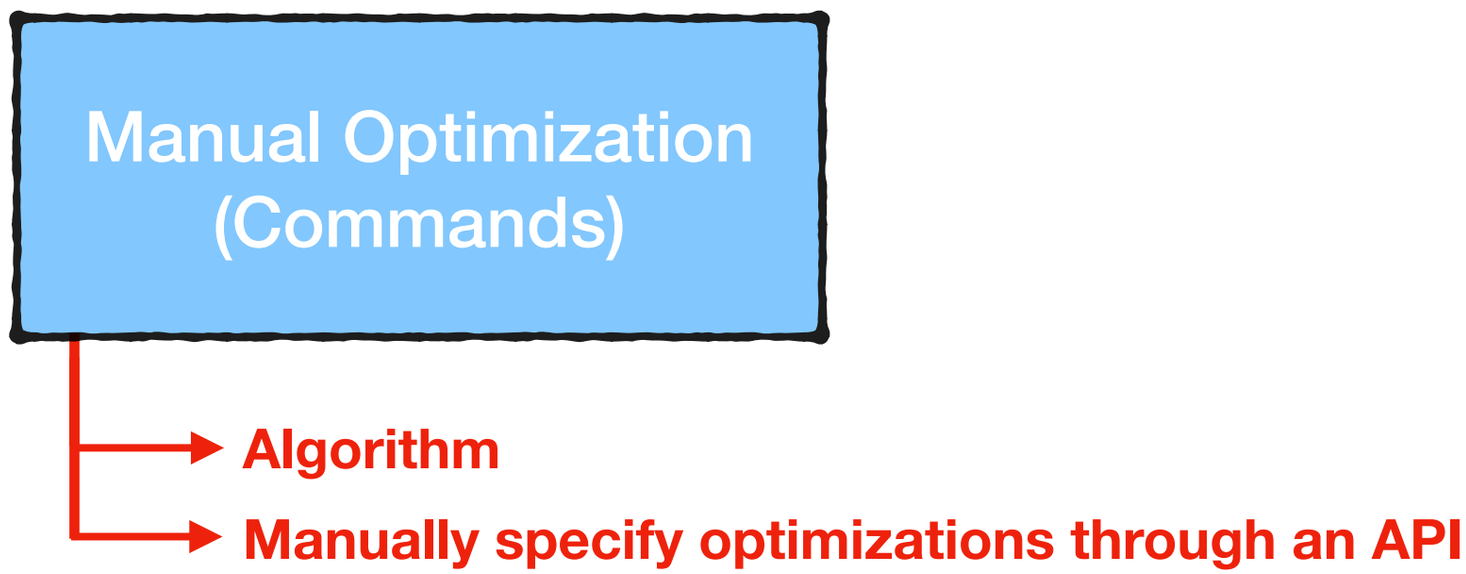
Tiramisu

Phase I:



Tiramisu

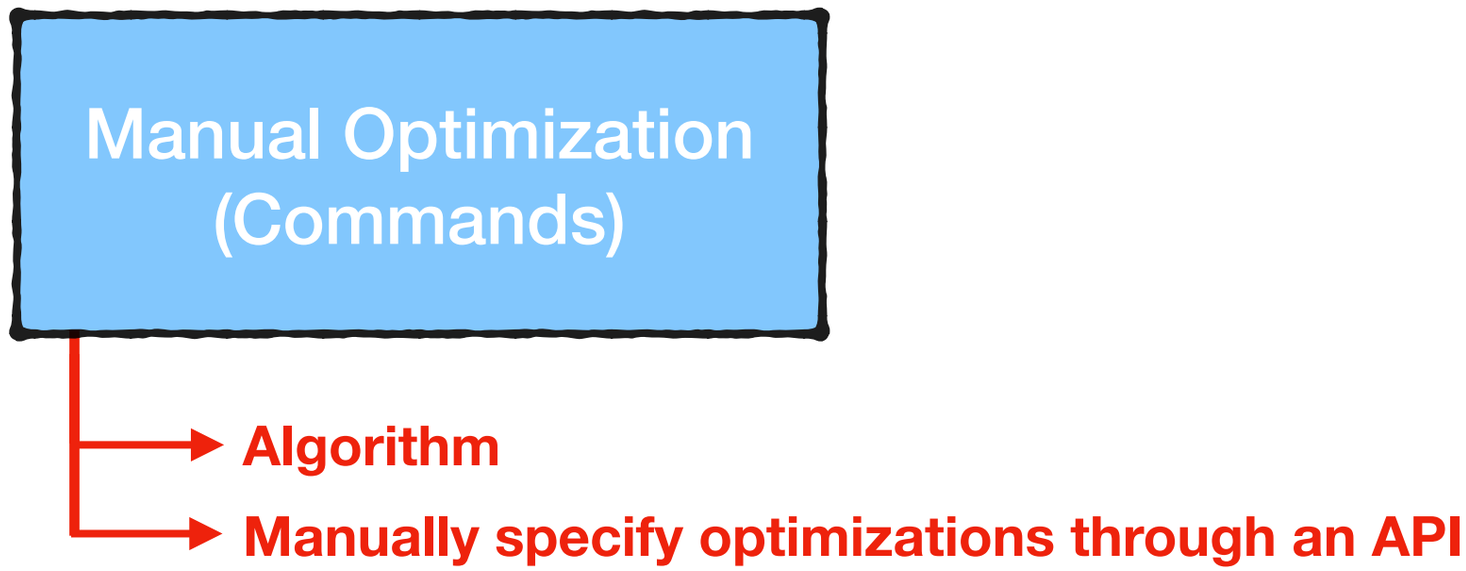
Phase I:



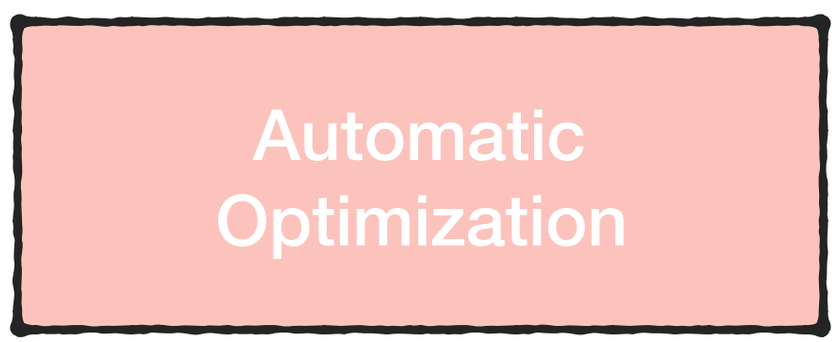
Phase II:

Tiramisu

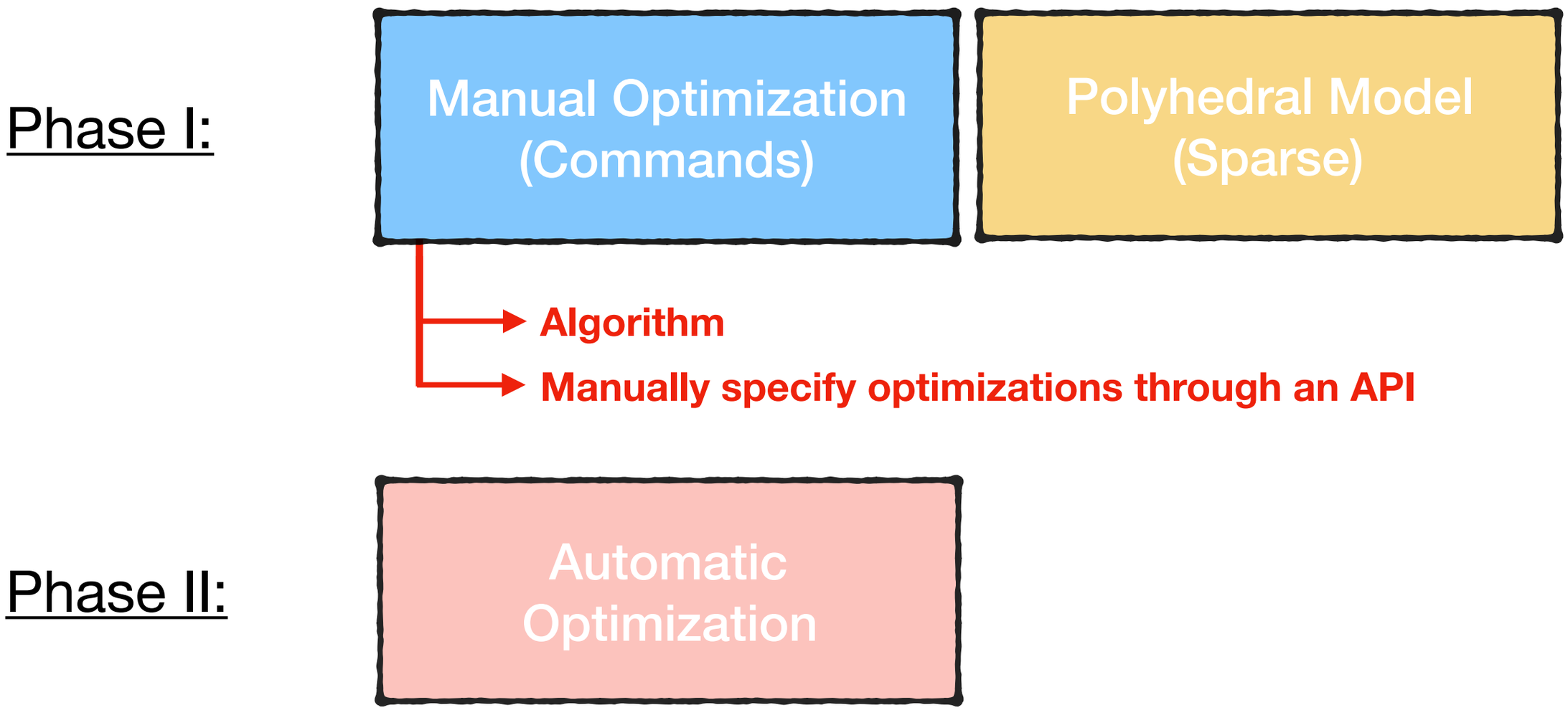
Phase I:



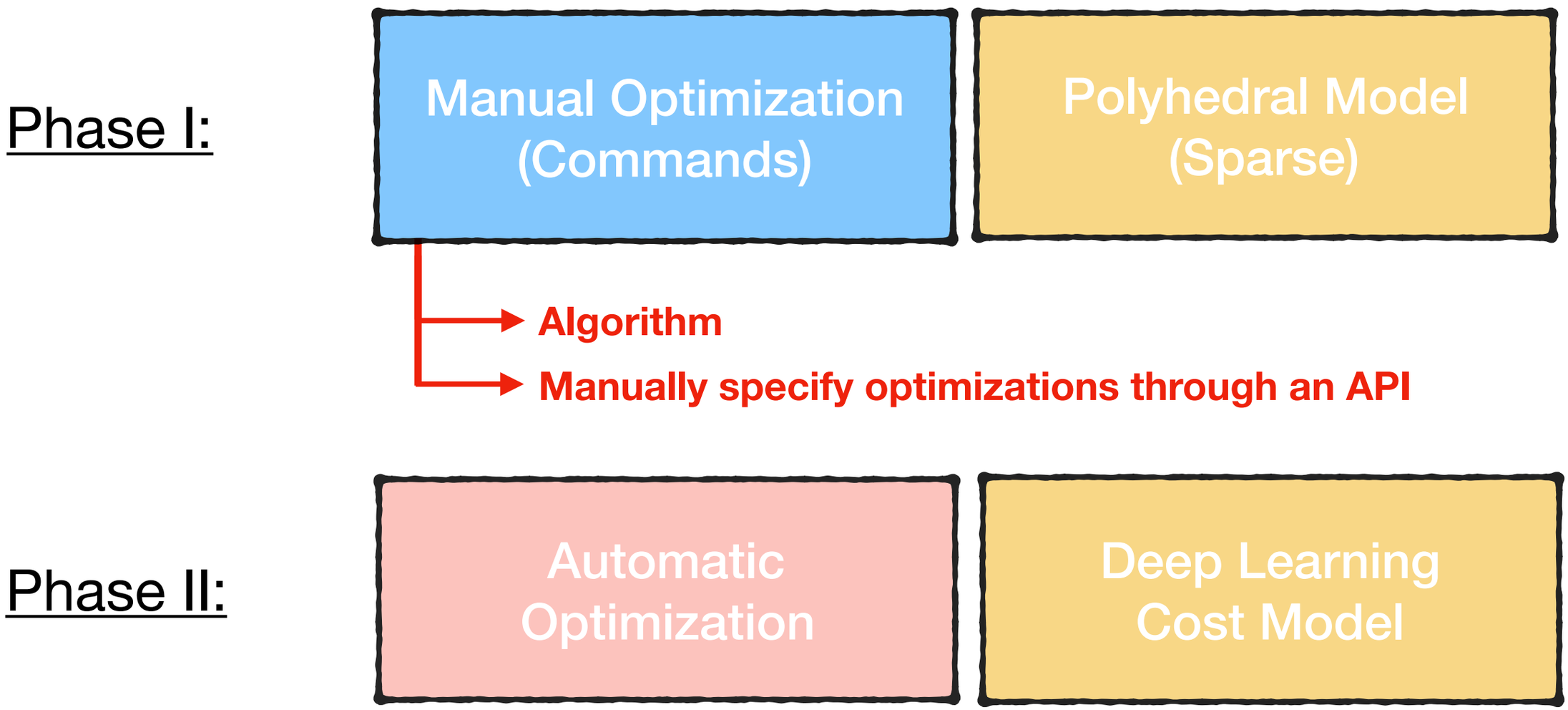
Phase II:



Tiramisu


Phase I:
Manual Optimization (Commands)
Polyhedral Model (Sparse)
Algorithm
Manually specify optimizations through an API
Phase II:
Automatic Optimization

Tiramisu


Phase I:
Manual Optimization (Commands)
Polyhedral Model (Sparse)
Algorithm
Manually specify optimizations through an API
Phase II:
Automatic Optimization
Deep Learning Cost Model

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs
- **Phase II**
  - Automatic Optimization

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Tiramisu

Tiramisu

- A polyhedral compiler

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

- Optimizations Commands

Tiramisu

- A polyhedral compiler
- **C++ API** for expressing algorithms and optimizations
- Designed for optimizing **loop nests manipulating arrays**
- Separates Algorithm from Optimization Commands
- Algorithm
  - Declare pure computations (unoptimized)

Pseudocode

```
for x in 0 … N
  for y in 0 … N
    out[x, y] = 0;
```

Tiramisu

```
var x(0, N), y(0, N);
computation out(x, y);
out(x, y) = 0;
```

- Optimizations Commands

```
out.parallelize(x);
out.vectorize(y, 4);
```

Advantages of the Polyhedral Model

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

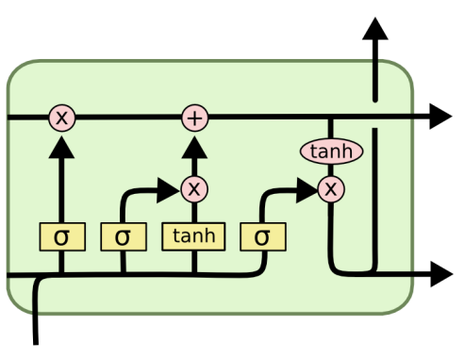
- Express algorithms with cyclic data flow graphs

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

- Express algorithms with cyclic data flow graphs



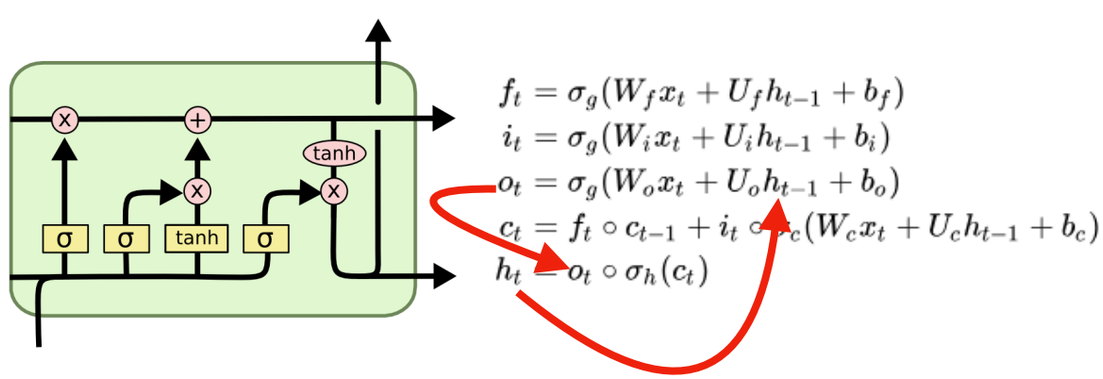
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$i_t = \sigma_g(W_i x_t + U_i h_{t-1} + b_i)$$
$$o_t = \sigma_g(W_o x_t + U_o h_{t-1} + b_o)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_c x_t + U_c h_{t-1} + b_c)$$
$$h_t = o_t \circ \sigma_h(c_t)$$

Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

- Express algorithms with cyclic data flow graphs

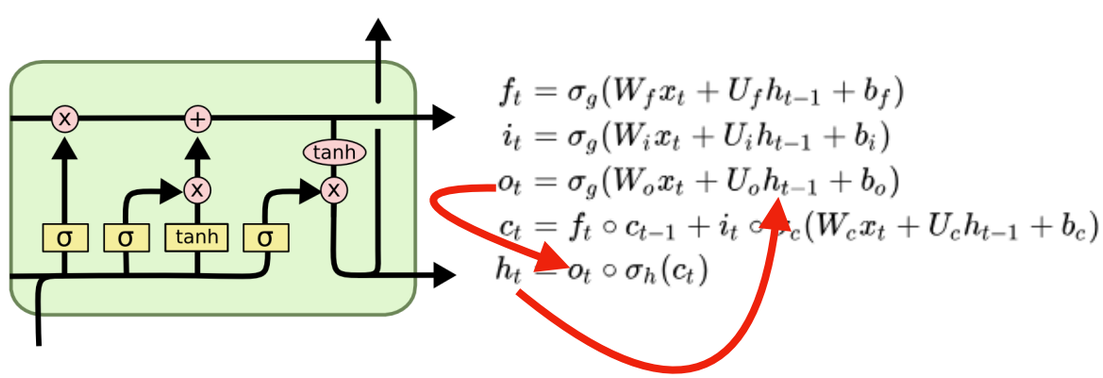


Advantages of the Polyhedral Model

- Complex transformations (all affine transformations)
- Naturally support non-rectangular loops

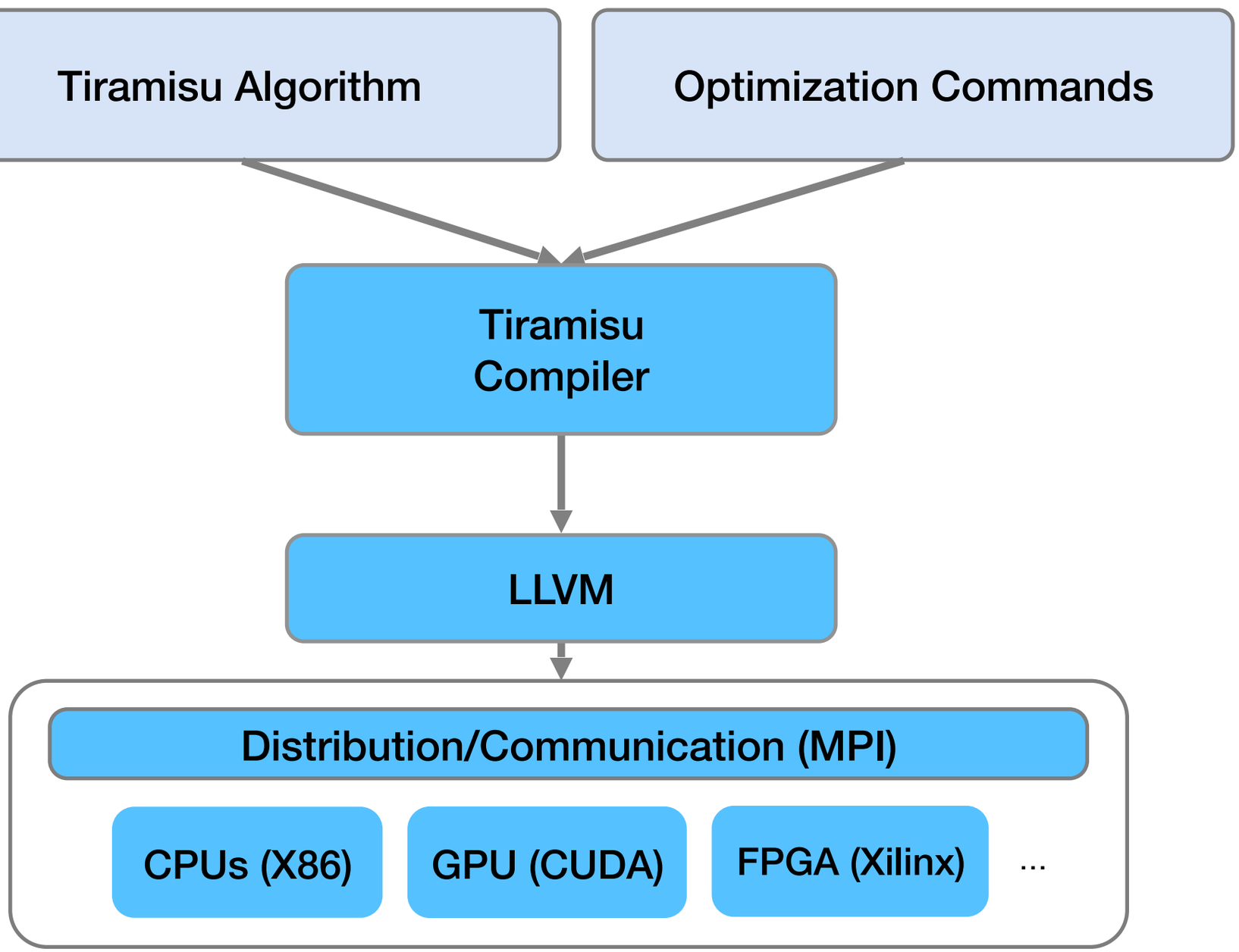
```
//Halide github issue #2373
for x in 0 … N
  for y in 0 … N
    for r in 0 … 32
      if (x >= r)
        out(x, y, r) = in(x - r, y);
```

- Express algorithms with cyclic data flow graphs

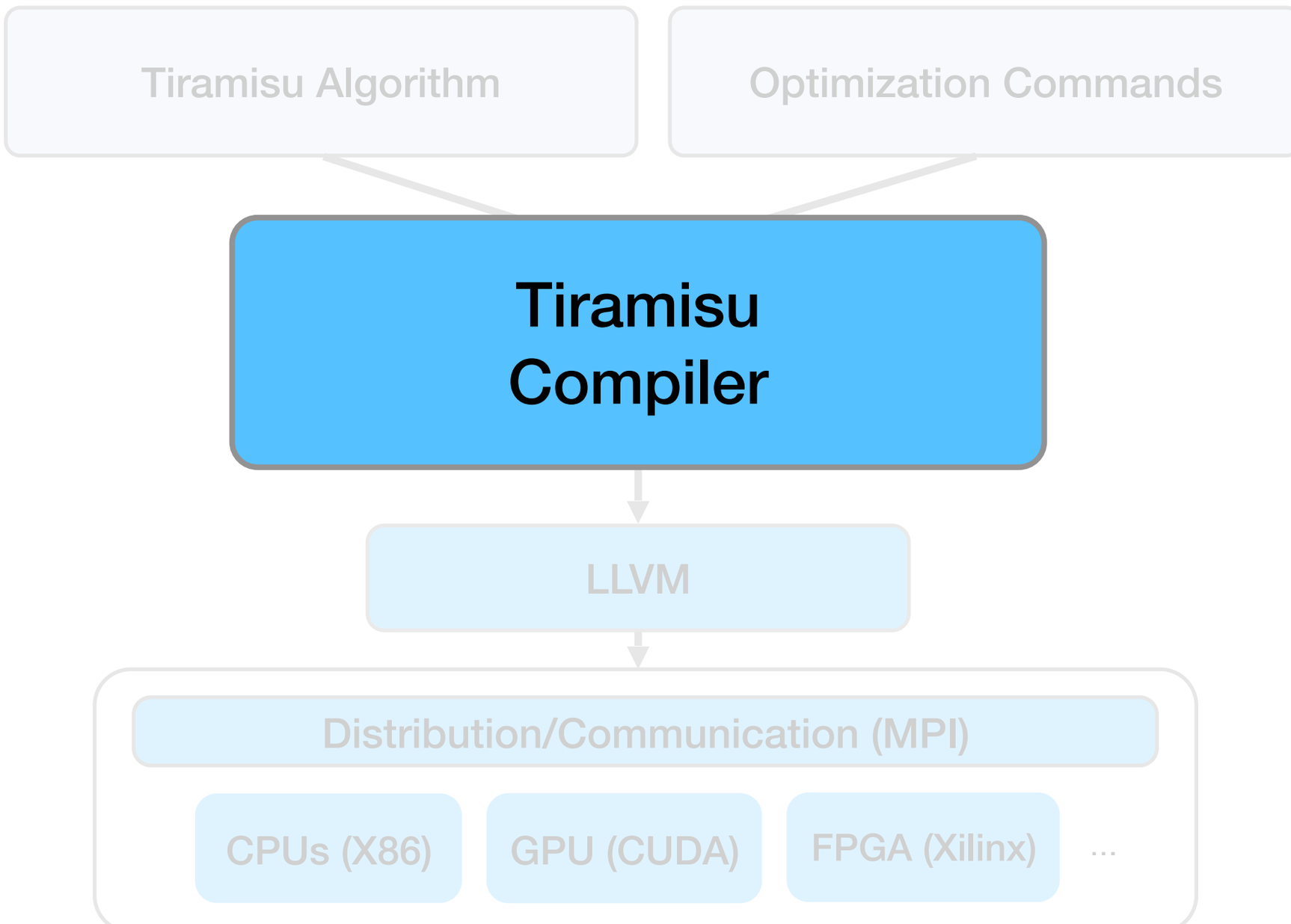


- Adequate for distributed code generation

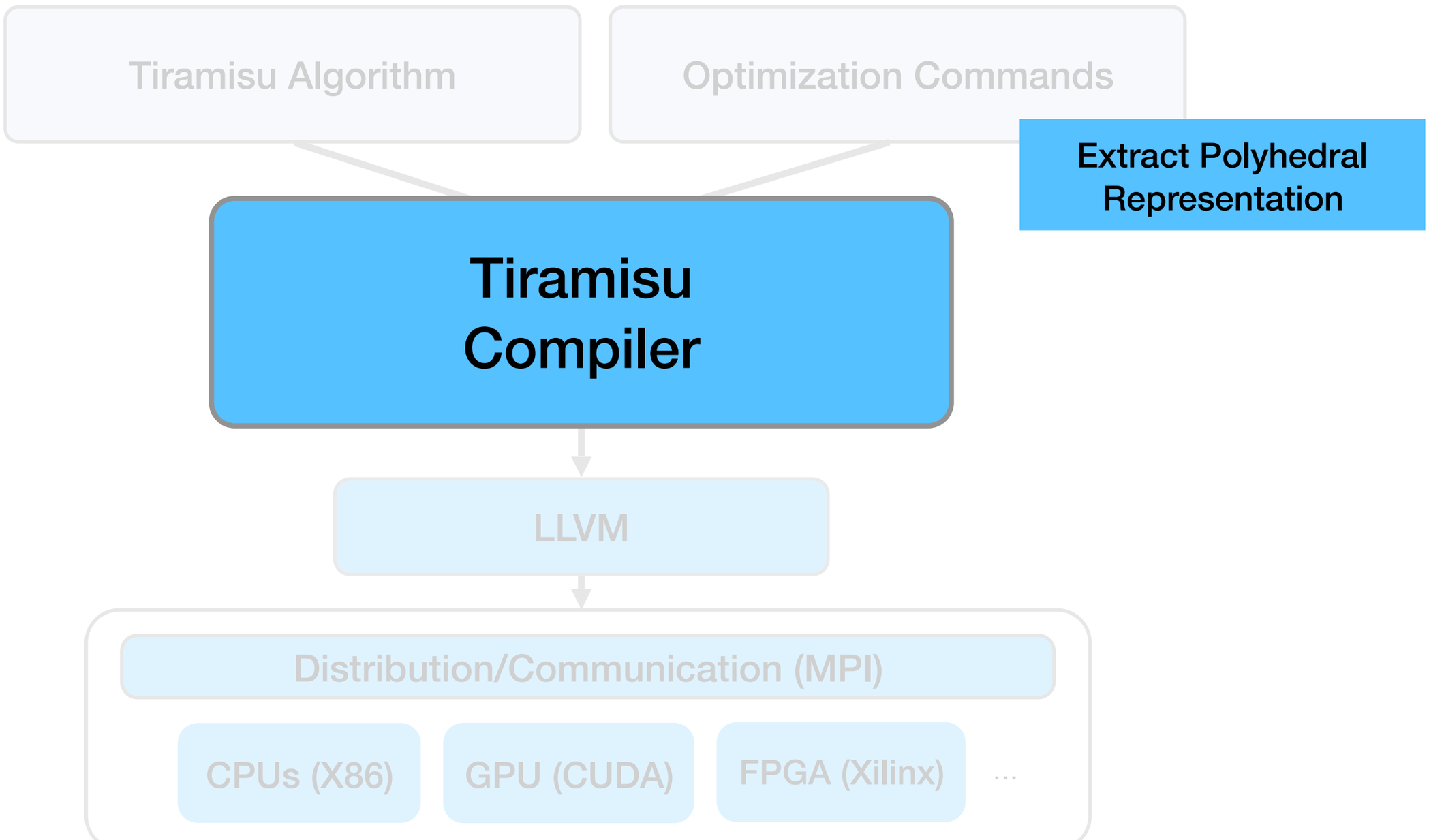
Compilation Flow



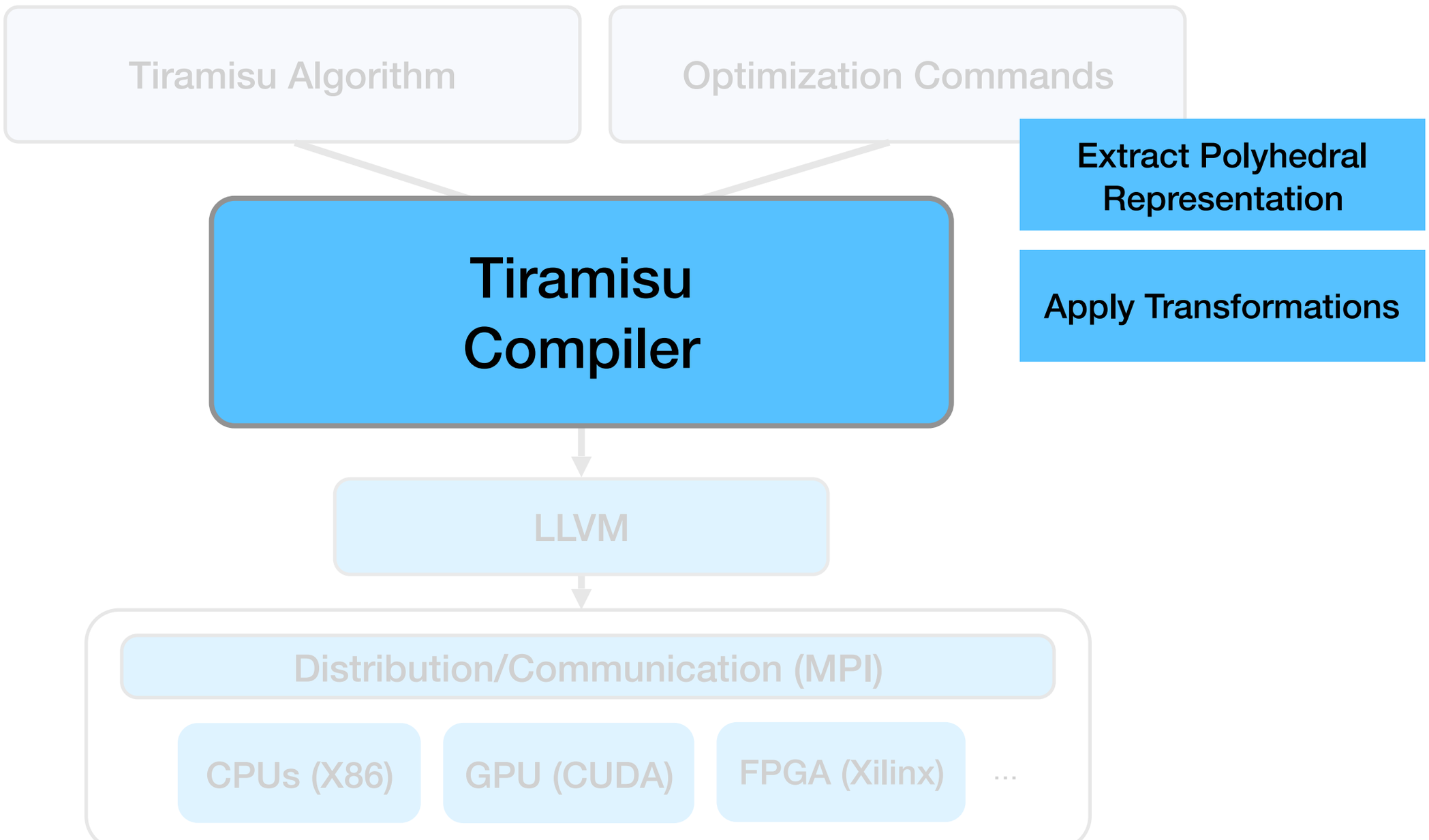
Compilation Flow



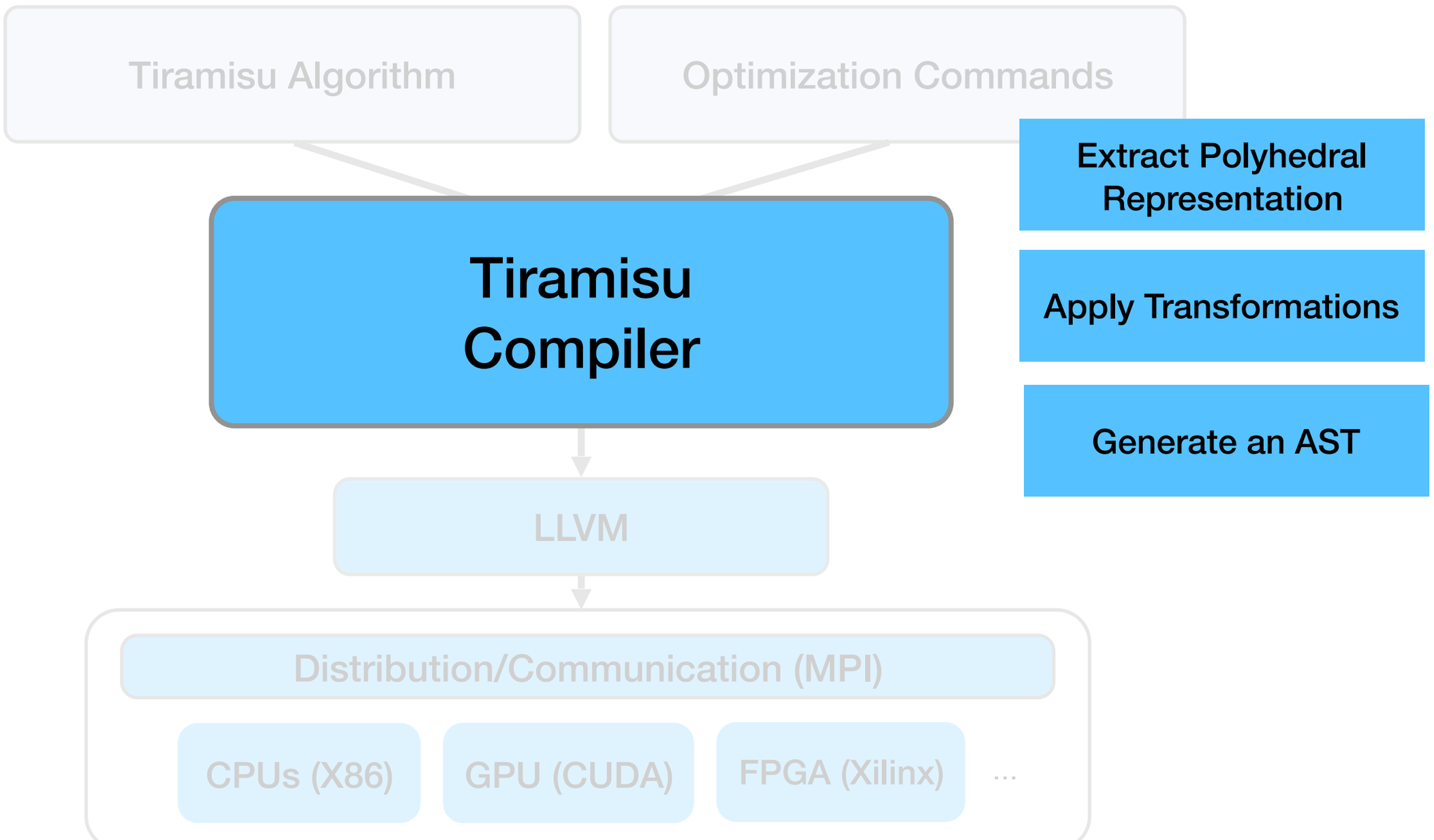
Compilation Flow



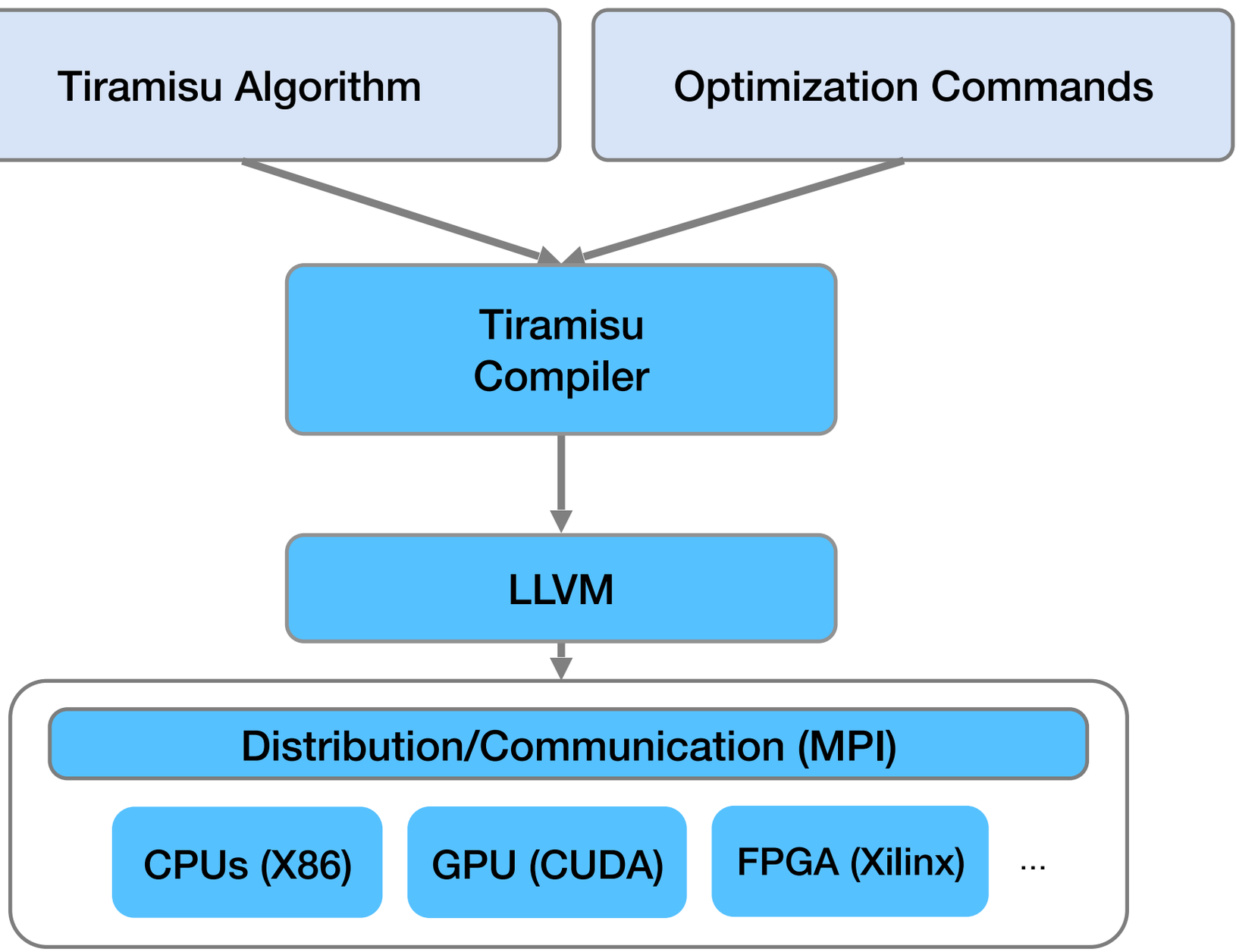
Compilation Flow



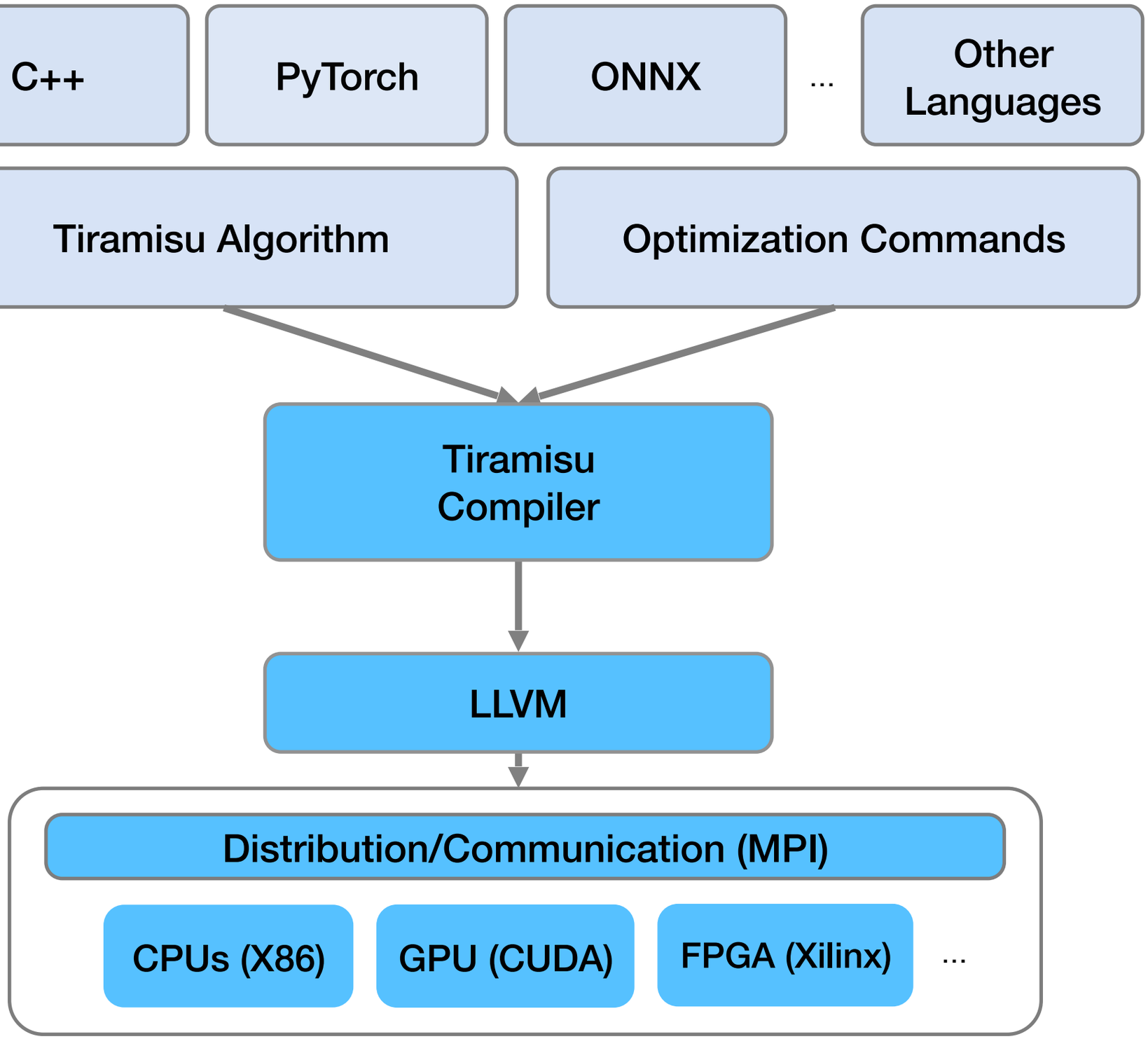
Compilation Flow



Compilation Flow



Compilation Flow



Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Scheduling

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs
- **Phase II**
  - Automatic Scheduling

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Scheduling

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);
```

```
For (int i = 0; i<N; i++)
  For (int j = 0; j<N; j++)
    C[i, j] = 0;
    For (int k = 0; k<N; k++)
      C[i, j] += A[i,k]*B[k,j];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule

C.parallel(i);
```

```
Parallel For (int i = 0; i<N; i++)
        For (int j = 0; j<N; j++)
          C[i, j] = 0;
          For (int k = 0; k<N; k++)
            C[i, j] += A[i,k]*B[k,j];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)


C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)

C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)


C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,        64,32,
             i0,j0,      i1,j1)

C.up(1).tile(i,j,k,      64,32,32,
             i0,j0,k0,   i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)


C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,        64,32,
             i0,j0,      i1,j1)

C.up(1).tile(i,j,k,        64,32,32,
             i0,j0,k0,     i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)


C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)


C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
          For (int i1 = i0; i1<min(i0+64, N); i1++)
            For (int j1 = j0; j1<min(i0+32, N); j1++)
              C[i1, j1] = 0;
          For (int k0 = 0; k0<N/32; k0++)
            For (int i1 = i0; i1<min(i0+64, N); i1++)
              For (int j1 = j0; j1<min(i0+32, N); j1++)
                For (int k1 = 0; k1<min(k0+32, N); k1++)
                  C[i1, j1] += A[i1, k1]*B[k1, j1];
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,        64,32,
             i0,j0,      i1,j1)
       .separate(i1, j1, 64, 32)

C.up(1).tile(i,j,k,        64,32,32,
             i0,j0,k0,     i1,j1,k1)
       .separate(j1, k1, 32, 32)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
         For (int i1 = i0; i1<i0+64; i1++)
          For (int j1 = j0; j1<i0+32; j1++)
           C[i1, j1] = 0;
         …
         …
         For (int k0 = 0; k0<N/32; k0++)
          For (int i1 = i0; i1<i0+64; i1++)
           For (int j1 = j0; j1<i0+32; j1++)
            For (int k1 = 0; k1<k0+32; k1++)
             C[i1, j1] += A[i1, k1]*B[k1, j1];
           …
           …
           …
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)
       .separate(i1, j1, 64, 32)
       .vectorize(j1,32)
C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
       .separate(j1, k1, 32, 32)
       .vectorize(j1,32)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
         For (int i1 = i0; i1<i0+64; i1++)
Vectorized For (int j1 = j0; j1<i0+32; j1++)
           C[i1, j1] = 0;
         …
         …
         For (int k0 = 0; k0<N/32; k0++)
          For (int i1 = i0; i1<i0+64; i1++)
Vectorized For (int j1 = j0; j1<i0+32; j1++)
            For (int k1 = 0; k1<k0+32; k1++)
             C[i1, j1] += A[i1, k1]*B[k1, j1];
           …
           …
           …
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)
       .separate(i1, j1, 64, 32)
       .vectorize(j1,32).unroll(i1)
C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
       .separate(j1, k1, 32, 32)
       .vectorize(j1,32).unroll(k1)
```

```
Parallel For (int i0 = 0; i0<N/64; i+=64)
        For (int j0 = 0; j0<N/32; j0+=32)
    Unroll For (int i1 = i0; i1<i0+64; i1++)
Vectorized For (int j1 = j0; j1<i0+32; j1++)
            C[i1, j1] = 0;
        …
        …
        For (int k0 = 0; k0<N/32; k0++)
         For (int i1 = i0; i1<i0+64; i1++)
Vectorized For (int j1 = j0; j1<i0+32; j1++)
      Unroll For (int k1 = 0; k1<k0+32; k1++)
              C[i1, j1] += A[i1, k1]*B[k1, j1];
        …
        …
        …
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32,
             i0,j0,     i1,j1)
       .separate(i1, j1, 64, 32)
       .vectorize(j1,32).unroll(i1);
C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
       .separate(j1, k1, 32, 32)
       .vectorize(j1,32).unroll(k1);

B.pack(64,32,32);
```

Matrix Multiplication

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;
C.parallel(i);
C.up(0).tile(i,j,       64,32
             i0,j0,     i1,j1)
       .separate(i1, j1, 64, 32)
       .vectorize(j1,32).unroll(i1);
C.up(1).tile(i,j,k,       64,32,32,
             i0,j0,k0,    i1,j1,k1)
       .separate(j1, k1, 32, 32)
       .vectorize(j1,32).unroll(k1);

B.pack(64,32,32);
```

Speedup: 350x (4 Cores)

```
let buf_A = _halide_buffer_get_host(buf_A.buffer)
let buf_B = _halide_buffer_get_host(buf_B.buffer)
let buf_C = _halide_buffer_get_host(buf_C.buffer)
let buf_SIZES = _halide_buffer_get_host(buf_SIZES.buffer)
let buf_alpha = _halide_buffer_get_host(buf_alpha.buffer)
let buf_beta = _halide_buffer_get_host(buf_beta.buffer)
produce  {
  let N = buf_SIZES[0]
  let M = buf_SIZES[1]
  let K = buf_SIZES[2]
  let a = buf_alpha[0]
  let b = buf_beta[0]
  allocate buf_temp[float32 * M * N]
  allocate buf_Bp[float32 * (M % 32) * K * (M/32)]
  let c1.old_max.s = int32(floor_f32(float32(((M + -32)/128))))
  let c1.new_max.s = min(((M + 96)/128), (c1.old_max.s + 1))
  parallel (c1, 0, max(c1.new_max.s, 0)) {
    let c3.old_max.s = int32(floor_f32(float32(((K + -32)/128))))
    let c3.new_max.s = min(((K + 96)/128), (c3.old_max.s + 1))
    for (c3, 0, max(c3.new_max.s, 0)) {
      for (c5, 0, min(((M/32) - (c1*4)), 4)) {
        for (c7, 0, min(((K/32) - (c3*4)), 4)) {
          for (c9, 0, 32) {
            for (c11, 0, 32) {
              buf_Bp[((((((c3*4) + c7)*32) + c11)*(M % 32)) + (((c1*4) + c5)*((M % 32)*K)))] = buf_B[(((((c3*4) +
c7)*32) + c11) + (((((c1*4) + c5)*32) + c9)*M))]
            }
          }
        }
      }
    }
  }
  free buf_Bp
  let c1.old_max$1 = int32(floor_f32(float32(((N + 64)/128))))
  let c1.new_max$1.s = min(((N + 64)/128), c1.old_max$1)
  parallel (c1, 0, max(c1.new_max$1.s, 0)) {
    let t53 = int32(floor_f32(float32(((M + -32)/128))))
    let t54 = min(((M + 96)/128), (t53 + 1))
    for (c3, 0, max(t54, 0)) {
      for (c5, 0, min(((N/64) - (c1*2)), 2)) {
        for (c7, 0, min(((M/32) - (c3*4)), 4)) {
          buf_temp[ramp(((((c3*4) + c7)*32) + ((((c1*2) + c5)*64)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 1)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 2)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 3)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 4)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 5)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 6)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 7)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 8)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 9)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 10)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 11)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 12)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 13)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 14)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 15)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 16)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 17)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 18)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 19)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 20)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 21)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 22)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 23)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 24)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 25)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 26)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 27)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 28)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 29)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 30)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 31)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 32)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 33)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 34)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 35)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 36)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 37)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 38)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 39)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 40)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 41)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 42)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 43)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 44)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 45)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 46)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 47)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 48)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 49)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 50)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 51)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 52)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 53)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 54)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 55)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 56)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 57)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 58)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 59)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 60)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 61)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 62)*M)), 1, 32)] = x32(0.000000f)
          buf_temp[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 63)*M)), 1, 32)] = x32(0.000000f)
        }
      }
    }
    for (c3, 0, max(t54, 0)) {
      let c5.old_max.s = int32(floor_f32(float32(((K + -32)/128))))
      let c5.new_max.s = min(((K + 96)/128), (c5.old_max.s + 1))
      for (c5, 0, max(c5.new_max.s, 0)) {
        for (c7, 0, min(((N/64) - (c1*2)), 2)) {
          for (c9, 0, min(((M/32) - (c3*4)), 4)) {
            for (c11, 0, min(((K/32) - (c5*4)), 4)) {
              for (c13, 0, 64) {
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K))])*buf_B[ramp(((((c3*4) + c9)*32) + ((((c5*4) + c11)*32)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 1)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 1)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 2)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 2)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 3)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 3)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 4)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 4)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 5)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 5)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 6)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 6)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 7)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 7)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 8)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 8)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 9)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 9)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 10)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 10)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 11)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 11)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 12)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 12)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 13)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 13)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 14)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 14)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 15)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 15)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 16)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 16)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 17)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 17)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 18)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 18)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 19)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 19)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 20)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 20)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 21)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 21)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 22)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 22)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 23)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 23)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 24)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 24)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 25)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 25)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 26)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 26)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 27)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 27)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 28)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 28)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 29)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 29)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 30)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 30)*M)), 1, 32)]))
                buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)] = (buf_temp[ramp(((((c3*4) + c9)*32) + (((((c1*2) + c7)*64) + c13)*M)), 1, 32)]
+ (x32(buf_A[(((((c5*4) + c11)*32) + (((((c1*2) + c7)*64) + c13)*K)) + 31)])*buf_B[ramp(((((c3*4) + c9)*32) + (((((c5*4) + c11)*32) + 31)*M)), 1, 32)]))
              }
            }
          }
        }
      }
    }
    for (c3, 0, max(t54, 0)) {
      for (c5, 0, min(((N/64) - (c1*2)), 2)) {
        for (c7, 0, min(((M/32) - (c3*4)), 4)) {
          buf_C[ramp(((((c3*4) + c7)*32) + ((((c1*2) + c5)*64)*M)), 1, 32)] = (let t152 = ((((c3*4) + c7)*32) + ((((c1*2) + c5)*64)*M)) in
((x32(a)*buf_temp[ramp(t152, 1, 32)]) + (x32(b)*buf_C[ramp(t152, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 1)*M)), 1, 32)] = (let t153 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 1)*M)) in
((x32(a)*buf_temp[ramp(t153, 1, 32)]) + (x32(b)*buf_C[ramp(t153, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 2)*M)), 1, 32)] = (let t154 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 2)*M)) in
((x32(a)*buf_temp[ramp(t154, 1, 32)]) + (x32(b)*buf_C[ramp(t154, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 3)*M)), 1, 32)] = (let t155 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 3)*M)) in
((x32(a)*buf_temp[ramp(t155, 1, 32)]) + (x32(b)*buf_C[ramp(t155, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 4)*M)), 1, 32)] = (let t156 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 4)*M)) in
((x32(a)*buf_temp[ramp(t156, 1, 32)]) + (x32(b)*buf_C[ramp(t156, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 5)*M)), 1, 32)] = (let t157 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 5)*M)) in
((x32(a)*buf_temp[ramp(t157, 1, 32)]) + (x32(b)*buf_C[ramp(t157, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 6)*M)), 1, 32)] = (let t158 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 6)*M)) in
((x32(a)*buf_temp[ramp(t158, 1, 32)]) + (x32(b)*buf_C[ramp(t158, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 7)*M)), 1, 32)] = (let t159 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 7)*M)) in
((x32(a)*buf_temp[ramp(t159, 1, 32)]) + (x32(b)*buf_C[ramp(t159, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 8)*M)), 1, 32)] = (let t160 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 8)*M)) in
((x32(a)*buf_temp[ramp(t160, 1, 32)]) + (x32(b)*buf_C[ramp(t160, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 9)*M)), 1, 32)] = (let t161 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 9)*M)) in
((x32(a)*buf_temp[ramp(t161, 1, 32)]) + (x32(b)*buf_C[ramp(t161, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 10)*M)), 1, 32)] = (let t162 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 10)*M)) in
((x32(a)*buf_temp[ramp(t162, 1, 32)]) + (x32(b)*buf_C[ramp(t162, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 11)*M)), 1, 32)] = (let t163 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 11)*M)) in
((x32(a)*buf_temp[ramp(t163, 1, 32)]) + (x32(b)*buf_C[ramp(t163, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 12)*M)), 1, 32)] = (let t164 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 12)*M)) in
((x32(a)*buf_temp[ramp(t164, 1, 32)]) + (x32(b)*buf_C[ramp(t164, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 13)*M)), 1, 32)] = (let t165 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 13)*M)) in
((x32(a)*buf_temp[ramp(t165, 1, 32)]) + (x32(b)*buf_C[ramp(t165, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 14)*M)), 1, 32)] = (let t166 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 14)*M)) in
((x32(a)*buf_temp[ramp(t166, 1, 32)]) + (x32(b)*buf_C[ramp(t166, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 15)*M)), 1, 32)] = (let t167 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 15)*M)) in
((x32(a)*buf_temp[ramp(t167, 1, 32)]) + (x32(b)*buf_C[ramp(t167, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 16)*M)), 1, 32)] = (let t168 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 16)*M)) in
((x32(a)*buf_temp[ramp(t168, 1, 32)]) + (x32(b)*buf_C[ramp(t168, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 17)*M)), 1, 32)] = (let t169 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 17)*M)) in
((x32(a)*buf_temp[ramp(t169, 1, 32)]) + (x32(b)*buf_C[ramp(t169, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 18)*M)), 1, 32)] = (let t170 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 18)*M)) in
((x32(a)*buf_temp[ramp(t170, 1, 32)]) + (x32(b)*buf_C[ramp(t170, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 19)*M)), 1, 32)] = (let t171 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 19)*M)) in
((x32(a)*buf_temp[ramp(t171, 1, 32)]) + (x32(b)*buf_C[ramp(t171, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 20)*M)), 1, 32)] = (let t172 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 20)*M)) in
((x32(a)*buf_temp[ramp(t172, 1, 32)]) + (x32(b)*buf_C[ramp(t172, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 21)*M)), 1, 32)] = (let t173 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 21)*M)) in
((x32(a)*buf_temp[ramp(t173, 1, 32)]) + (x32(b)*buf_C[ramp(t173, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 22)*M)), 1, 32)] = (let t174 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 22)*M)) in
((x32(a)*buf_temp[ramp(t174, 1, 32)]) + (x32(b)*buf_C[ramp(t174, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 23)*M)), 1, 32)] = (let t175 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 23)*M)) in
((x32(a)*buf_temp[ramp(t175, 1, 32)]) + (x32(b)*buf_C[ramp(t175, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 24)*M)), 1, 32)] = (let t176 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 24)*M)) in
((x32(a)*buf_temp[ramp(t176, 1, 32)]) + (x32(b)*buf_C[ramp(t176, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 25)*M)), 1, 32)] = (let t177 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 25)*M)) in
((x32(a)*buf_temp[ramp(t177, 1, 32)]) + (x32(b)*buf_C[ramp(t177, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 26)*M)), 1, 32)] = (let t178 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 26)*M)) in
((x32(a)*buf_temp[ramp(t178, 1, 32)]) + (x32(b)*buf_C[ramp(t178, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 27)*M)), 1, 32)] = (let t179 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 27)*M)) in
((x32(a)*buf_temp[ramp(t179, 1, 32)]) + (x32(b)*buf_C[ramp(t179, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 28)*M)), 1, 32)] = (let t180 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 28)*M)) in
((x32(a)*buf_temp[ramp(t180, 1, 32)]) + (x32(b)*buf_C[ramp(t180, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 29)*M)), 1, 32)] = (let t181 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 29)*M)) in
((x32(a)*buf_temp[ramp(t181, 1, 32)]) + (x32(b)*buf_C[ramp(t181, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 30)*M)), 1, 32)] = (let t182 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 30)*M)) in
((x32(a)*buf_temp[ramp(t182, 1, 32)]) + (x32(b)*buf_C[ramp(t182, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 31)*M)), 1, 32)] = (let t183 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 31)*M)) in
((x32(a)*buf_temp[ramp(t183, 1, 32)]) + (x32(b)*buf_C[ramp(t183, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 32)*M)), 1, 32)] = (let t184 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 32)*M)) in
((x32(a)*buf_temp[ramp(t184, 1, 32)]) + (x32(b)*buf_C[ramp(t184, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 33)*M)), 1, 32)] = (let t185 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 33)*M)) in
((x32(a)*buf_temp[ramp(t185, 1, 32)]) + (x32(b)*buf_C[ramp(t185, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 34)*M)), 1, 32)] = (let t186 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 34)*M)) in
((x32(a)*buf_temp[ramp(t186, 1, 32)]) + (x32(b)*buf_C[ramp(t186, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 35)*M)), 1, 32)] = (let t187 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 35)*M)) in
((x32(a)*buf_temp[ramp(t187, 1, 32)]) + (x32(b)*buf_C[ramp(t187, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 36)*M)), 1, 32)] = (let t188 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 36)*M)) in
((x32(a)*buf_temp[ramp(t188, 1, 32)]) + (x32(b)*buf_C[ramp(t188, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 37)*M)), 1, 32)] = (let t189 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 37)*M)) in
((x32(a)*buf_temp[ramp(t189, 1, 32)]) + (x32(b)*buf_C[ramp(t189, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 38)*M)), 1, 32)] = (let t190 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 38)*M)) in
((x32(a)*buf_temp[ramp(t190, 1, 32)]) + (x32(b)*buf_C[ramp(t190, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 39)*M)), 1, 32)] = (let t191 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 39)*M)) in
((x32(a)*buf_temp[ramp(t191, 1, 32)]) + (x32(b)*buf_C[ramp(t191, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 40)*M)), 1, 32)] = (let t192 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 40)*M)) in
((x32(a)*buf_temp[ramp(t192, 1, 32)]) + (x32(b)*buf_C[ramp(t192, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 41)*M)), 1, 32)] = (let t193 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 41)*M)) in
((x32(a)*buf_temp[ramp(t193, 1, 32)]) + (x32(b)*buf_C[ramp(t193, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 42)*M)), 1, 32)] = (let t194 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 42)*M)) in
((x32(a)*buf_temp[ramp(t194, 1, 32)]) + (x32(b)*buf_C[ramp(t194, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 43)*M)), 1, 32)] = (let t195 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 43)*M)) in
((x32(a)*buf_temp[ramp(t195, 1, 32)]) + (x32(b)*buf_C[ramp(t195, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 44)*M)), 1, 32)] = (let t196 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 44)*M)) in
((x32(a)*buf_temp[ramp(t196, 1, 32)]) + (x32(b)*buf_C[ramp(t196, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 45)*M)), 1, 32)] = (let t197 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 45)*M)) in
((x32(a)*buf_temp[ramp(t197, 1, 32)]) + (x32(b)*buf_C[ramp(t197, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 46)*M)), 1, 32)] = (let t198 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 46)*M)) in
((x32(a)*buf_temp[ramp(t198, 1, 32)]) + (x32(b)*buf_C[ramp(t198, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 47)*M)), 1, 32)] = (let t199 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 47)*M)) in
((x32(a)*buf_temp[ramp(t199, 1, 32)]) + (x32(b)*buf_C[ramp(t199, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 48)*M)), 1, 32)] = (let t200 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 48)*M)) in
((x32(a)*buf_temp[ramp(t200, 1, 32)]) + (x32(b)*buf_C[ramp(t200, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 49)*M)), 1, 32)] = (let t201 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 49)*M)) in
((x32(a)*buf_temp[ramp(t201, 1, 32)]) + (x32(b)*buf_C[ramp(t201, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 50)*M)), 1, 32)] = (let t202 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 50)*M)) in
((x32(a)*buf_temp[ramp(t202, 1, 32)]) + (x32(b)*buf_C[ramp(t202, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 51)*M)), 1, 32)] = (let t203 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 51)*M)) in
((x32(a)*buf_temp[ramp(t203, 1, 32)]) + (x32(b)*buf_C[ramp(t203, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 52)*M)), 1, 32)] = (let t204 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 52)*M)) in
((x32(a)*buf_temp[ramp(t204, 1, 32)]) + (x32(b)*buf_C[ramp(t204, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 53)*M)), 1, 32)] = (let t205 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 53)*M)) in
((x32(a)*buf_temp[ramp(t205, 1, 32)]) + (x32(b)*buf_C[ramp(t205, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 54)*M)), 1, 32)] = (let t206 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 54)*M)) in
((x32(a)*buf_temp[ramp(t206, 1, 32)]) + (x32(b)*buf_C[ramp(t206, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 55)*M)), 1, 32)] = (let t207 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 55)*M)) in
((x32(a)*buf_temp[ramp(t207, 1, 32)]) + (x32(b)*buf_C[ramp(t207, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 56)*M)), 1, 32)] = (let t208 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 56)*M)) in
((x32(a)*buf_temp[ramp(t208, 1, 32)]) + (x32(b)*buf_C[ramp(t208, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 57)*M)), 1, 32)] = (let t209 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 57)*M)) in
((x32(a)*buf_temp[ramp(t209, 1, 32)]) + (x32(b)*buf_C[ramp(t209, 1, 32)])))
          buf_C[ramp(((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 58)*M)), 1, 32)] = (let t210 = ((((c3*4) + c7)*32) + (((((c1*2) + c5)*64) + 58)*M)) in
```

Matrix Multiplication (GPU)

```
// Algorithm
var i(0, N), j(0, N), k(0, N);
input A(i,j), B(i,j);
computation C(i,j);

C(i,j) = 0;
C(i,j) += A(i, k) * B(k, j);

// Schedule
var i0, j0, k0, i1, j1, k1;

C.gpu_tile(i,j,k, 64,32,32, i0,j0,k0, i1,j1,k1).separate();

A.cache_shared_at(C.update(1), j0);
A.cache_local_at(C.update(1), j0);
B.cache_shared_at(C.update(1), j0);
B.cache_local_at(C.update(1), j0);
```

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
    };
};
```

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
    };
};
```

Host/device data copies

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
    };
};
```

Host/device data copies

global —> shared

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
    };
};
```

Host/device data copies

global —> shared

shared —> local

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
     };
};
```

Host/device data copies

Compute shared/local memory sizes

global —> shared

shared —> local

Matrix Multiplication (CUDA)

```
// Host code

float *b_A_glb, *b_B_glb, *b_C_glb;
extern "C" void matmul(float *b_A, float *b_B, float *b_C)
{
  cudaMalloc(&b_A_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_B_glb, 3072 * 3072 * sizeof(float));
  cudaMalloc(&b_C_glb, 3072 * 3072 * sizeof(float));
  cudaMemcpy(b_A_glb, b_A, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_B_glb, b_B, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_C_glb, b_C, 3072 * 3072 * sizeof(float), cudaMemcpyHostToDevice);
  {
    dim3 blocks((15 + 1), (11 + 1), 1);
    dim3 threads((15 + 1), (15 + 1), 1);
    _kernel_0<<<blocks, threads>>>(b_A_glb, b_B_glb, b_C_glb);
   };
   cudaMemcpy(b_C, b_C_glb, 3072 * 3072 * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(b_A_glb);
   cudaFree(b_B_glb);
   cudaFree(b_C_glb);
}

// Device code
static __global__ void _kernel_0(float *b_A_glb, float *b_B_glb, float *b_C_glb)
{
   const int32_t __bx__ = (blockIdx.x + 0);
   const int32_t __by__ = (blockIdx.y + 0);
   const int32_t __tx__ = (threadIdx.x + 0);
   const int32_t __ty__ = (threadIdx.y + 0);
   __shared__ float b_A_shr[16 * 193];
   __shared__ float b_B_shr[16 * 256];
   float b_A_reg;
   float b_B_reg[16];
   float b_acc[12 * 16];
   for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] = 0;
      };
   };

   for (int32_t c9 = 0; (c9 <= 191); (c9 += 1))
   {
      for (int32_t c11 = 0; (c11 <= 11); (c11 += 1))
      {
         b_A_shr[((0 + (((12 * __tx__) + c11) * 1)) + (__ty__ * (1 * ((int32_t) 193))))] =
            b_A_glb[((0 + ((__ty__ + (16 * c9)) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c11) * (1 * ((int32_t) 3072))))];
      }
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         b_B_shr[((0 + ((__tx__ + (16 * __ty__)) * 1)) + (c11 * (1 * ((int32_t) 256))))] =
            b_B_glb[((0 + ((((256 * __by__) + __tx__) + (16 * __ty__)) * 1)) + (((16 * c9) + c11) * (1 * ((int32_t) 3072))))];
      }
      __syncthreads();
      for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
      {
         for (int32_t c13 = 0; (c13 <= 15); (c13 += 1))
         {
            b_B_reg[(0 + (c13 * 1))] =
               b_B_shr[((0 + (((16 * __ty__) + c13) * 1)) + (c11 * (1 * ((int32_t) 256))))];
         };
         for (int32_t c13 = 0; (c13 <= 11); (c13 += 1))
         {
            b_A_reg = b_A_shr[((0 + (((12 * __tx__) + c13) * 1)) + (c11 * (1 * ((int32_t) 193))))];
            for (int32_t c15 = 0; (c15 <= 15); (c15 += 1))
            {
              b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] =
                 (b_acc[((0 + (c15 * 1)) + (c13 * (1 * ((int32_t) 16))))] + (b_A_reg * b_B_reg[(0 + (c15 * 1))]));
            };
         };
       };
       __syncthreads();
    };
    for (int32_t c9 = 0; (c9 <= 11); (c9 += 1))
    {
       for (int32_t c11 = 0; (c11 <= 15); (c11 += 1))
       {
          b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] =
            ((b_acc[((0 + (c11 * 1)) + (c9 * (1 * ((int32_t) 16))))] * 3) +
             (b_C_glb[((0 + ((((256 * __by__) + (16 * __ty__)) + c11) * 1)) + ((((192 * __bx__) + (12 * __tx__)) + c9) * (1 * ((int32_t) 3072))))] * 2));
       };
    };
};
```

Host/device data copies

Compute shared/local memory sizes

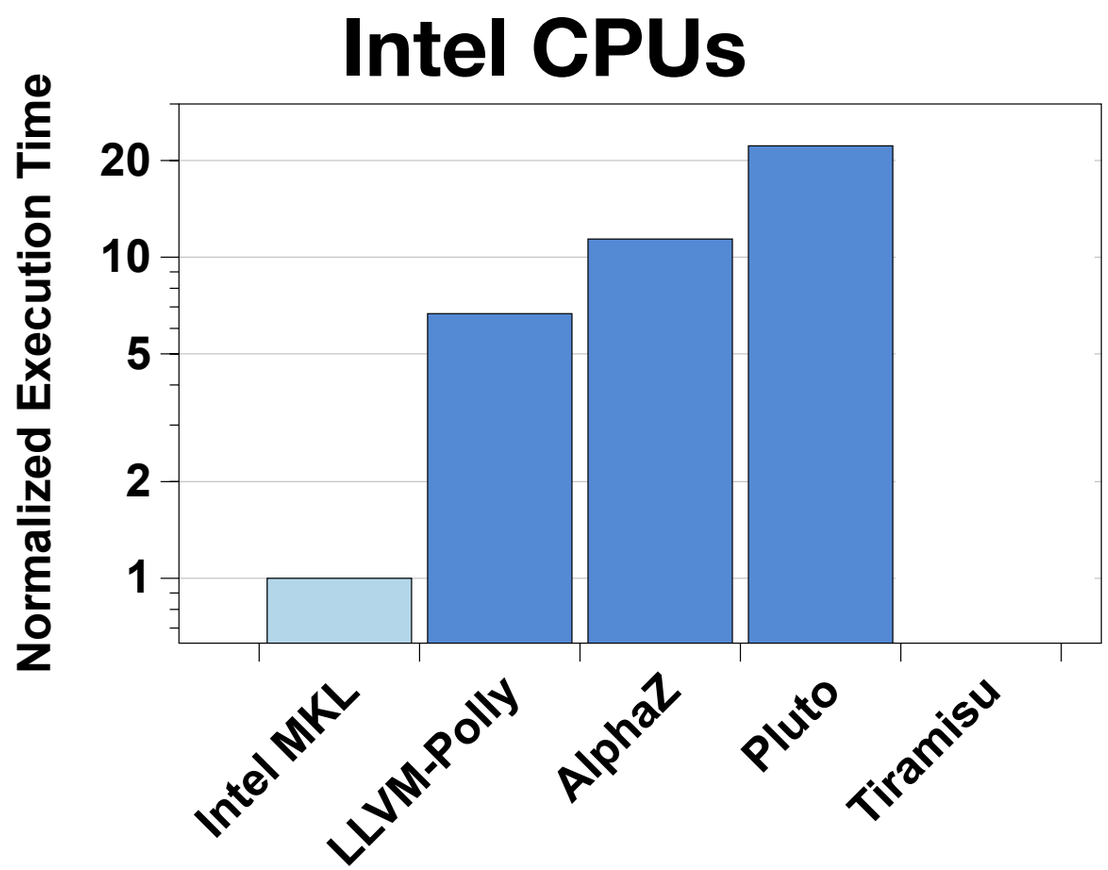
global —> shared

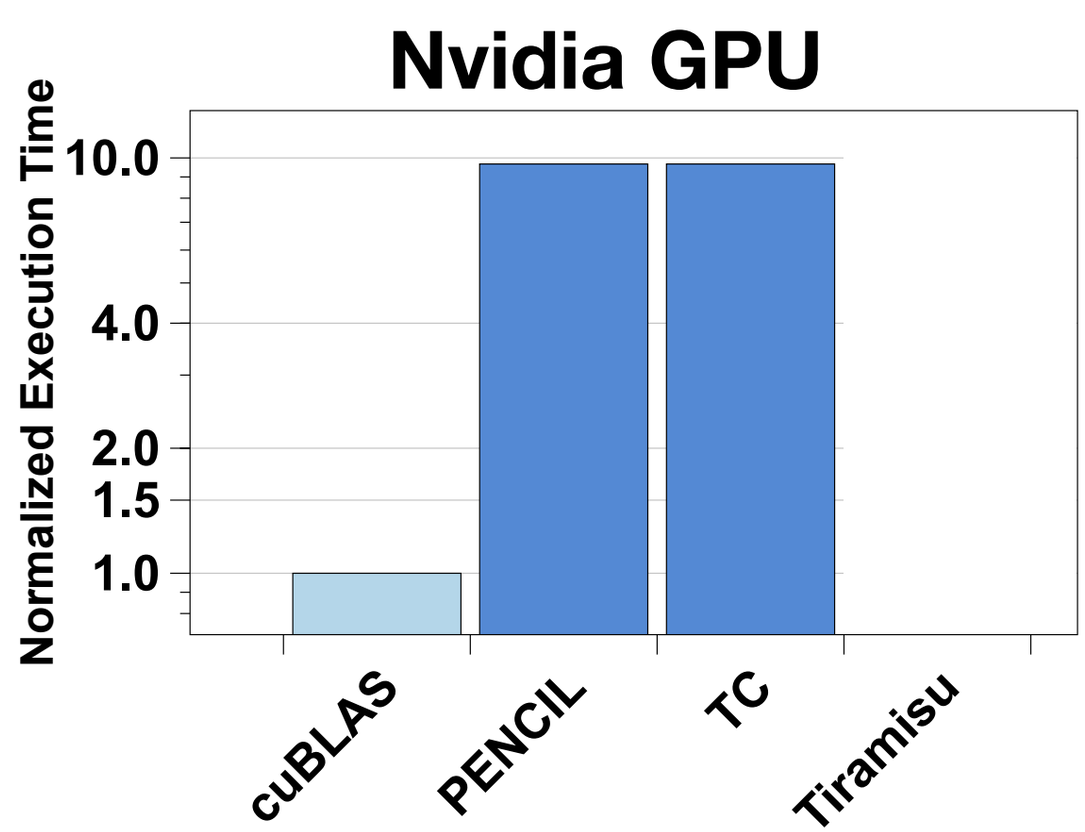
shared —> local

Synchronization

GEMM (Comparison with Polyhedral Compilers)

GEMM (Comparison with Polyhedral Compilers)

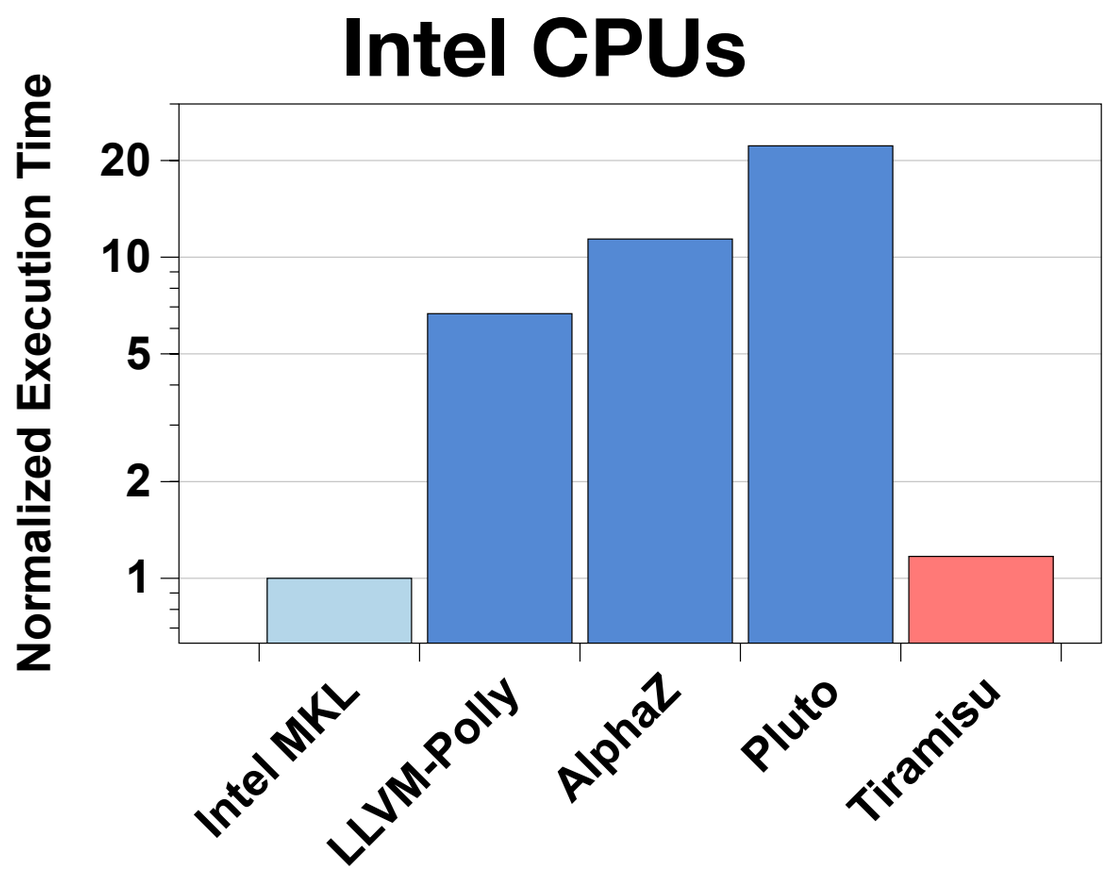


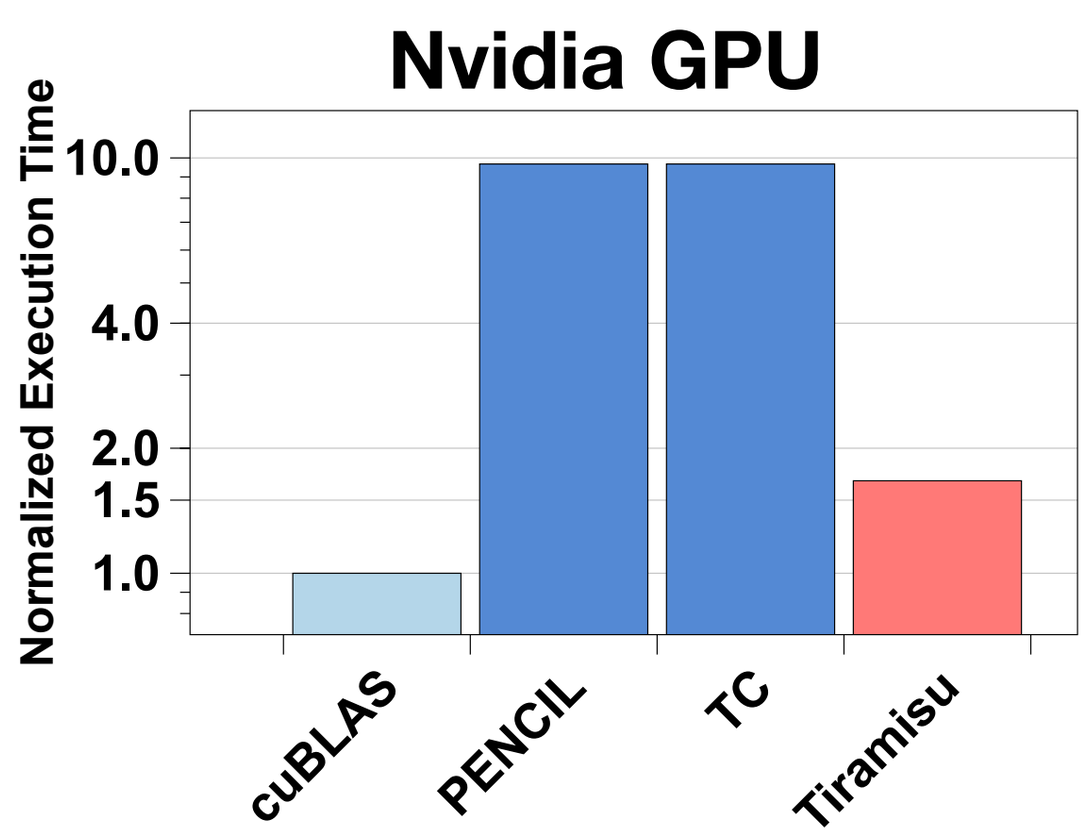


[Matrix size: 1060x1060]

[4-Cores CPU, Nvidia Pascal P4]

GEMM (Comparison with Polyhedral Compilers)





[Matrix size: 1060x1060]

[4-Cores CPU, Nvidia Pascal P4]

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs
- **Phase II**
  - Automatic Optimization

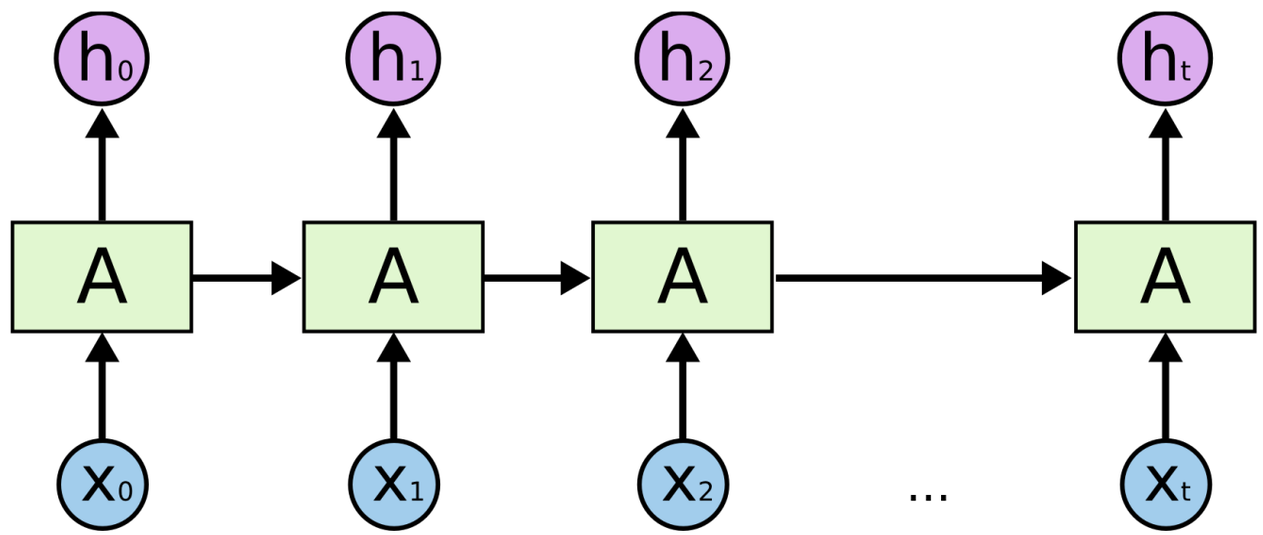
Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

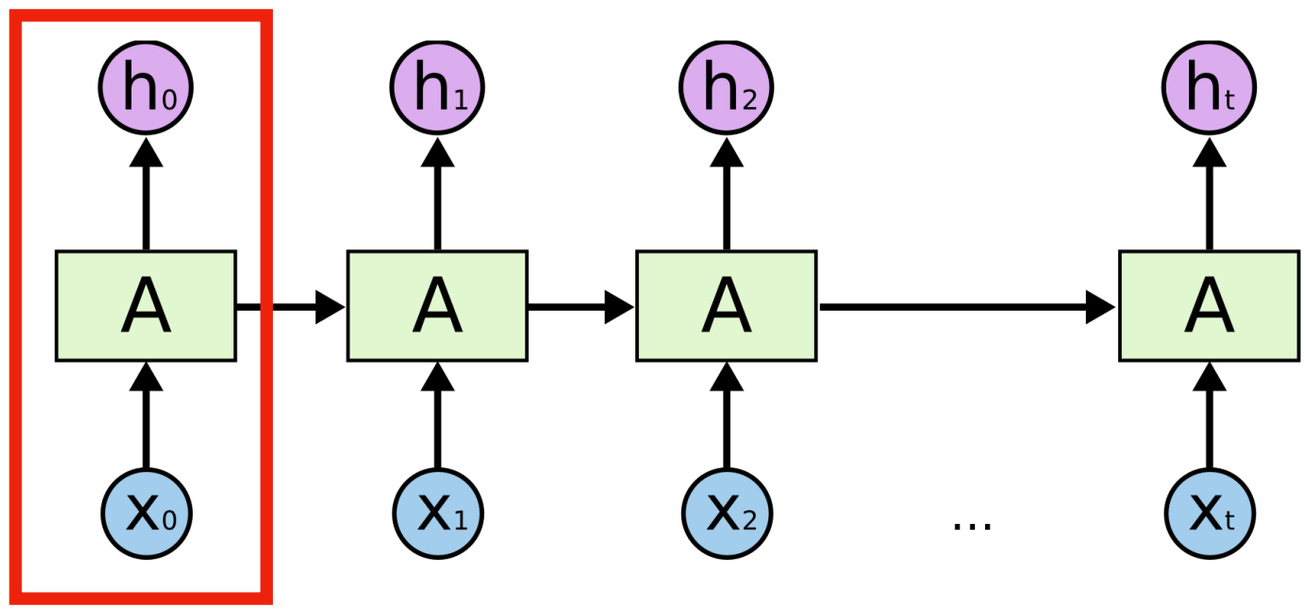
- **Phase II**
  - Automatic Optimization

Recurrent Neural Networks

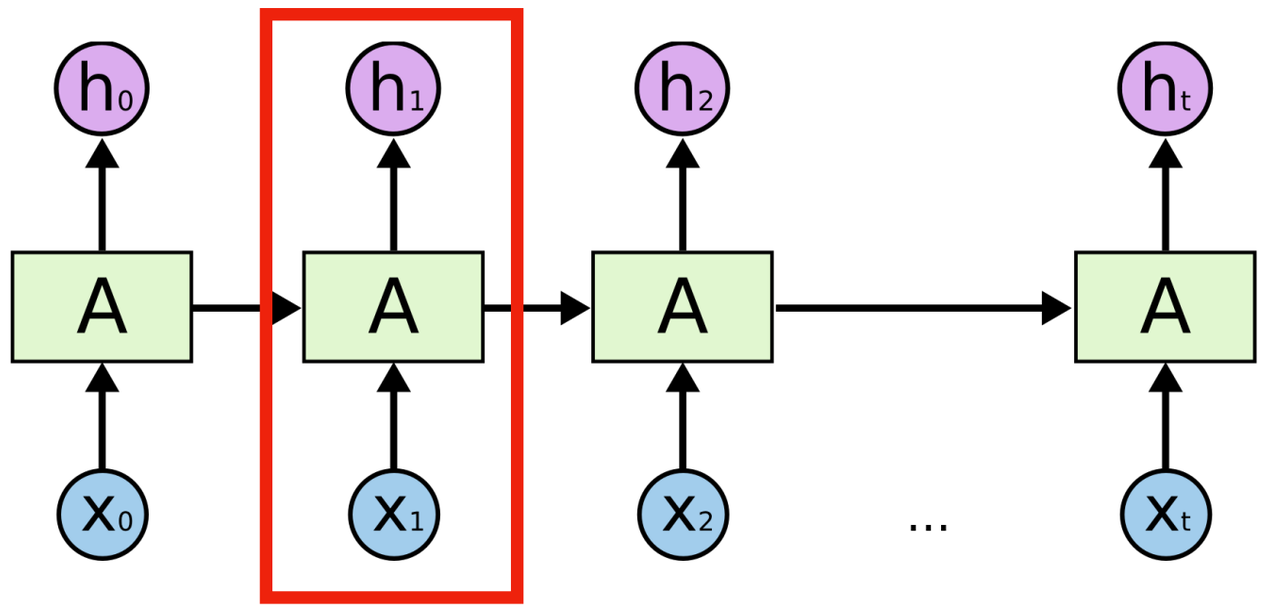
Recurrent Neural Networks



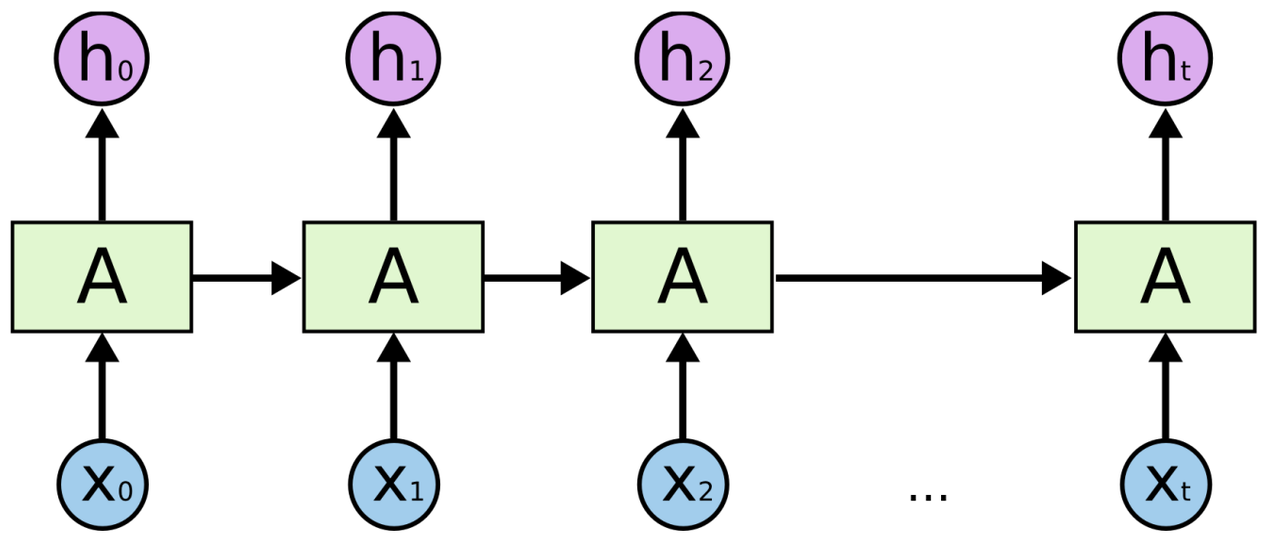
Recurrent Neural Networks



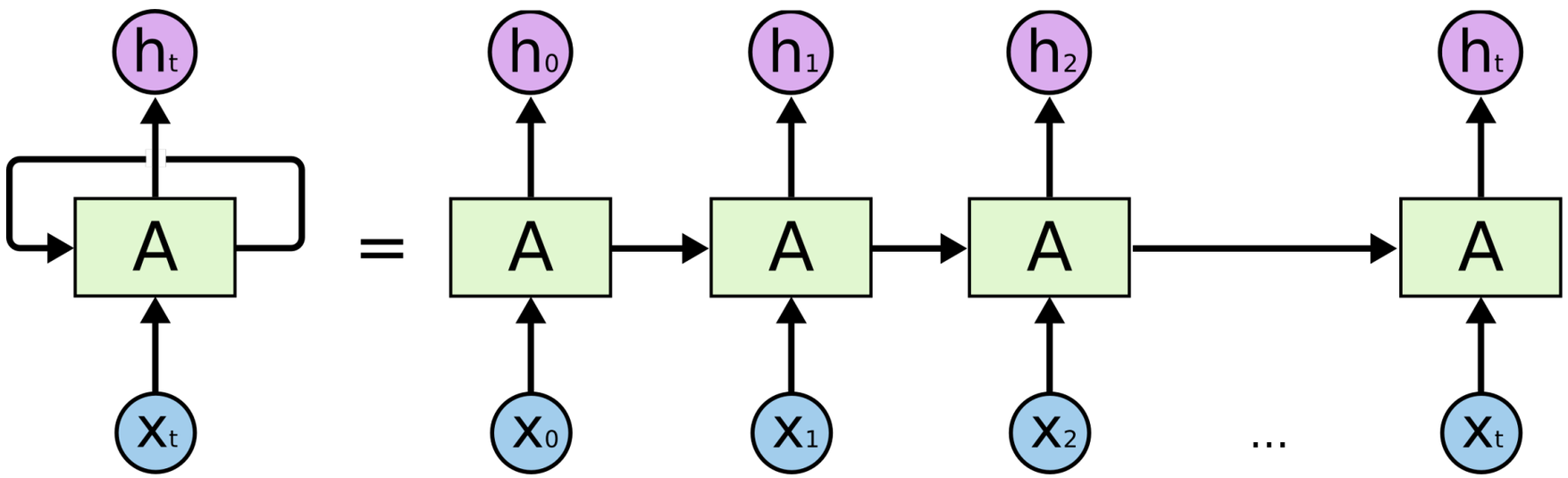
Recurrent Neural Networks



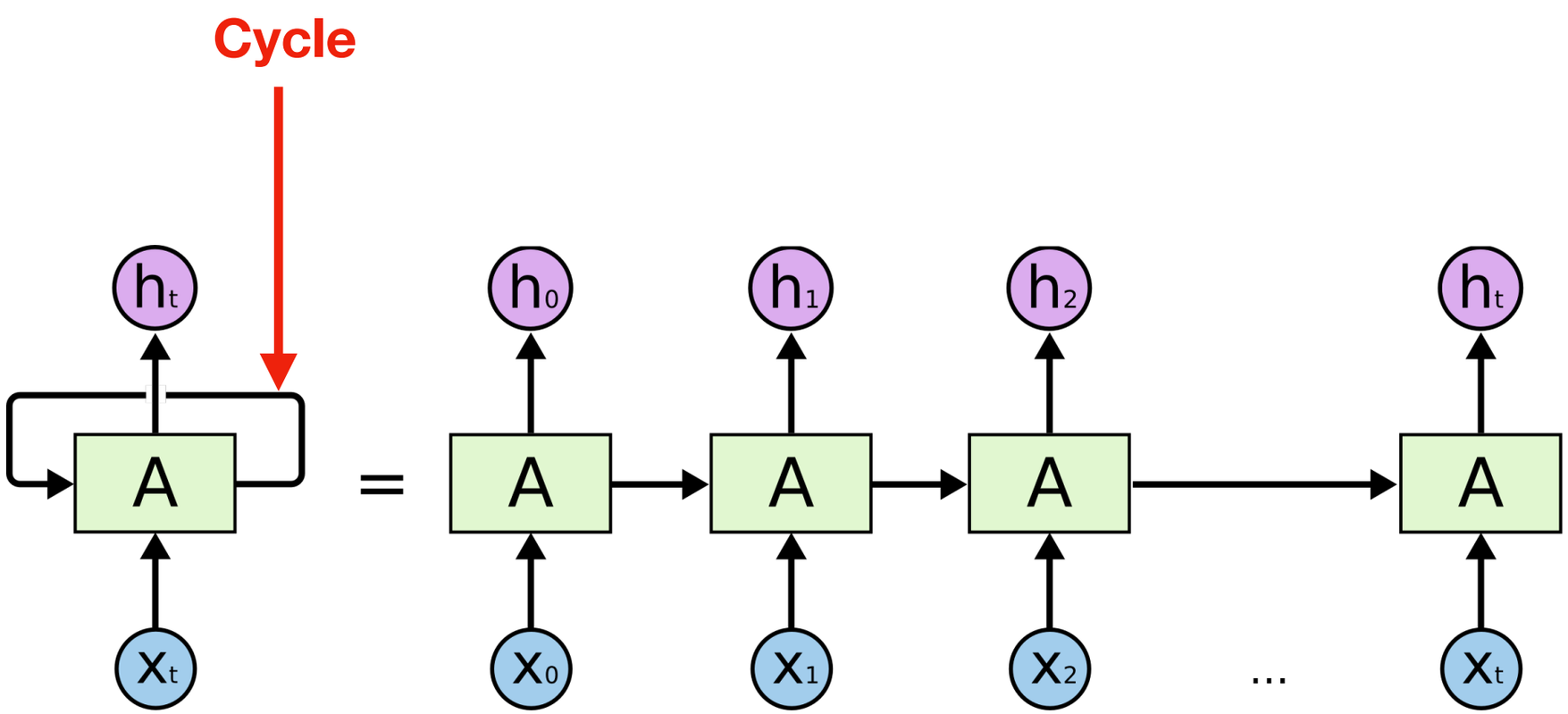
Recurrent Neural Networks



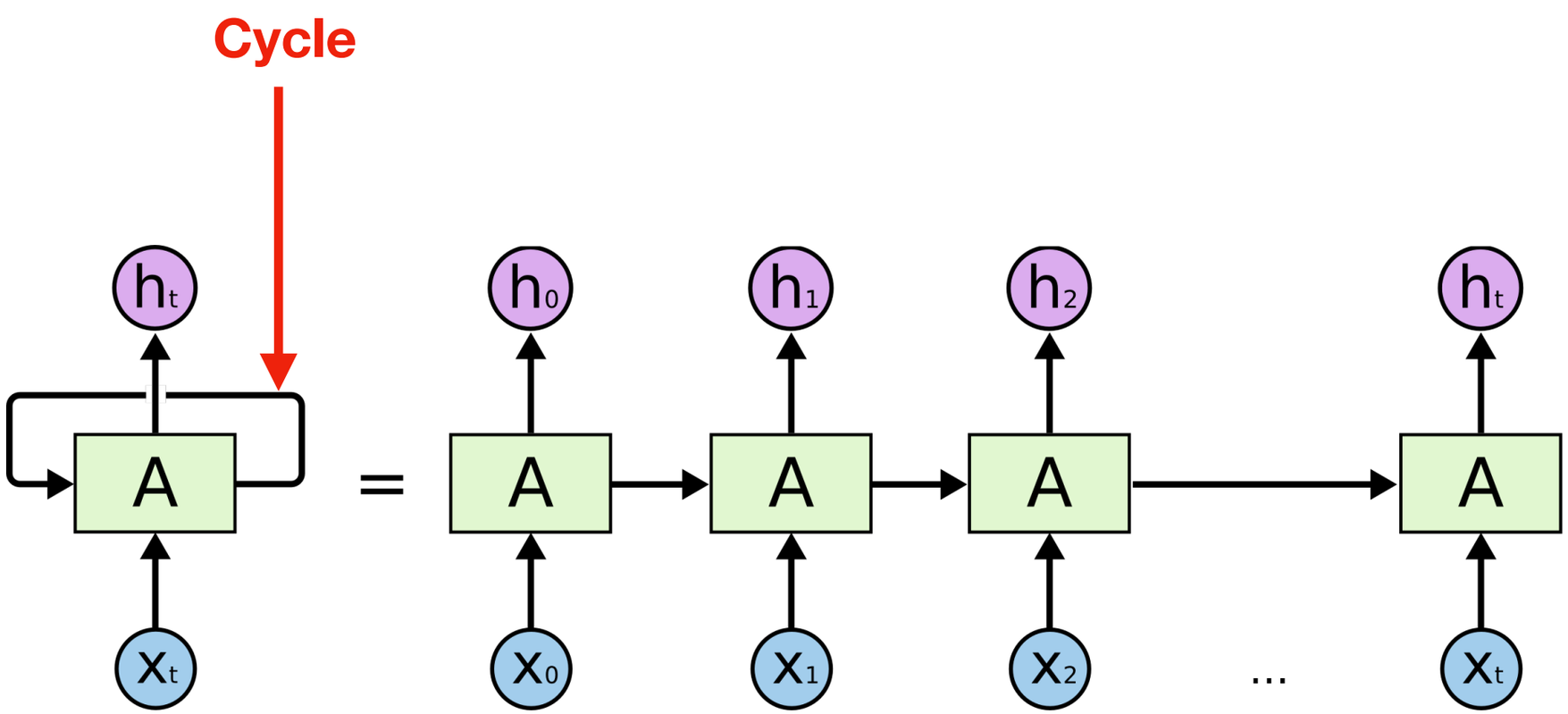
Recurrent Neural Networks



Recurrent Neural Networks

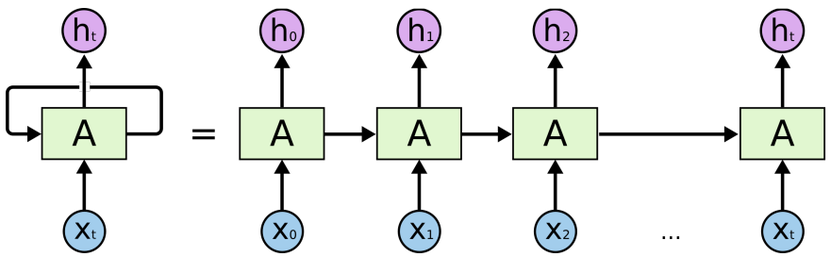


Recurrent Neural Networks

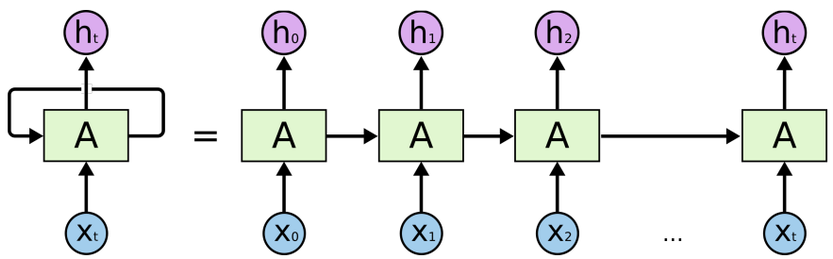


We use dependence analysis!

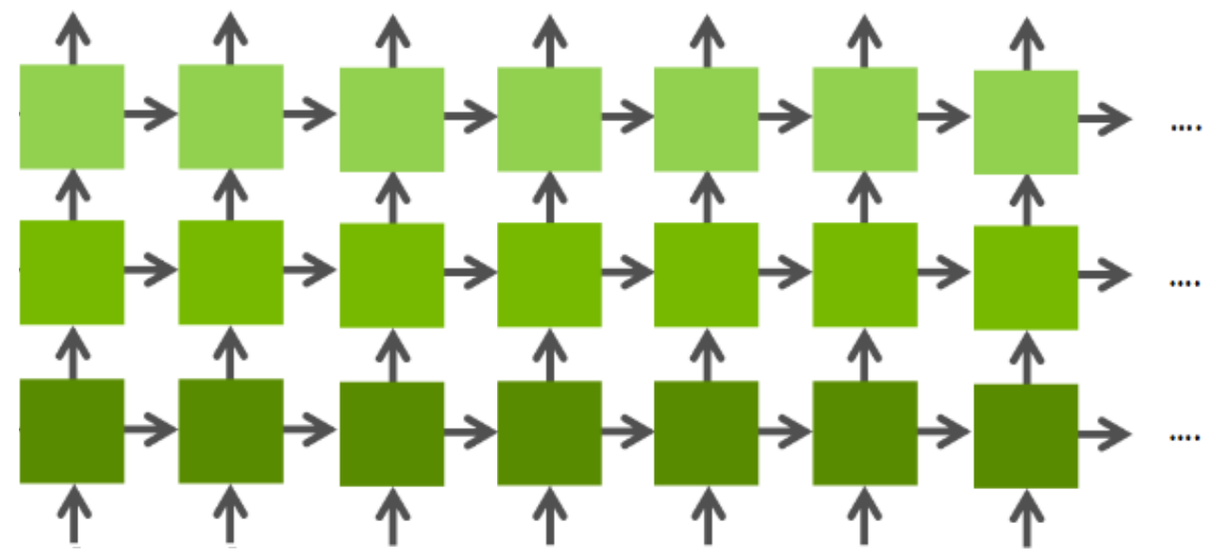
Recurrent Neural Networks



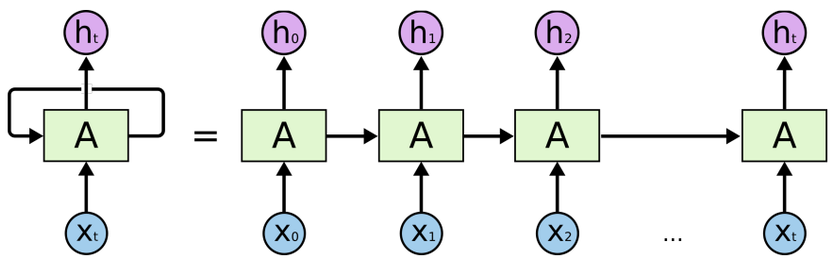
Recurrent Neural Networks



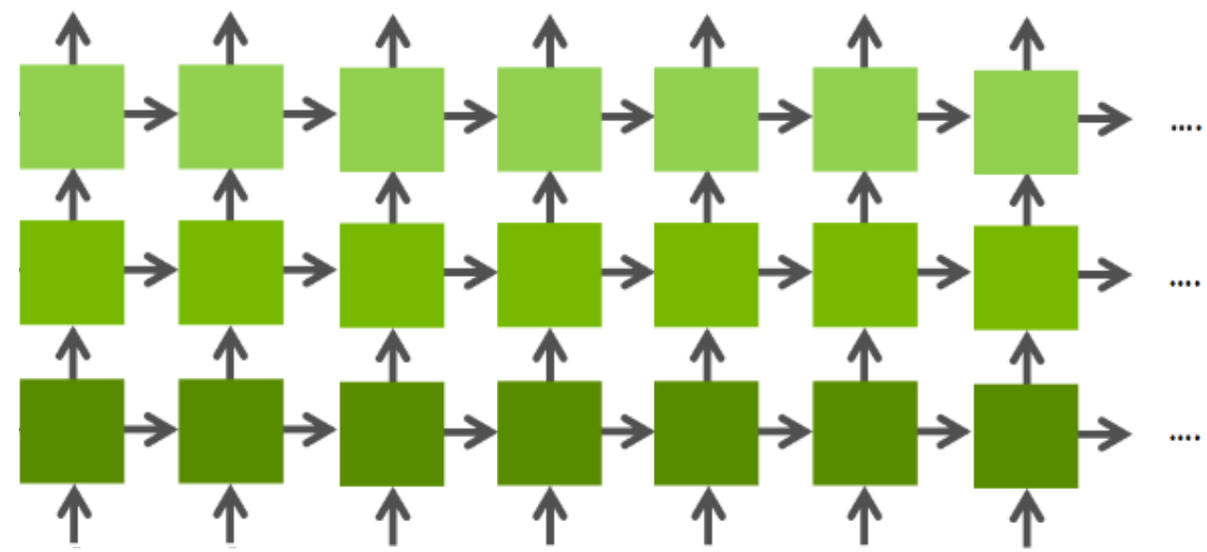
Multi-layer LSTM



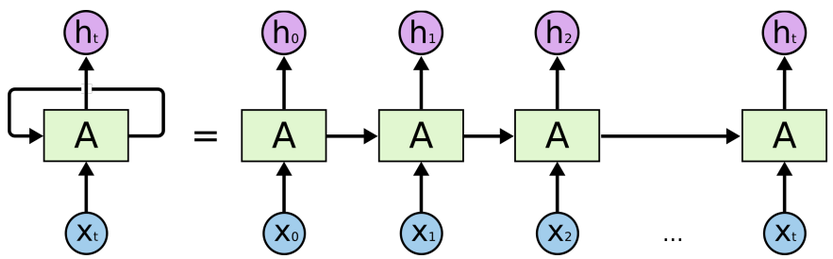
Recurrent Neural Networks

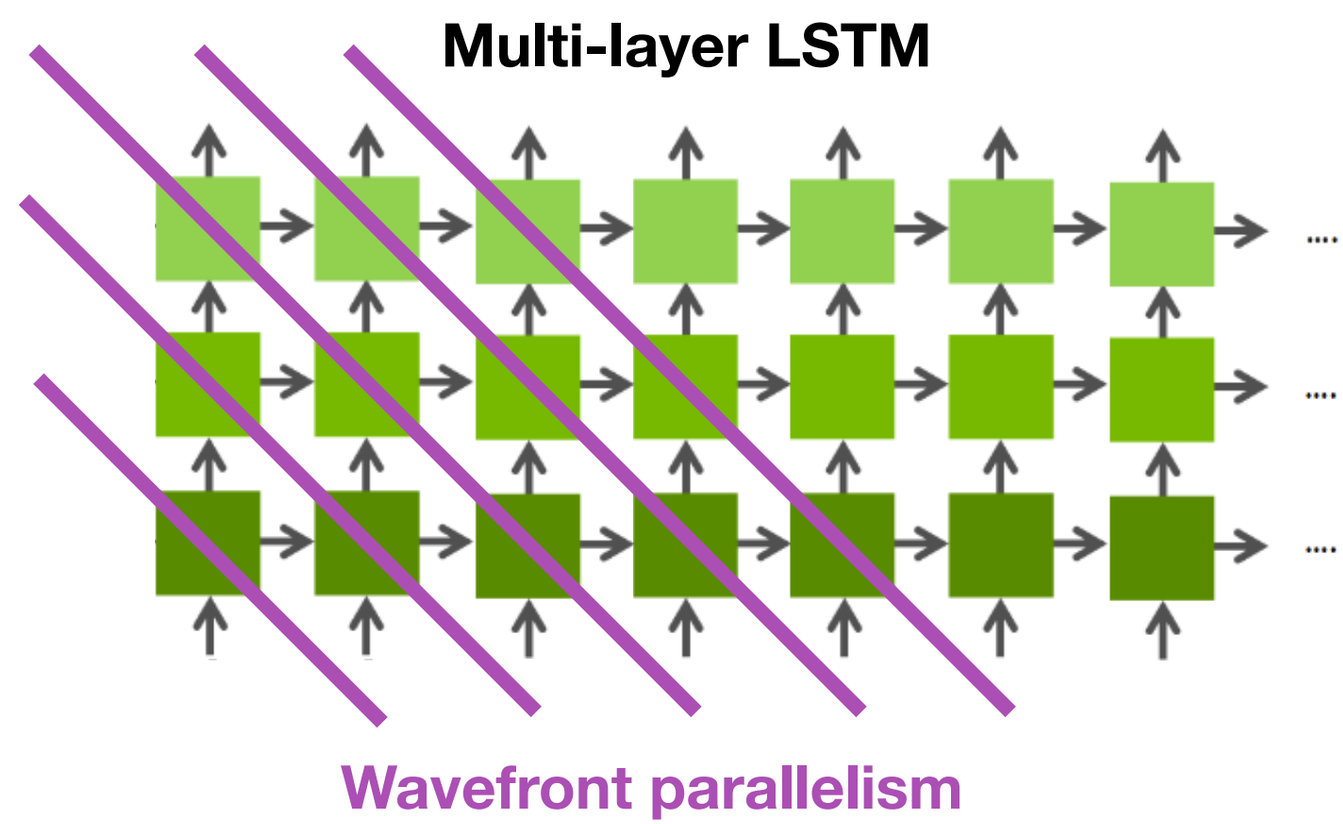


Multi-layer LSTM

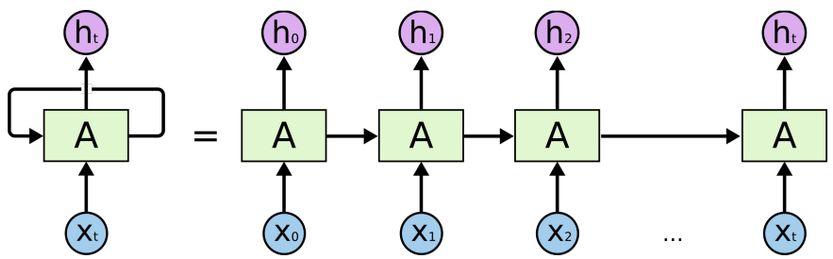


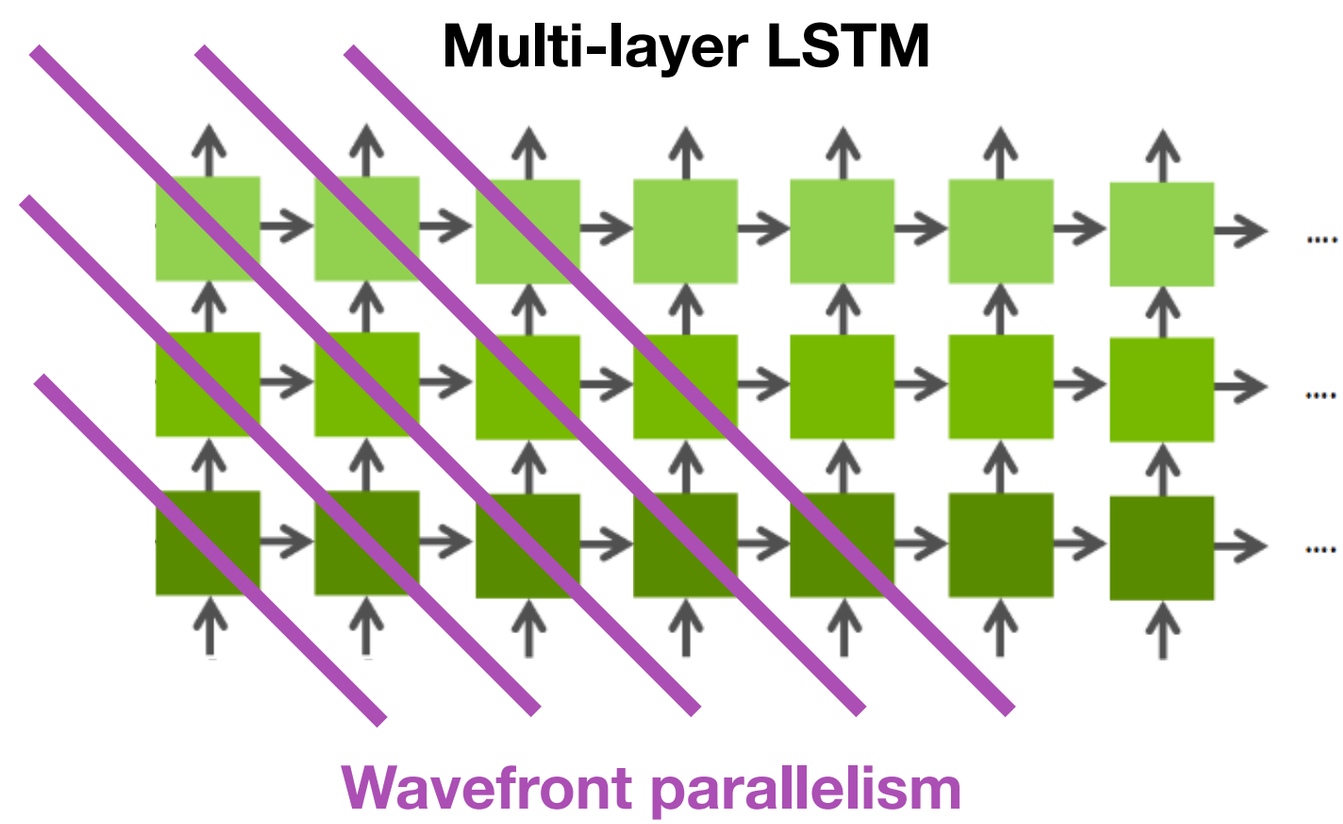
Recurrent Neural Networks





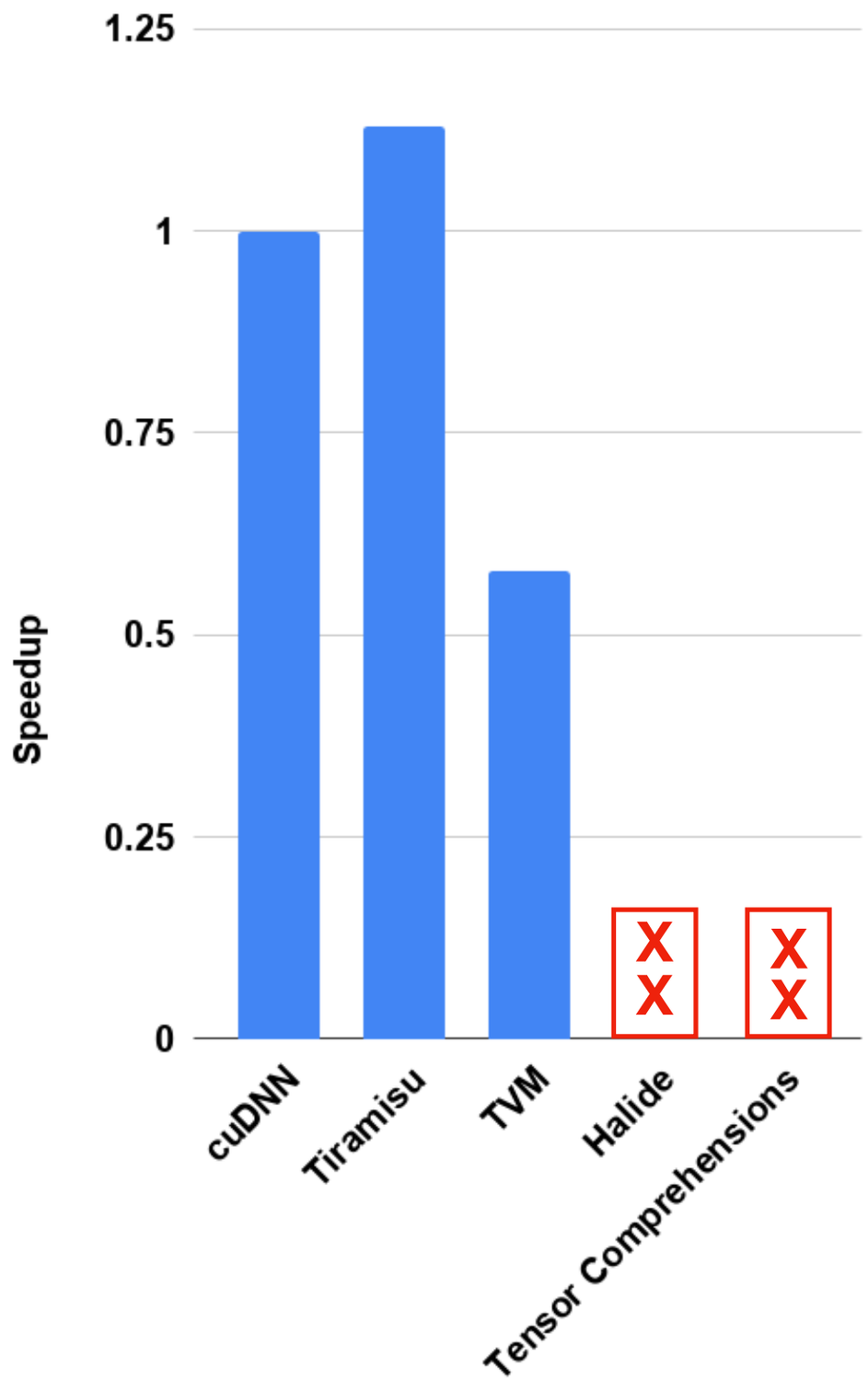
Recurrent Neural Networks



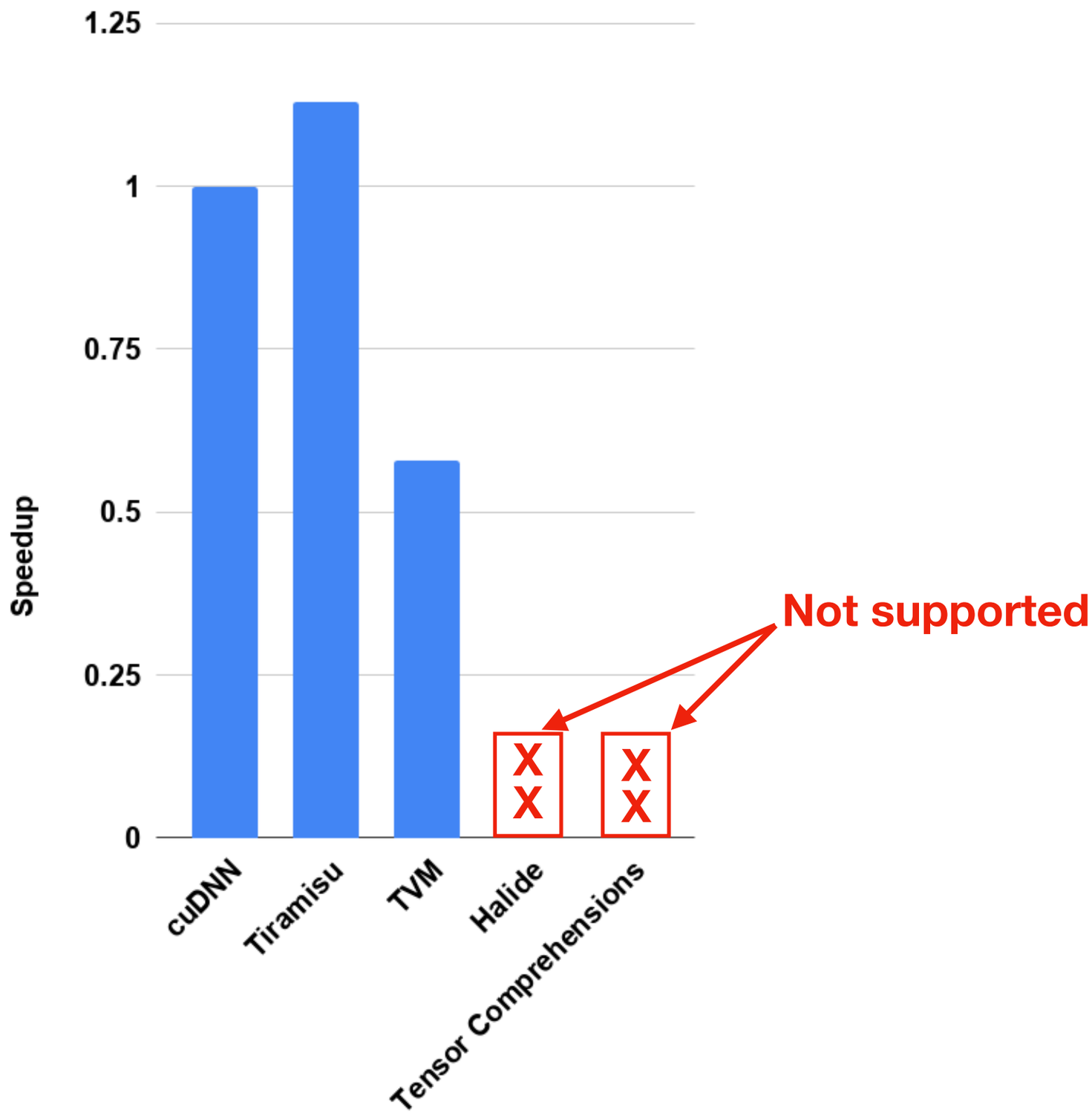


We use loop skewing

Multi-layer LSTM (GPU)



Multi-layer LSTM (GPU)



Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Outline

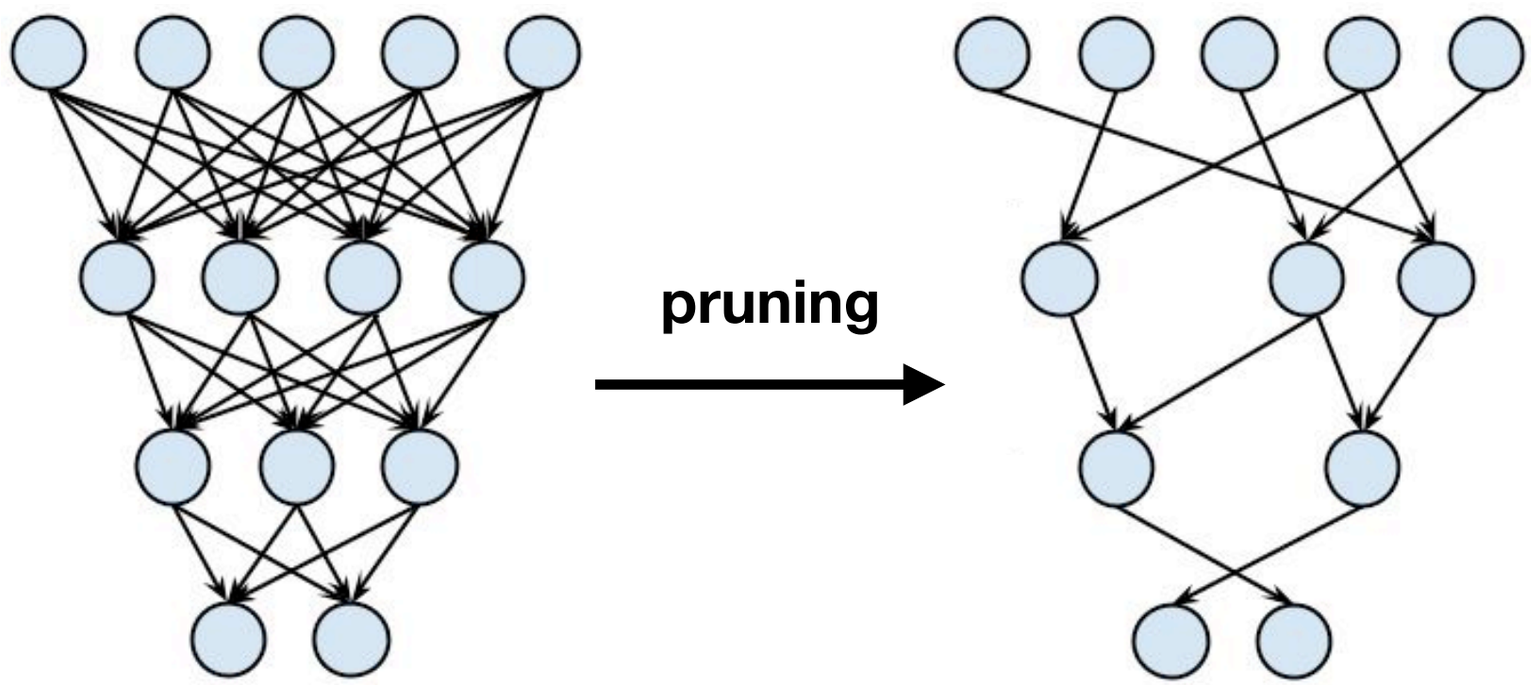
- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs
- **Phase II**
  - Automatic Optimization

Outline

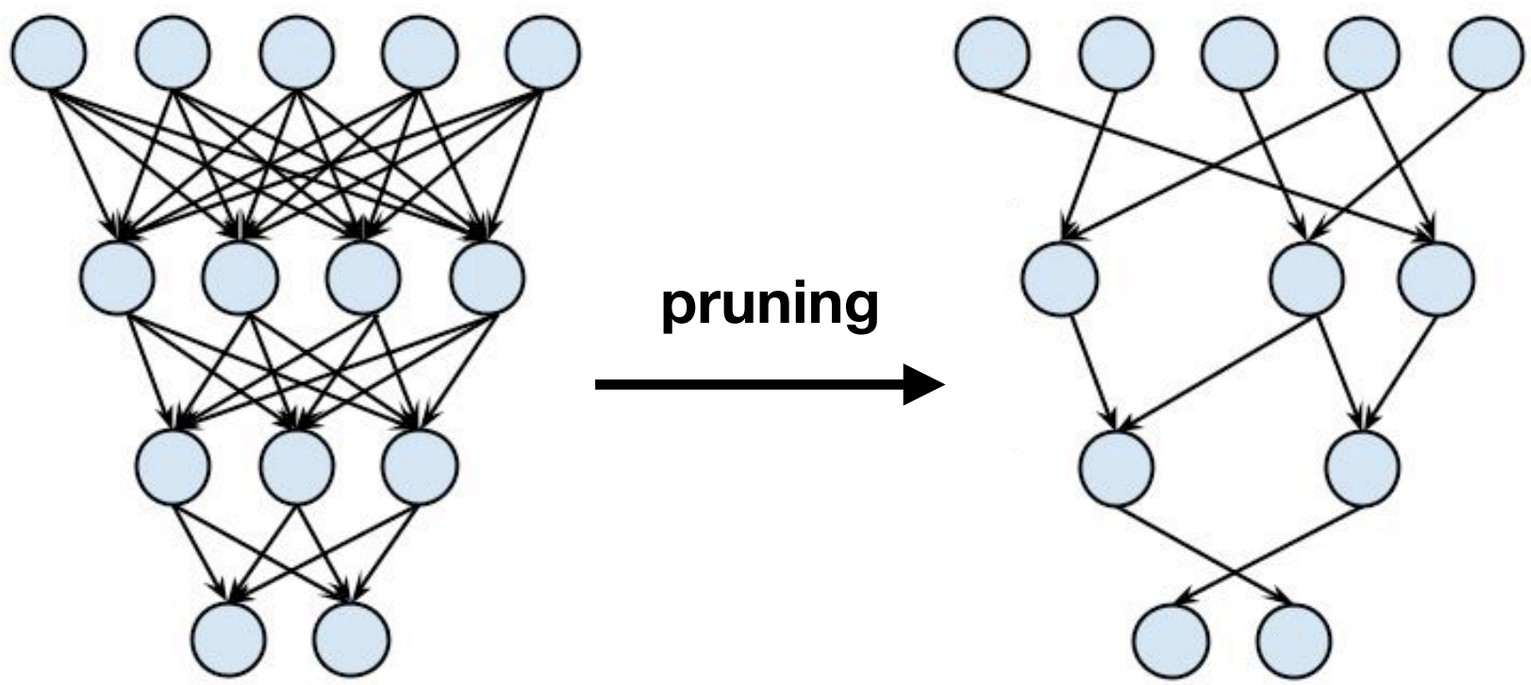
- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Exploiting Sparsity in DNN

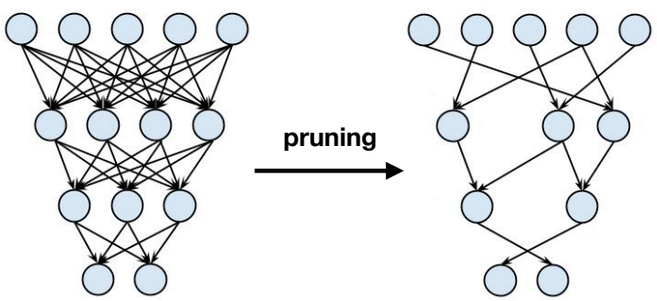


Exploiting Sparsity in DNN



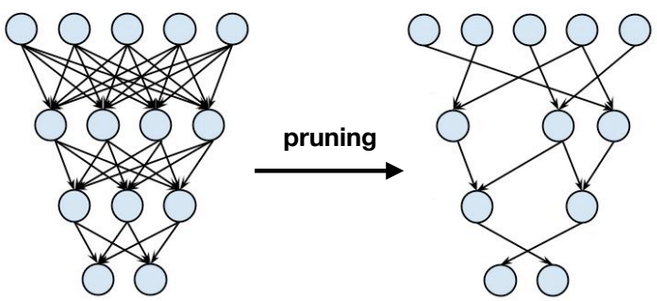
Reduce weights by orders of magnitude

Exploiting Sparsity in DNN

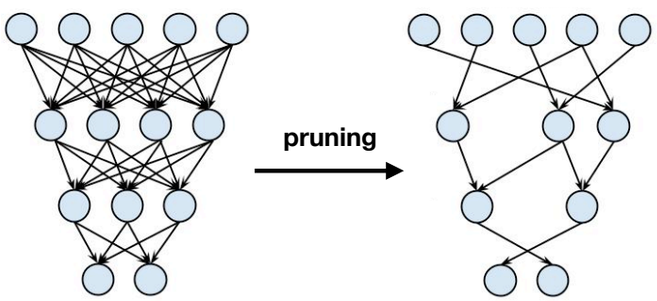


Exploiting Sparsity in DNN

Compiling sparse code is challenging!



Exploiting Sparsity in DNN

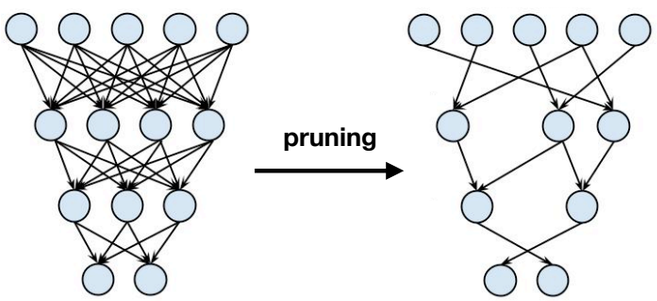


Compiling sparse code is challenging!

Irregular loop bounds

```
.........
for j in (rowptr[n], rowptr[n+1])
.........
```

Exploiting Sparsity in DNN

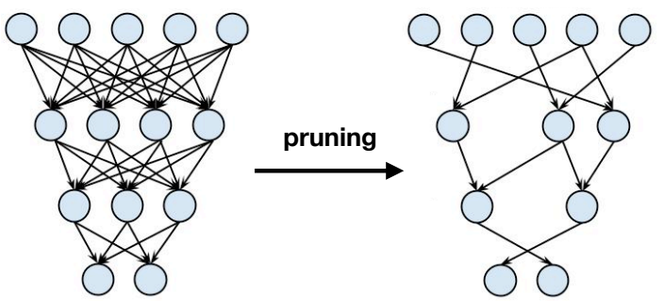


Compiling sparse code is challenging!

Irregular loop bounds

```
.........
for j in (rowptr[n], rowptr[n+1])
.........
```

Exploiting Sparsity in DNN



Compiling sparse code is challenging!

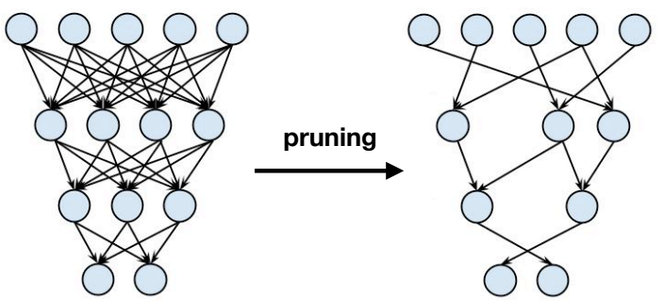
Irregular loop bounds

```
.........
for j in (rowptr[n], rowptr[n+1])
.........
```

Indirect array accesses

```
.........
out[n][y][x] = … + in[colidx[j]]
.........
```

Exploiting Sparsity in DNN



Compiling sparse code is challenging!

Irregular loop bounds

```
.........
for j in (rowptr[n], rowptr[n+1])
.........
```

Indirect array accesses

```
.........
out[n][y][x] = … + in[colidx[j]]
.........
```

Sparsity in DNNs

<u>Median sparsity</u>

ResNet: 20%

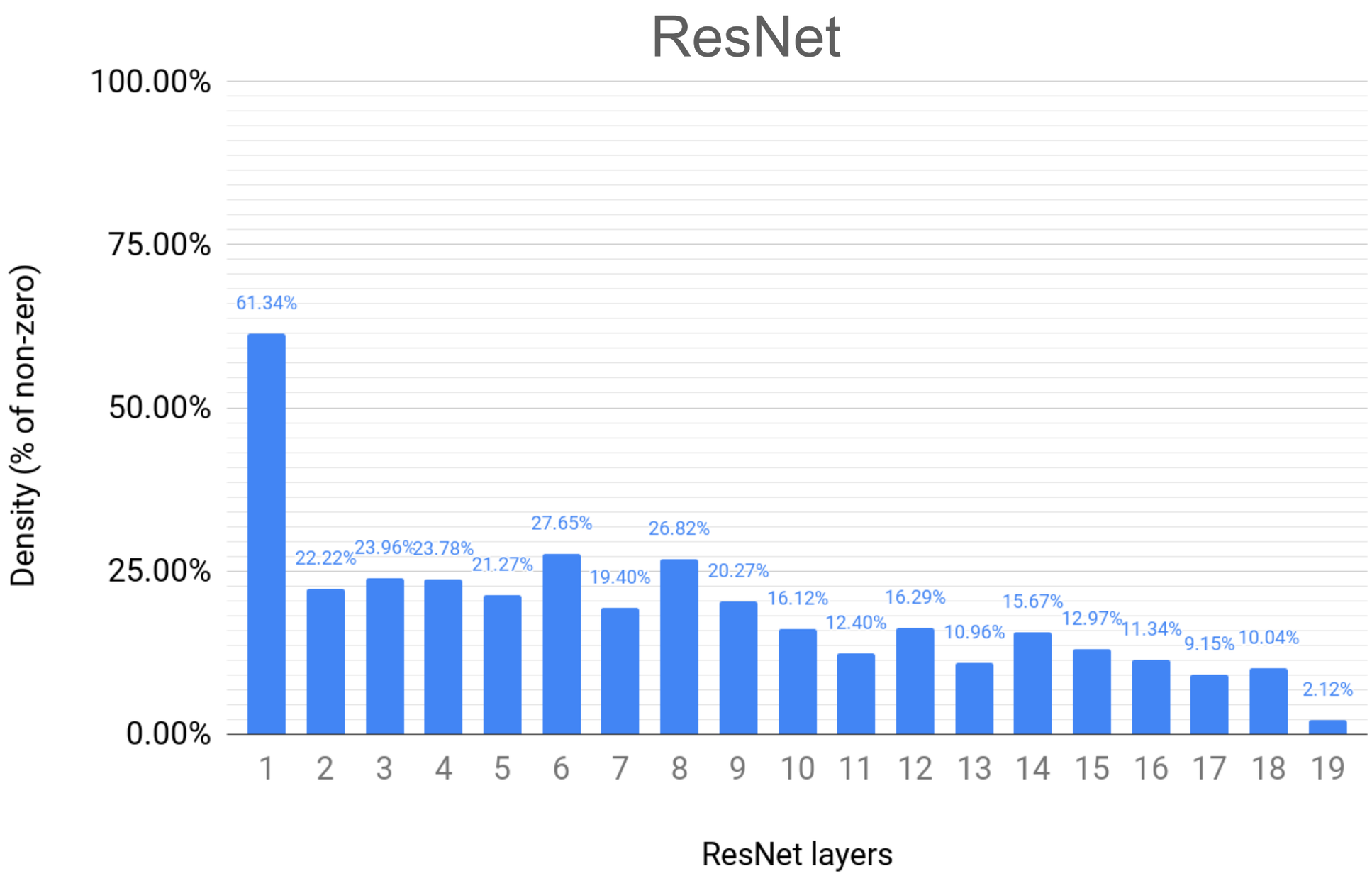
VGG: 2%

[Frankle & Carbin, 2019]

Sparsity in DNNs

[Frankle & Carbin, 2019]

Sparsity in DNNs

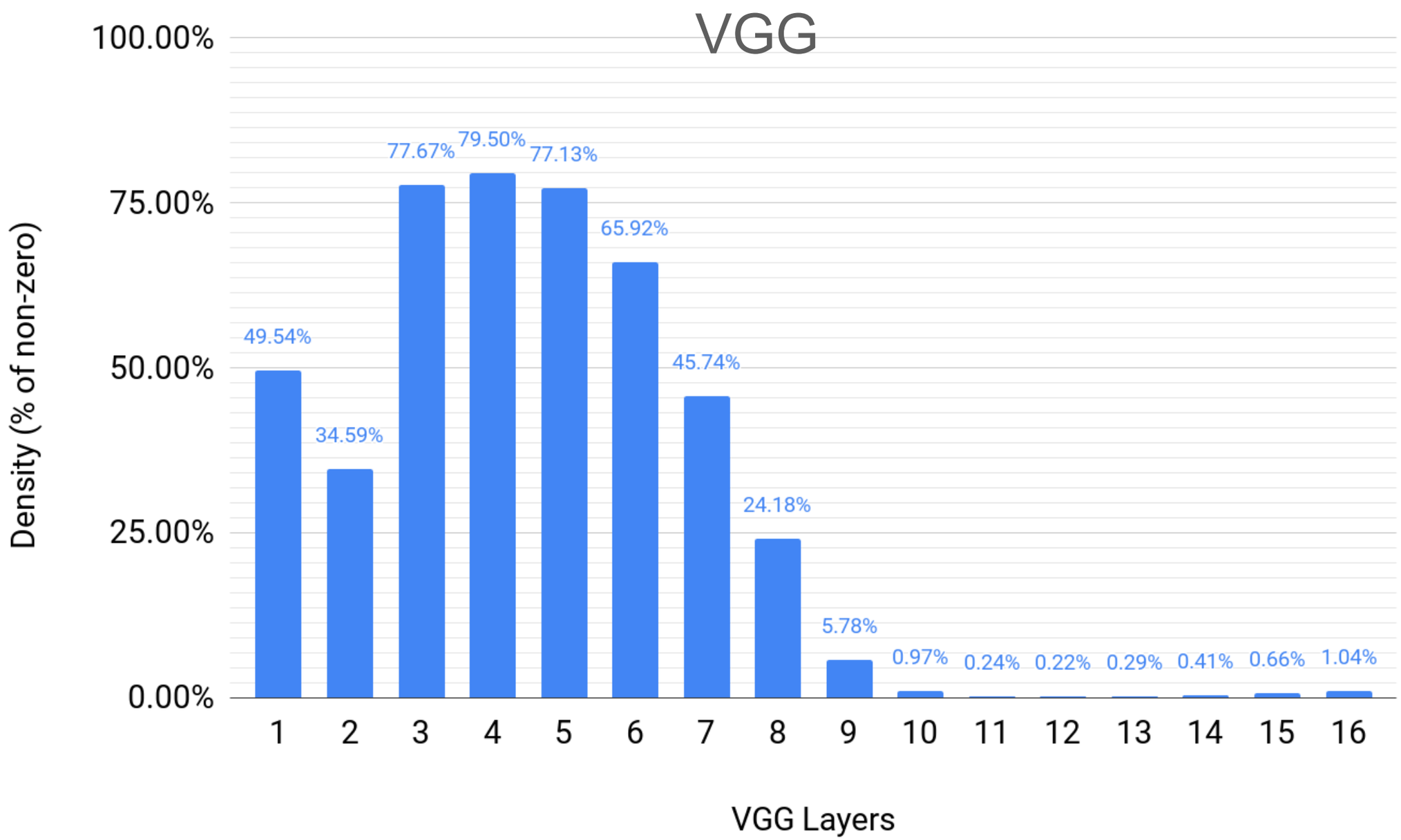


[Frankle & Carbin, 2019]

Sparsity in DNNs

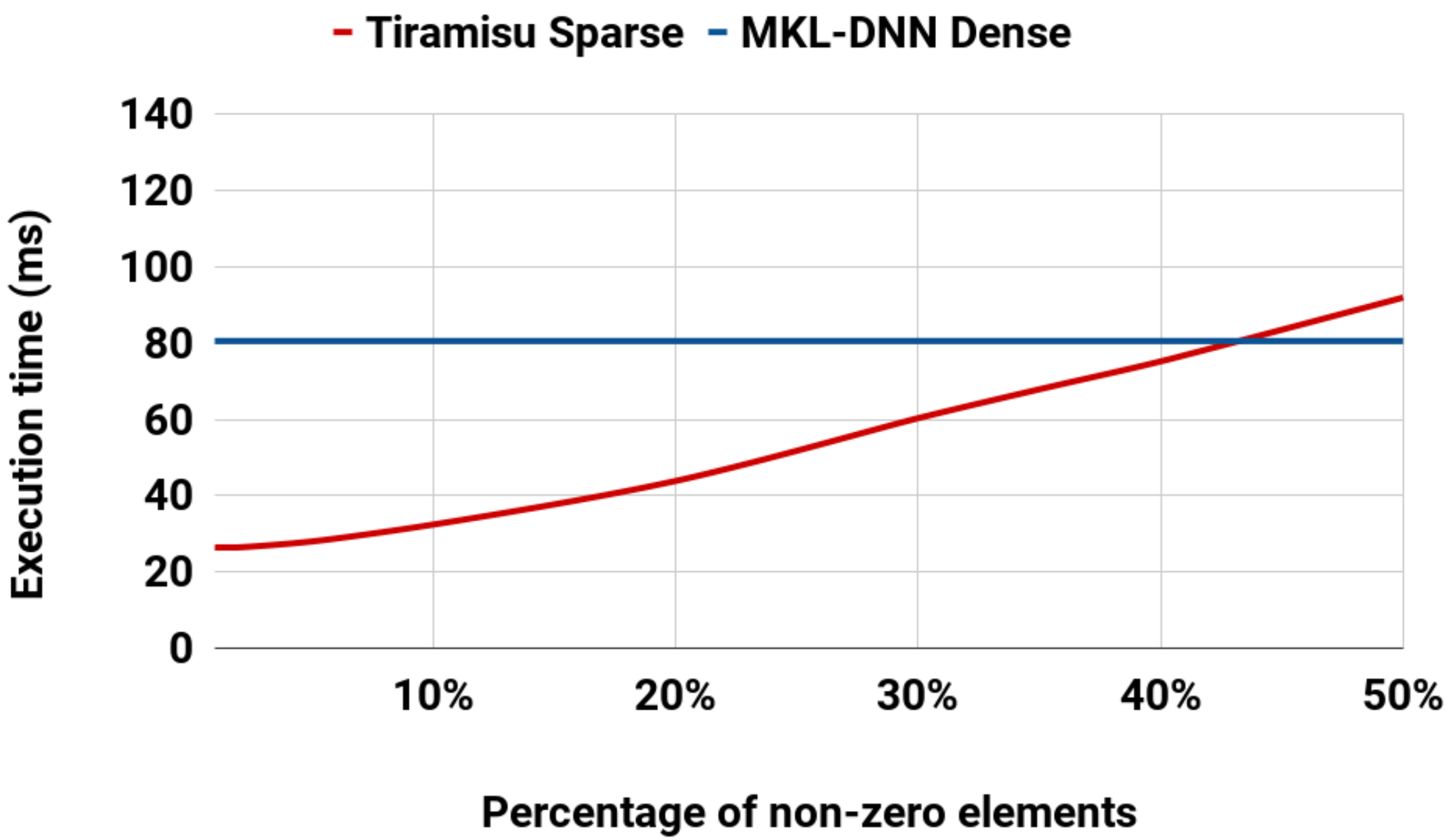
[Frankle & Carbin, 2019]

Sparsity in DNNs

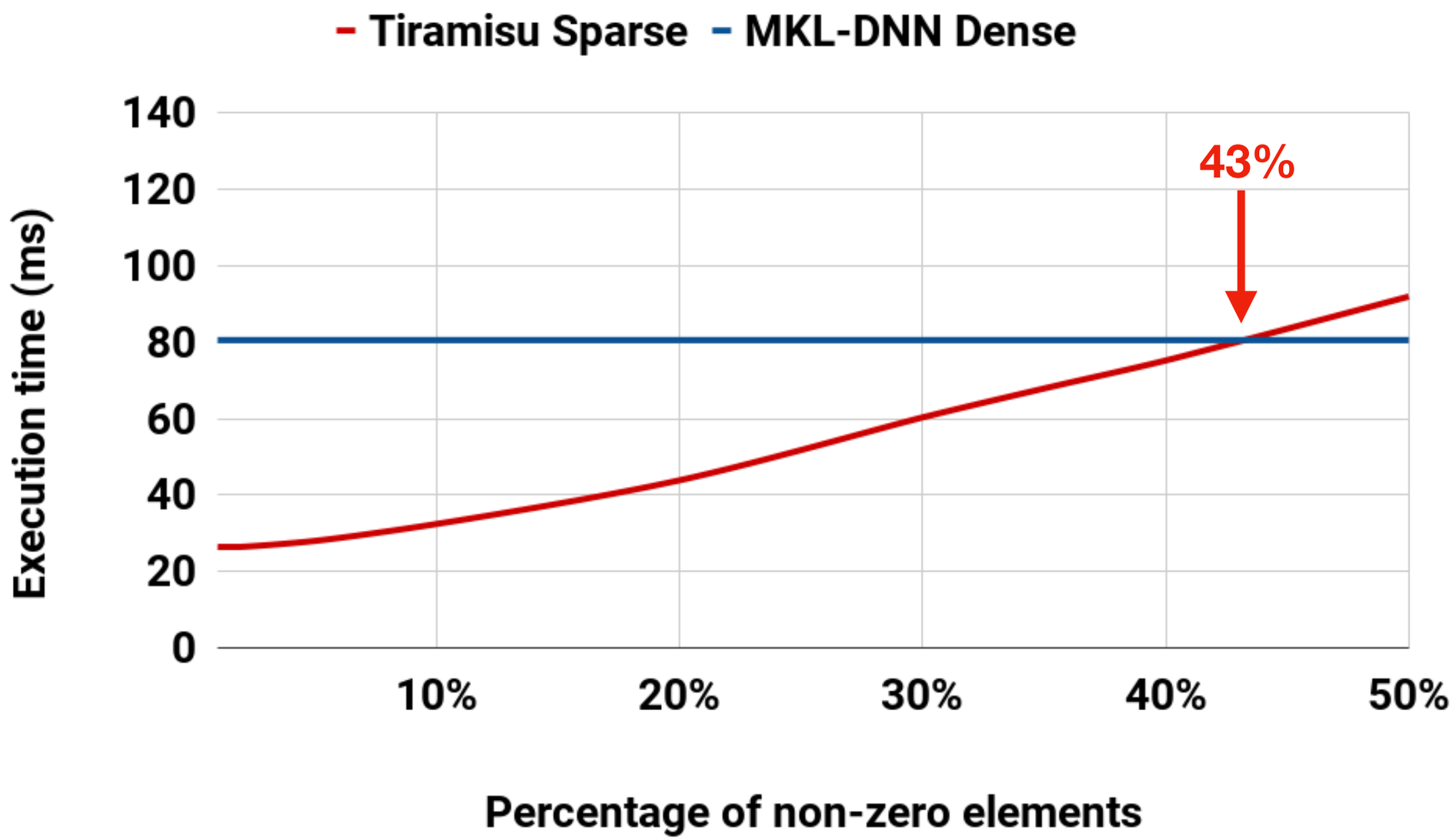


[Frankle & Carbin, 2019]

Break-even Density Level



Break-even Density Level



Optimizing Sparse DNNs

Optimizing Sparse DNNs

- Focus on weight sparsity

Optimizing Sparse DNNs

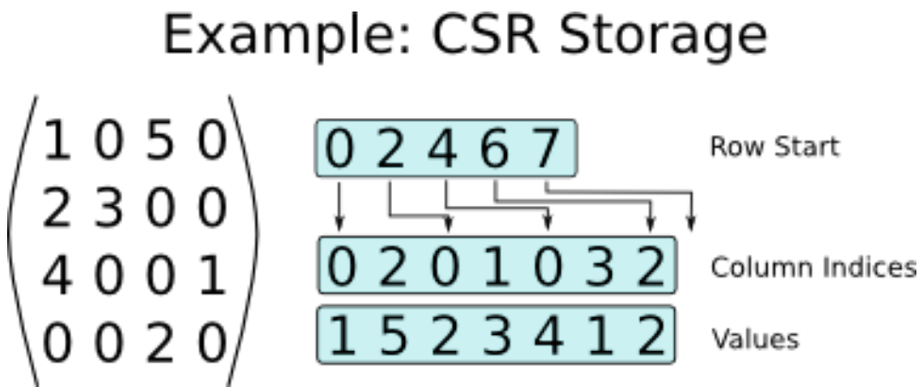
- Focus on weight sparsity
- CSR format for the weights

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

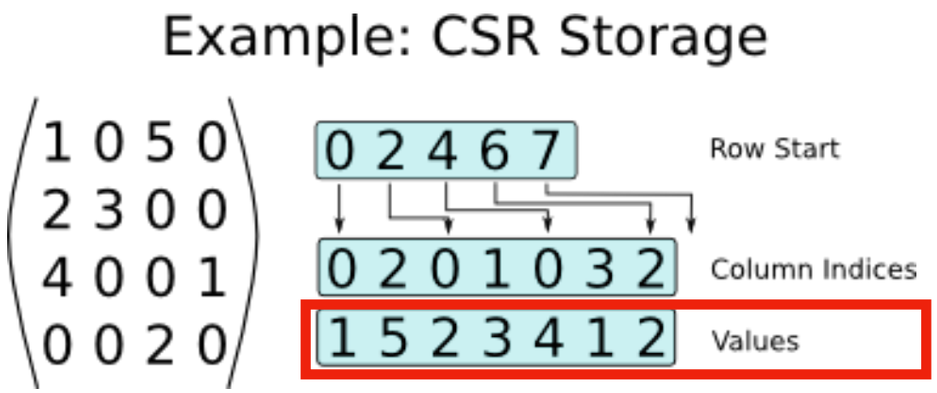
Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



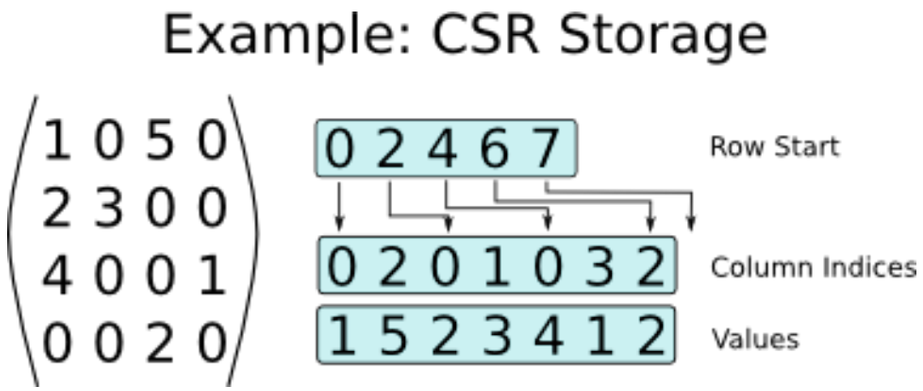
Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



Optimizing Sparse DNNs

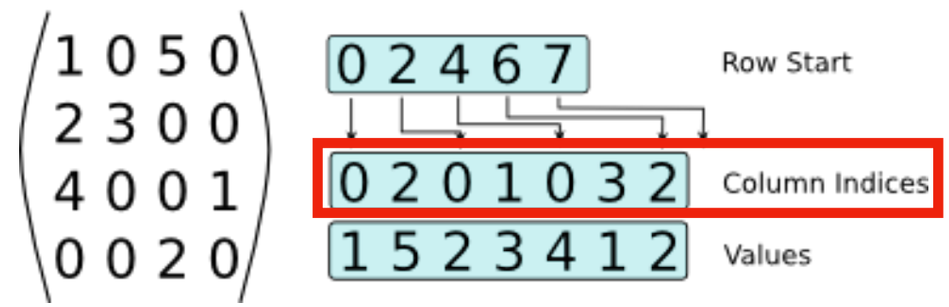
- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



Optimizing Sparse DNNs

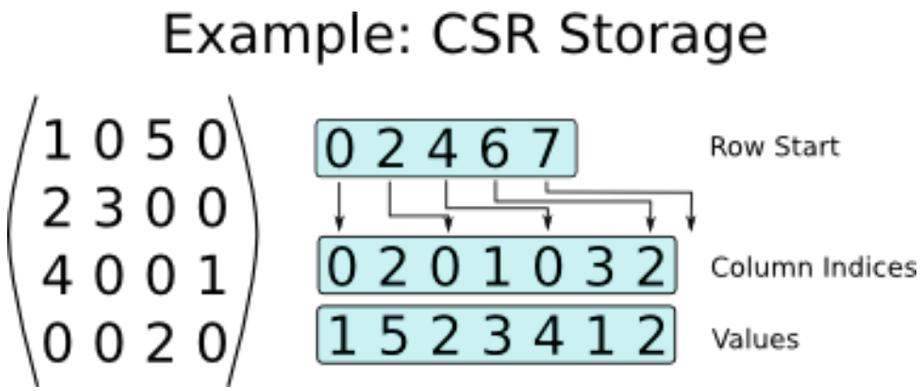
- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

Example: CSR Storage



Optimizing Sparse DNNs

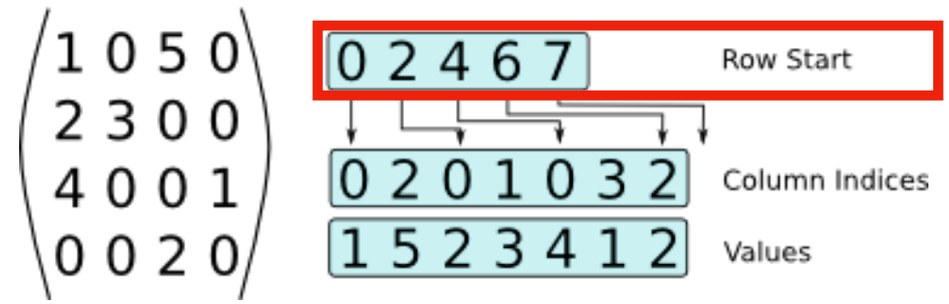
- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



Optimizing Sparse DNNs

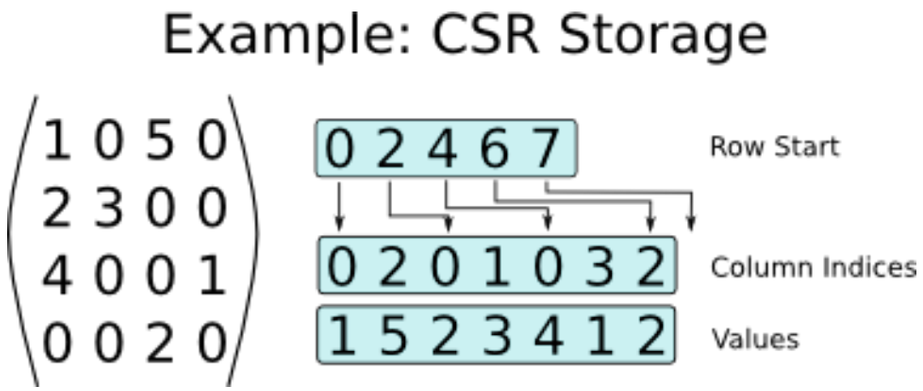
- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

Example: CSR Storage



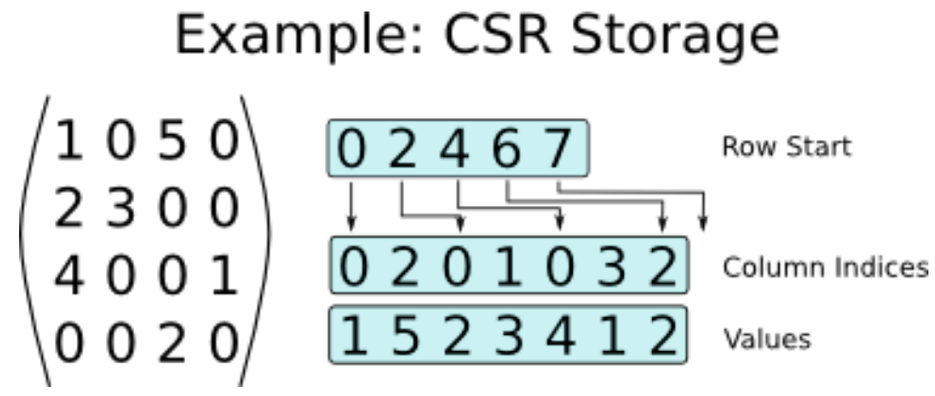
Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

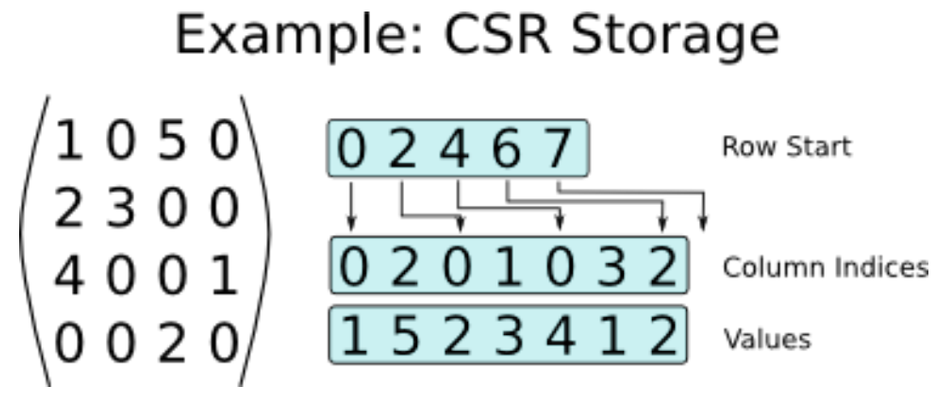



```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

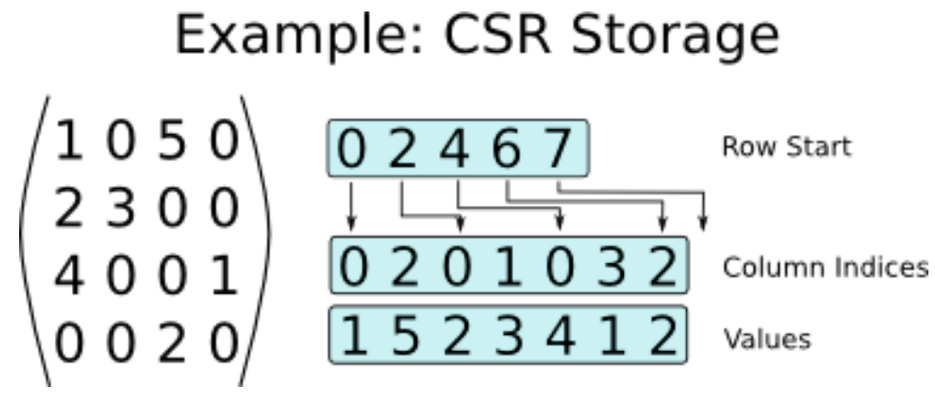



```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution



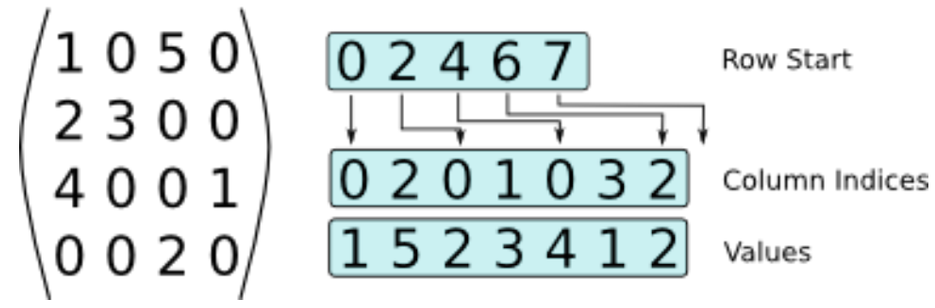

```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

Example: CSR Storage



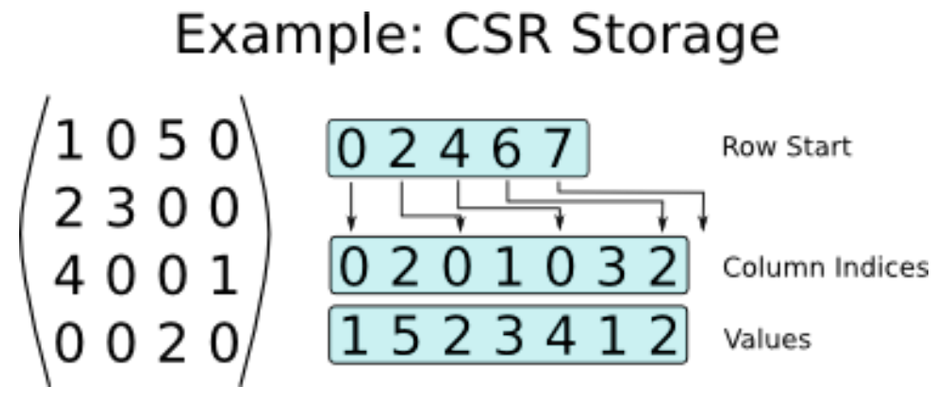

```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

Get a non-zero weight

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

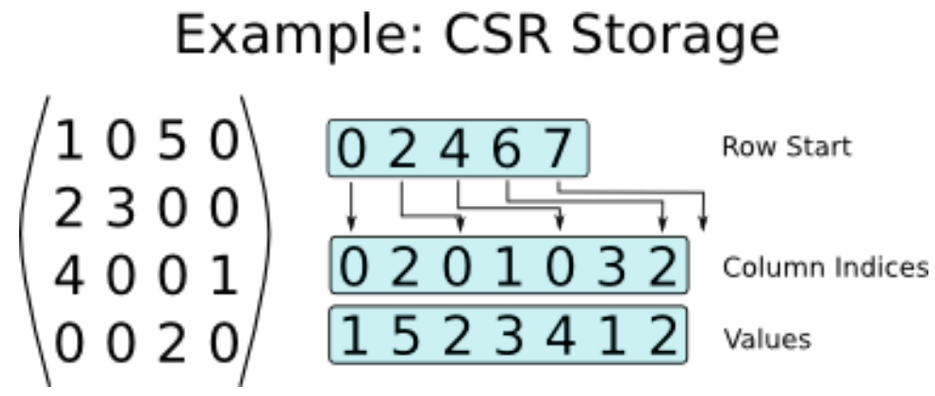



```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution

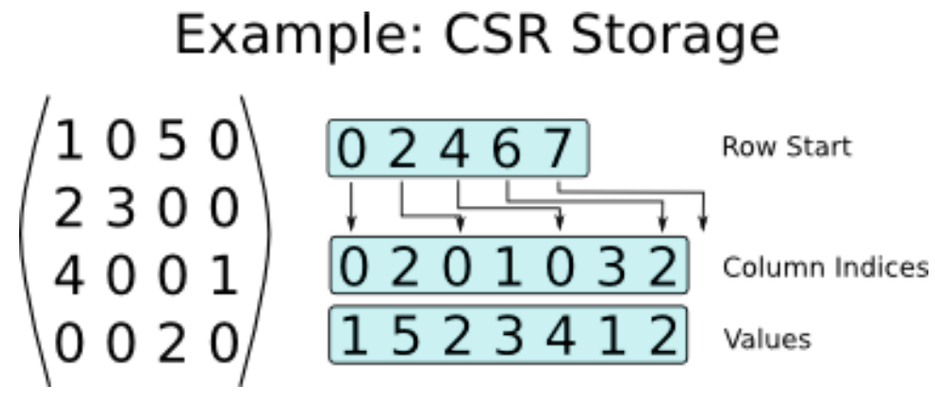



```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Optimizing Sparse DNNs

- Focus on weight sparsity
- CSR format for the weights
- Use direct convolution




```
for each output channel n
  for j in (W.rowptr[n], W.rowptr[n+1])
  {
    off = W.colidx[j];
    coeff = W.value[j];
    for (int y = 0; y < H_OUT; ++y)
      for (int x = 0; x < W_OUT; ++x)
        out[n][y][x] += coeff*in[y*W_OUT+x+off)]
  }
```

[Park et al, 2016]

Extensions to the Polyhedral Model

Extensions to the Polyhedral Model

- Indirect array accesses

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

```
for (int y = 0; y < H_OUT; ++y)
  p = A[y]
  if (p)
    out[y] = …
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

```
for (int y = 0; y < H_OUT; ++y)
  p = A[y]
  if (p)
    out[y] = …
```

Extensions to the Polyhedral Model

- Indirect array accesses

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    out[n][y][x] += …*in[colidx[j]]
```

```
for (int y = 0; y < H_OUT; ++y)
  for (int x = 0; x < W_OUT; ++x)
    p = colidx[j]
    out[n][y][x] += …*in[p]
```

- Irregular loop bounds

```
for j in (rowptr[n], rowptr[n+1])
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

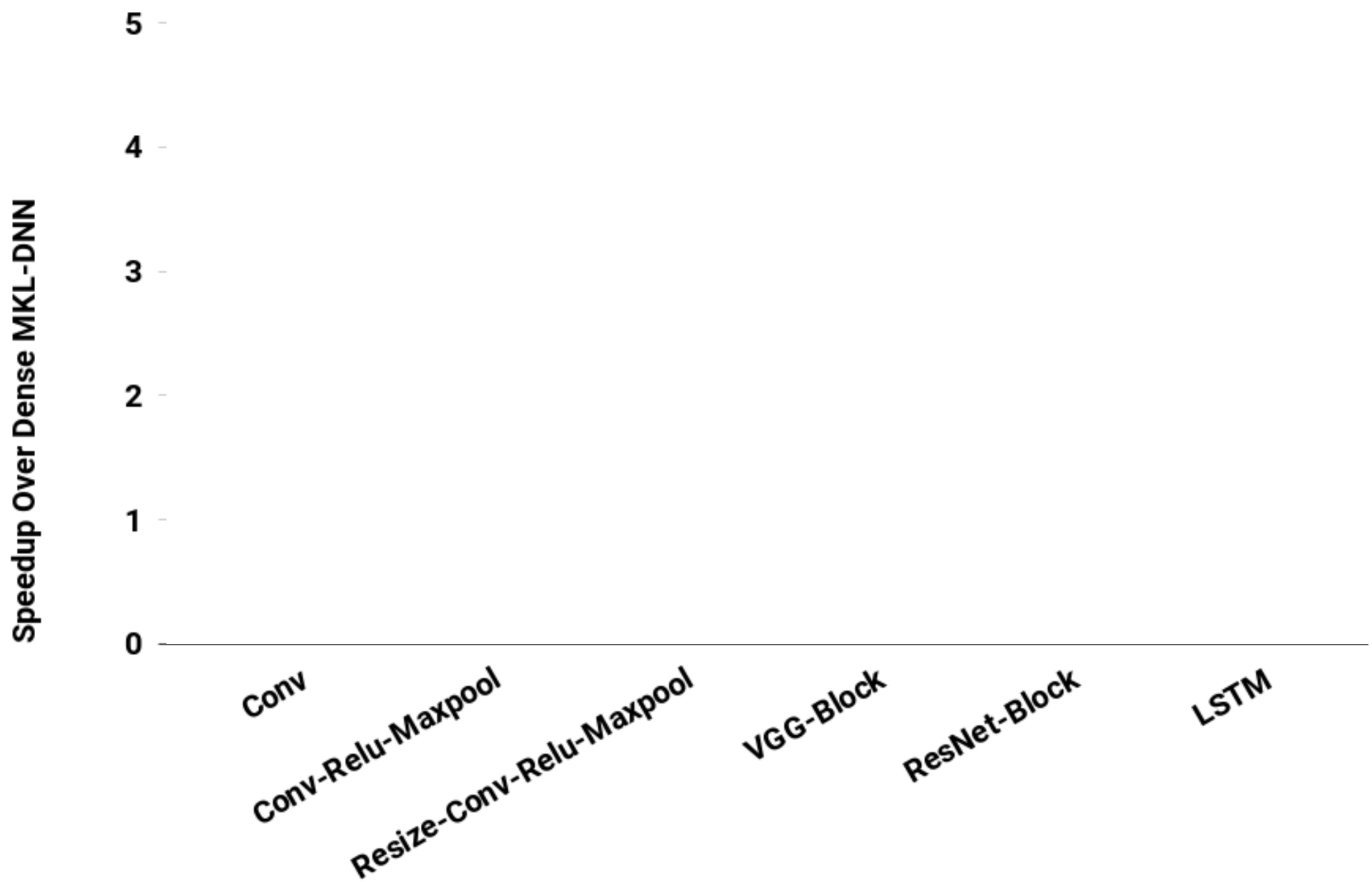
```
b0 = rowptr[n]; b1 = rowptr[n+1];
for j in (b0, b1)
  for (int y = 0; y < H_OUT; ++y)
    for (int x = 0; x < W_OUT; ++x)
      out[n][y][x] += …*in[…]
```

- Irregular conditionals

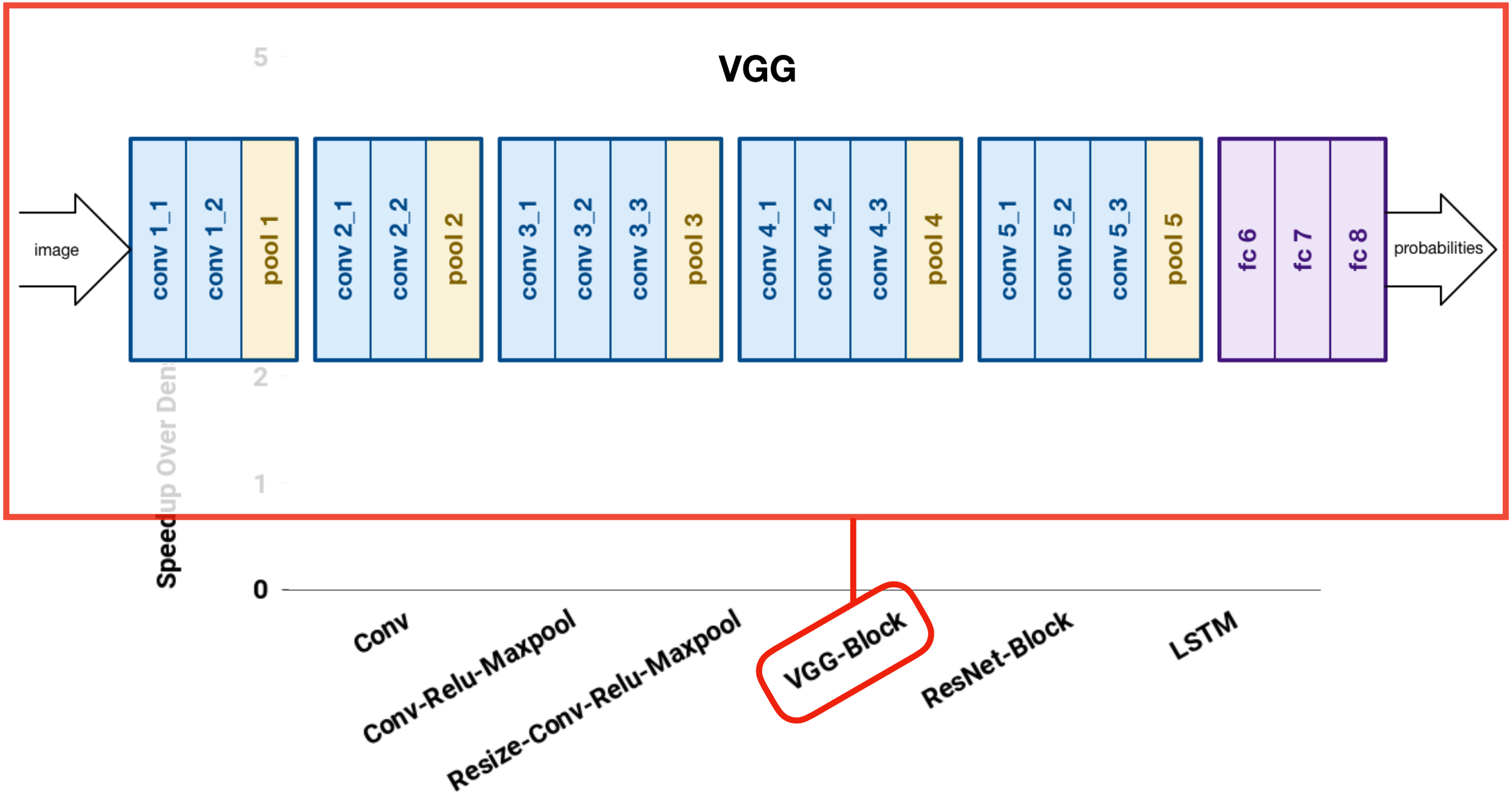
```
for (int y = 0; y < H_OUT; ++y)
  if (A[y])
    out[y] = …
```

```
for (int y = 0; y < H_OUT; ++y)
  p = A[y]
  if (p)
    out[y] = …
```

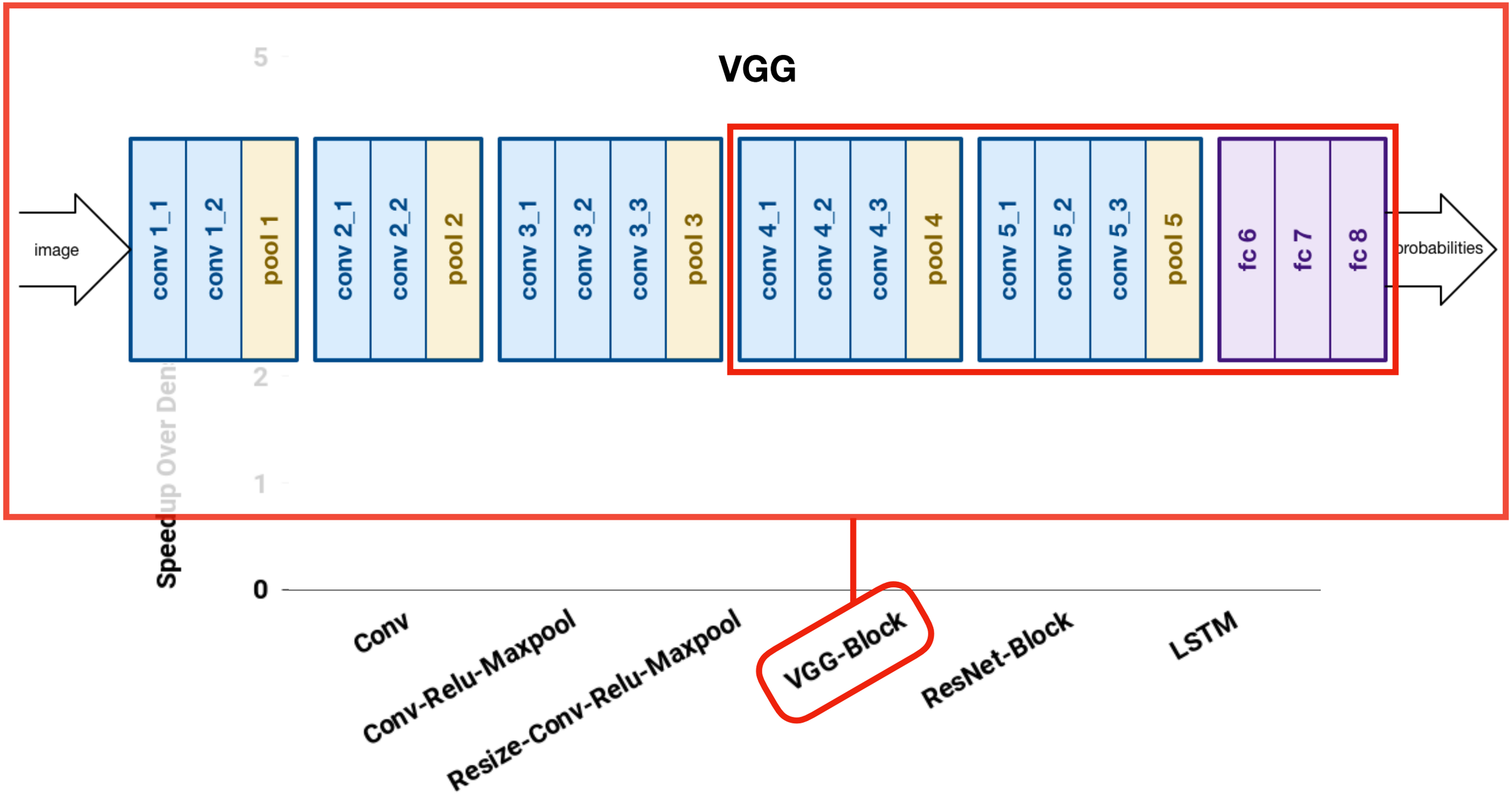
Tiramisu Dense and Sparse (CPU - 4 cores)



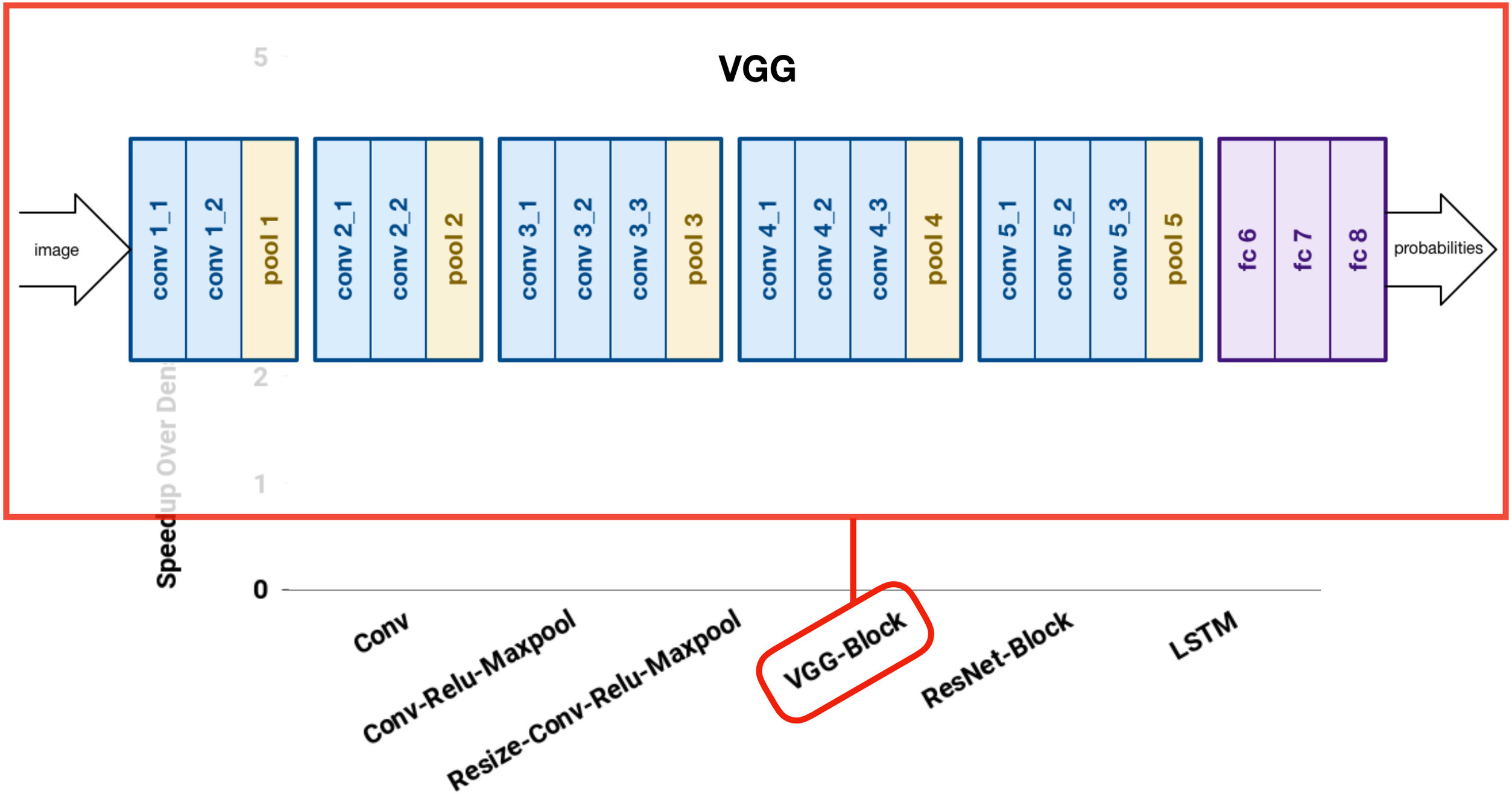
Tiramisu Dense and Sparse (CPU - 4 cores)



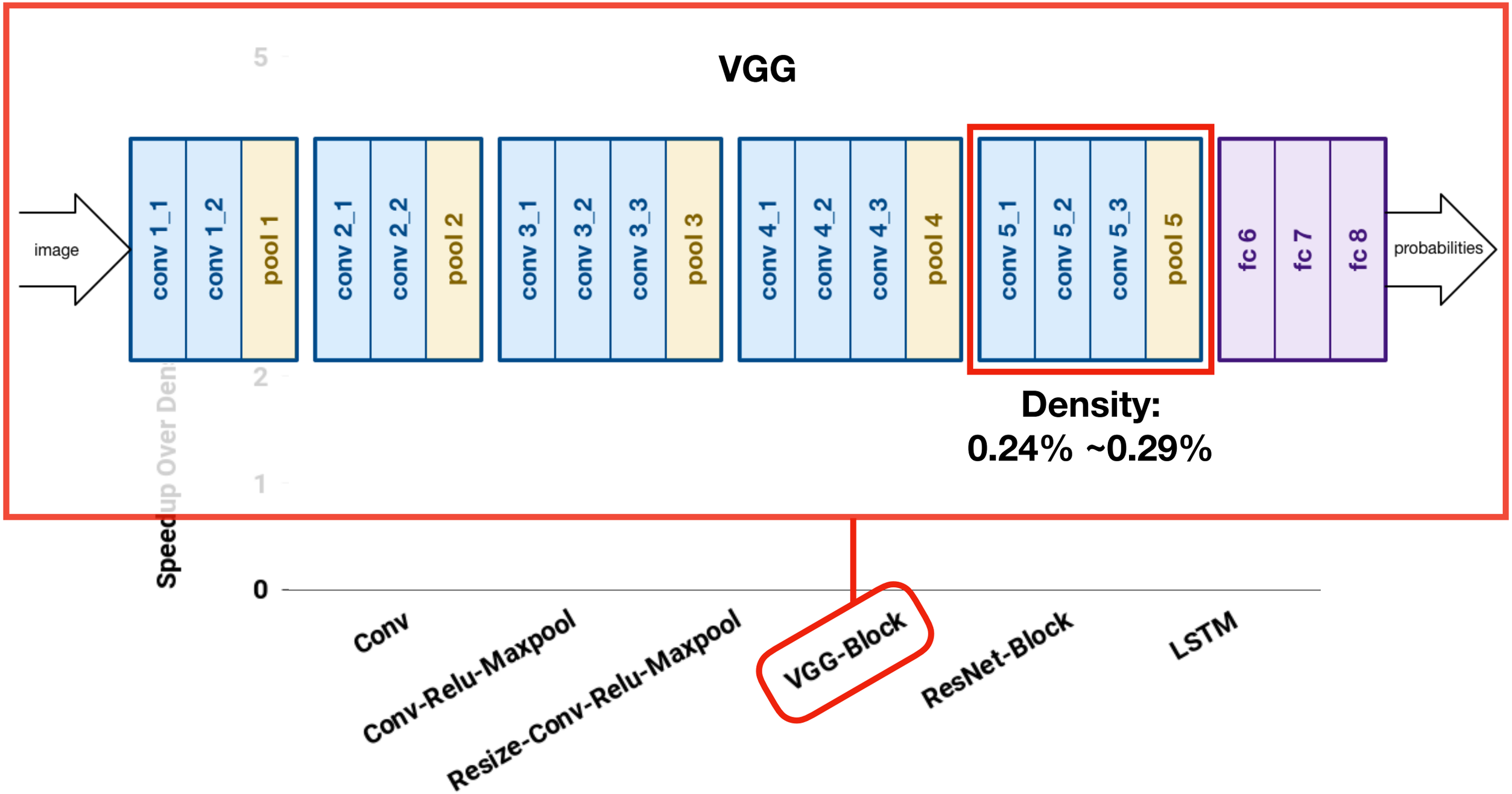
Tiramisu Dense and Sparse (CPU - 4 cores)



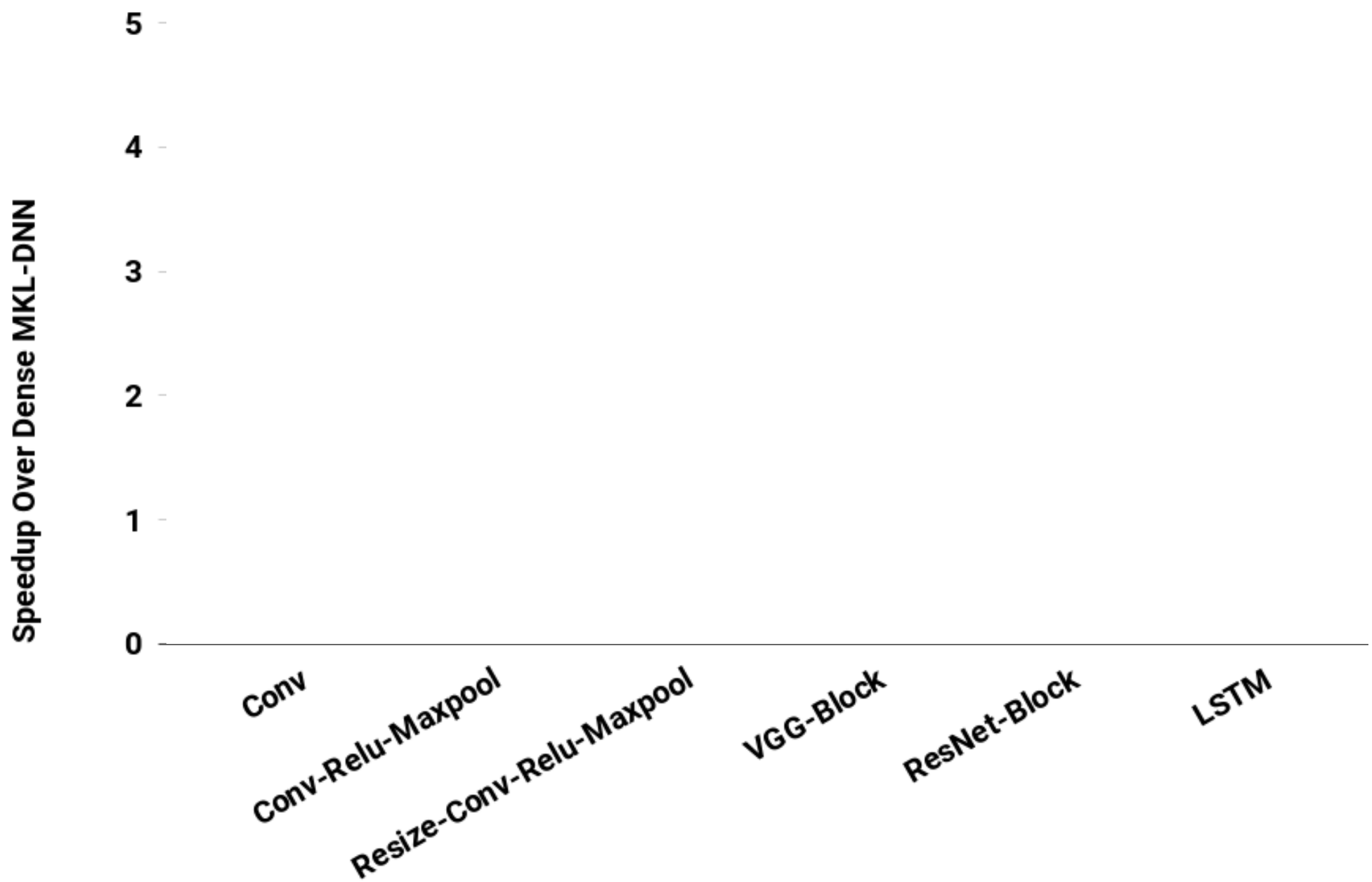
Tiramisu Dense and Sparse (CPU - 4 cores)



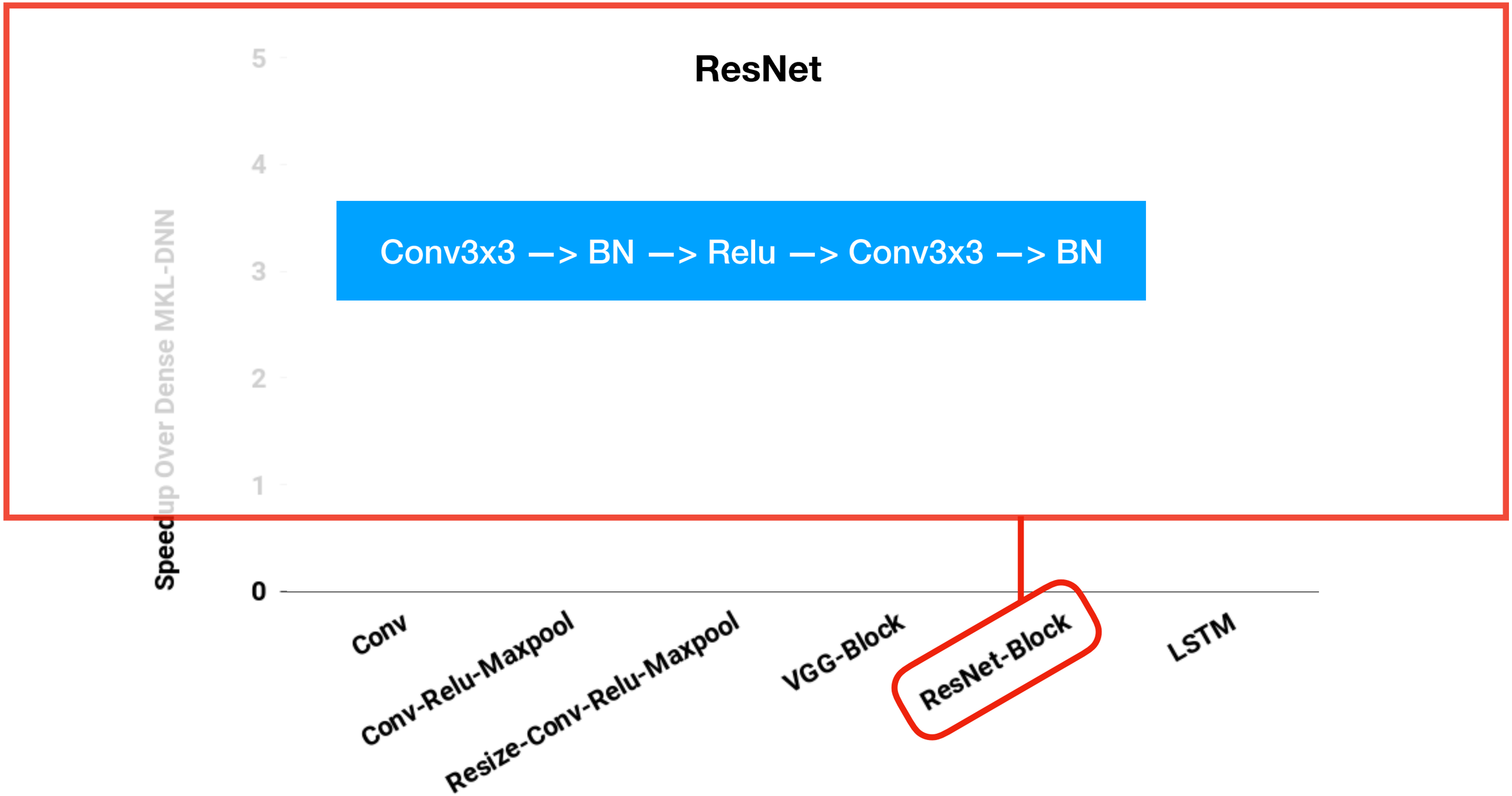
Tiramisu Dense and Sparse (CPU - 4 cores)



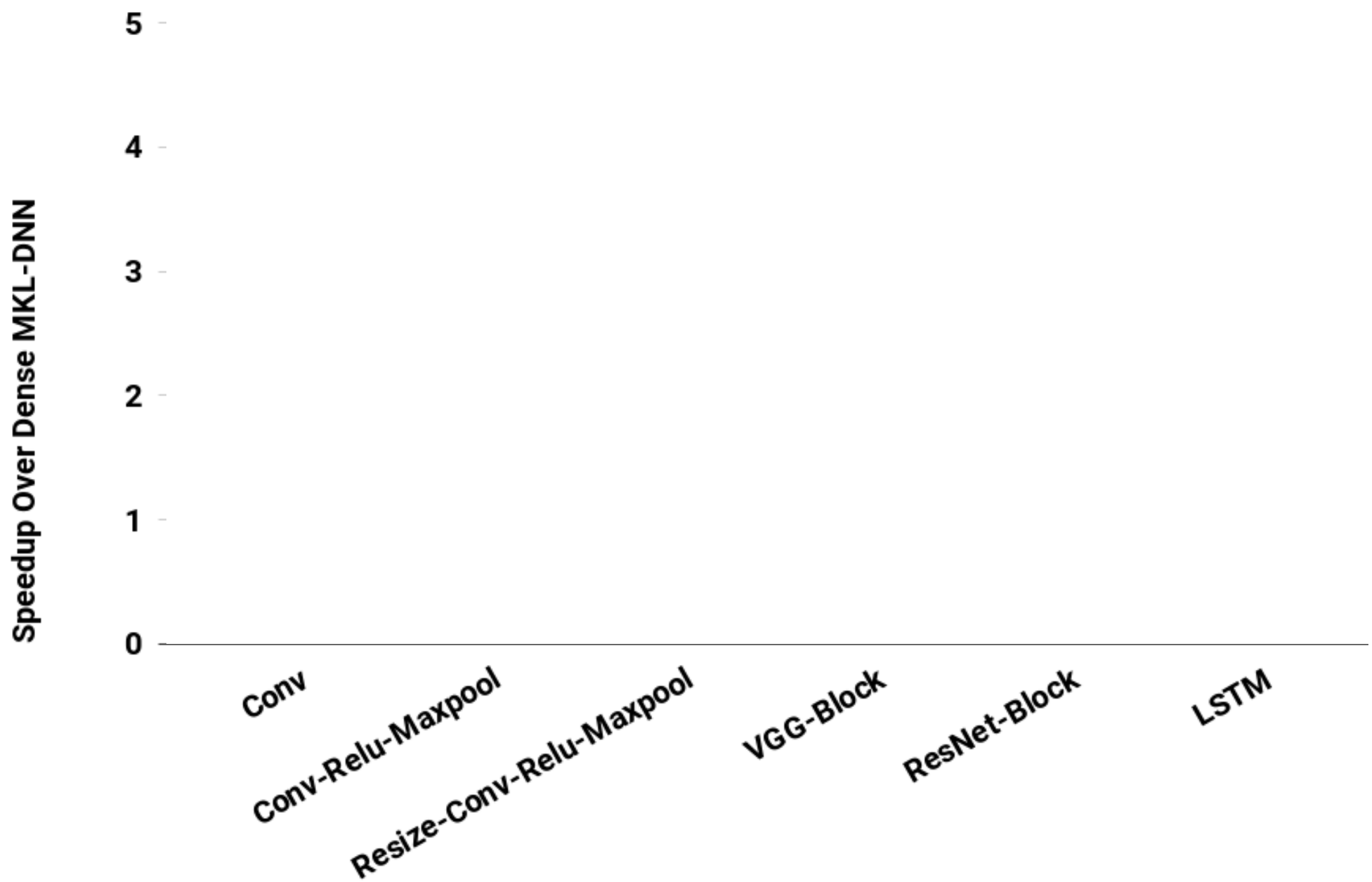
Tiramisu Dense and Sparse (CPU - 4 cores)



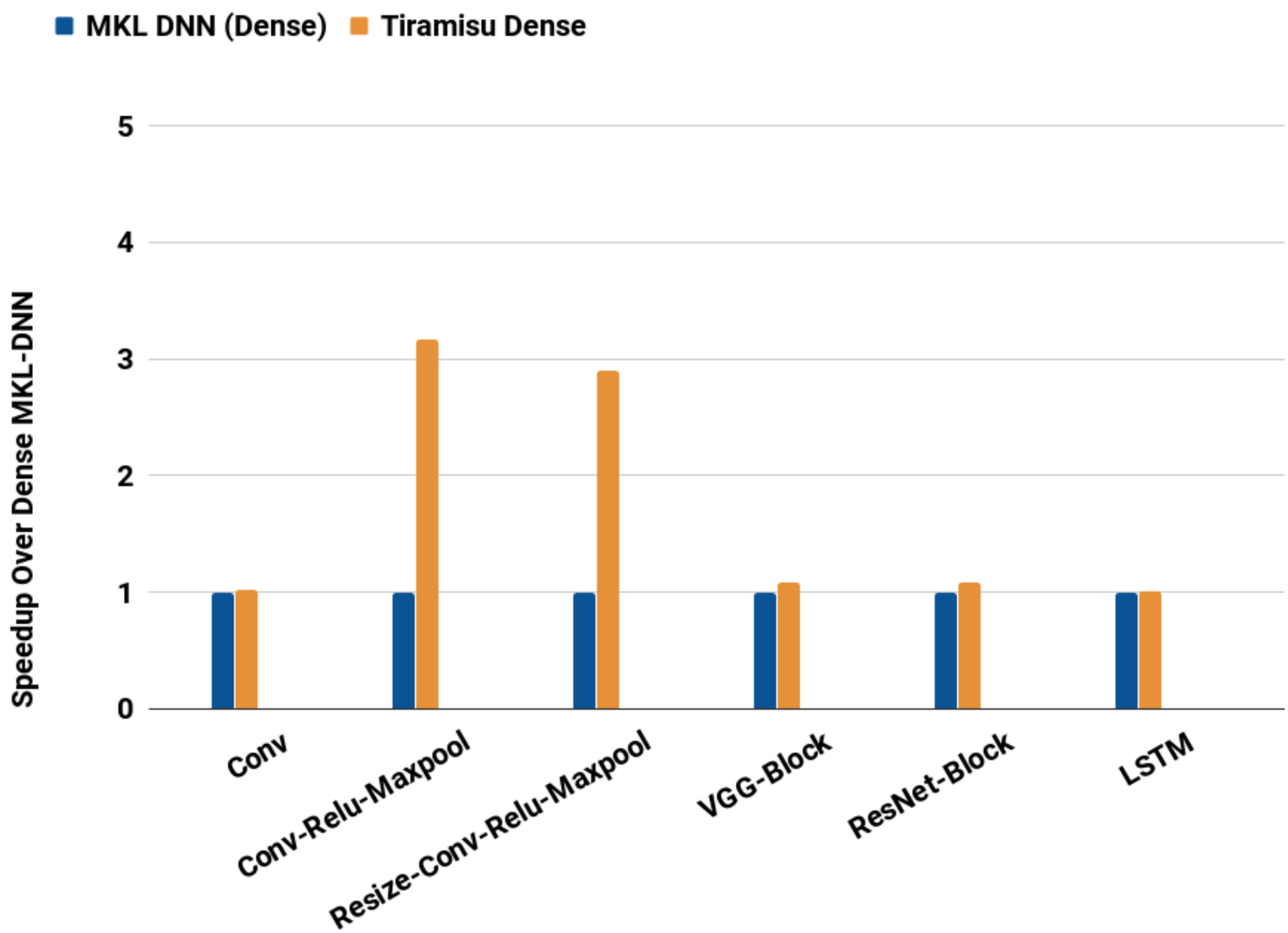
Tiramisu Dense and Sparse (CPU - 4 cores)



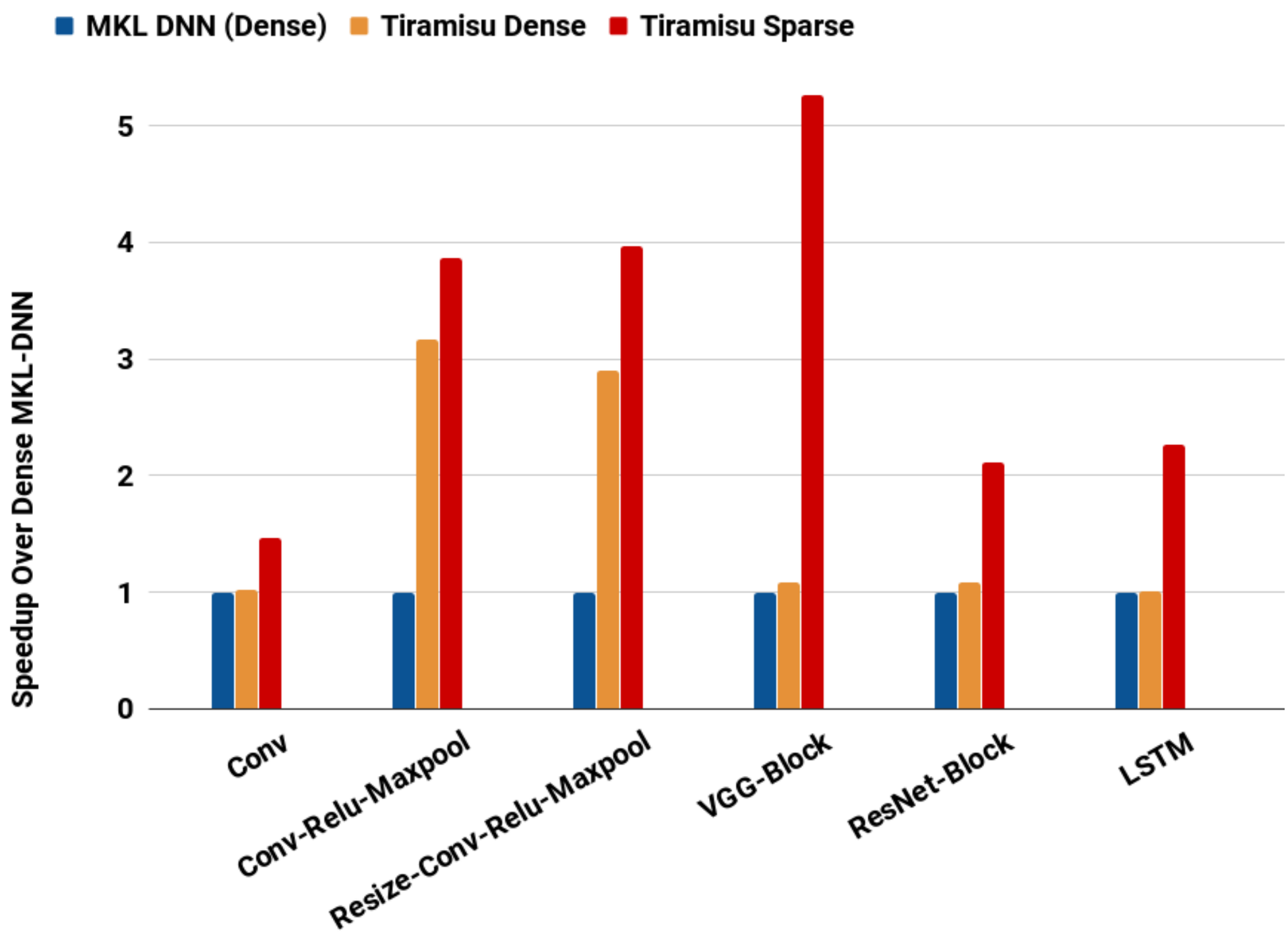
Tiramisu Dense and Sparse (CPU - 4 cores)



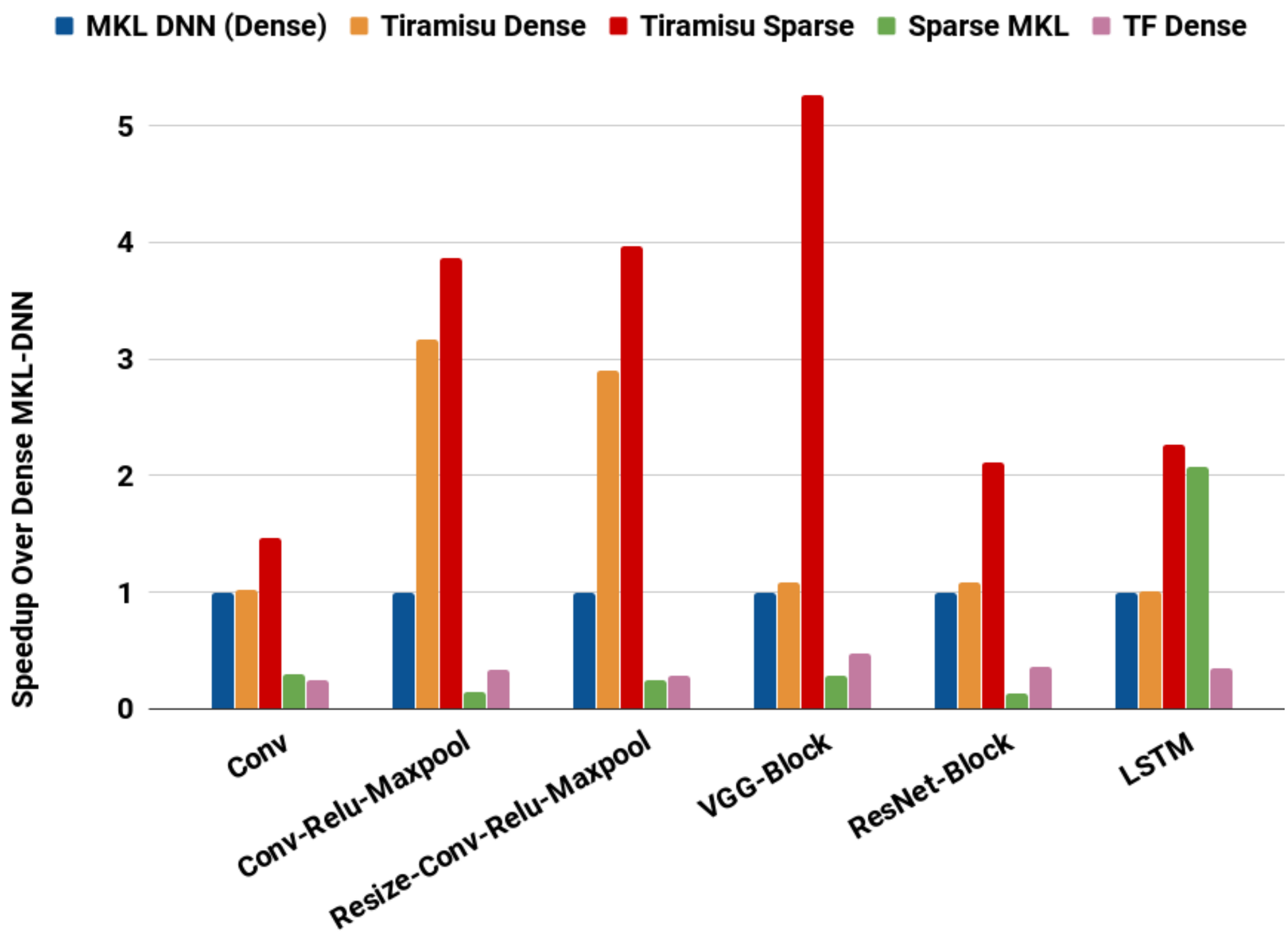
Tiramisu Dense and Sparse (CPU - 4 cores)



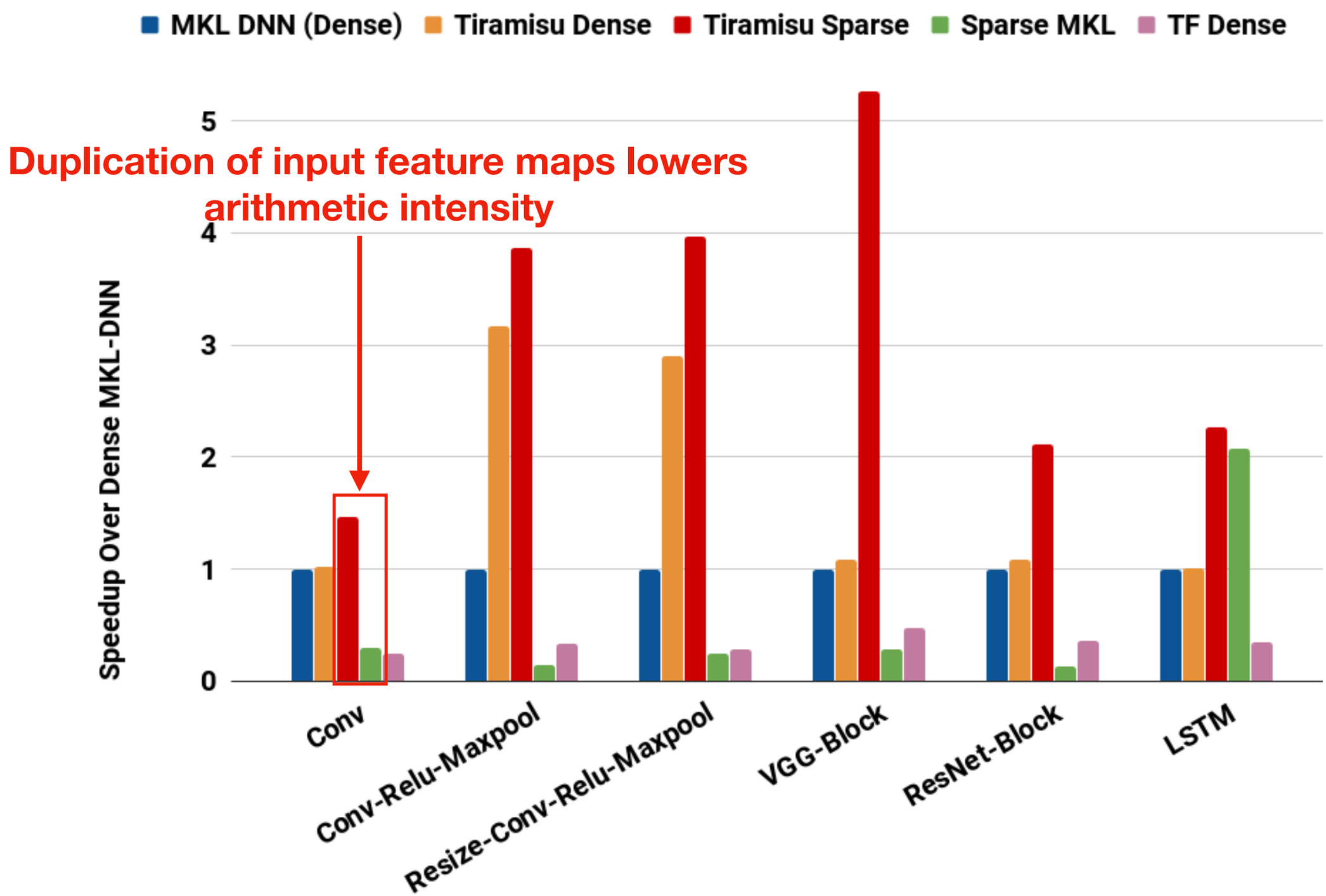
Tiramisu Dense and Sparse (CPU - 4 cores)



Tiramisu Dense and Sparse (CPU - 4 cores)



Tiramisu Dense and Sparse (CPU - 4 cores)



Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Outline

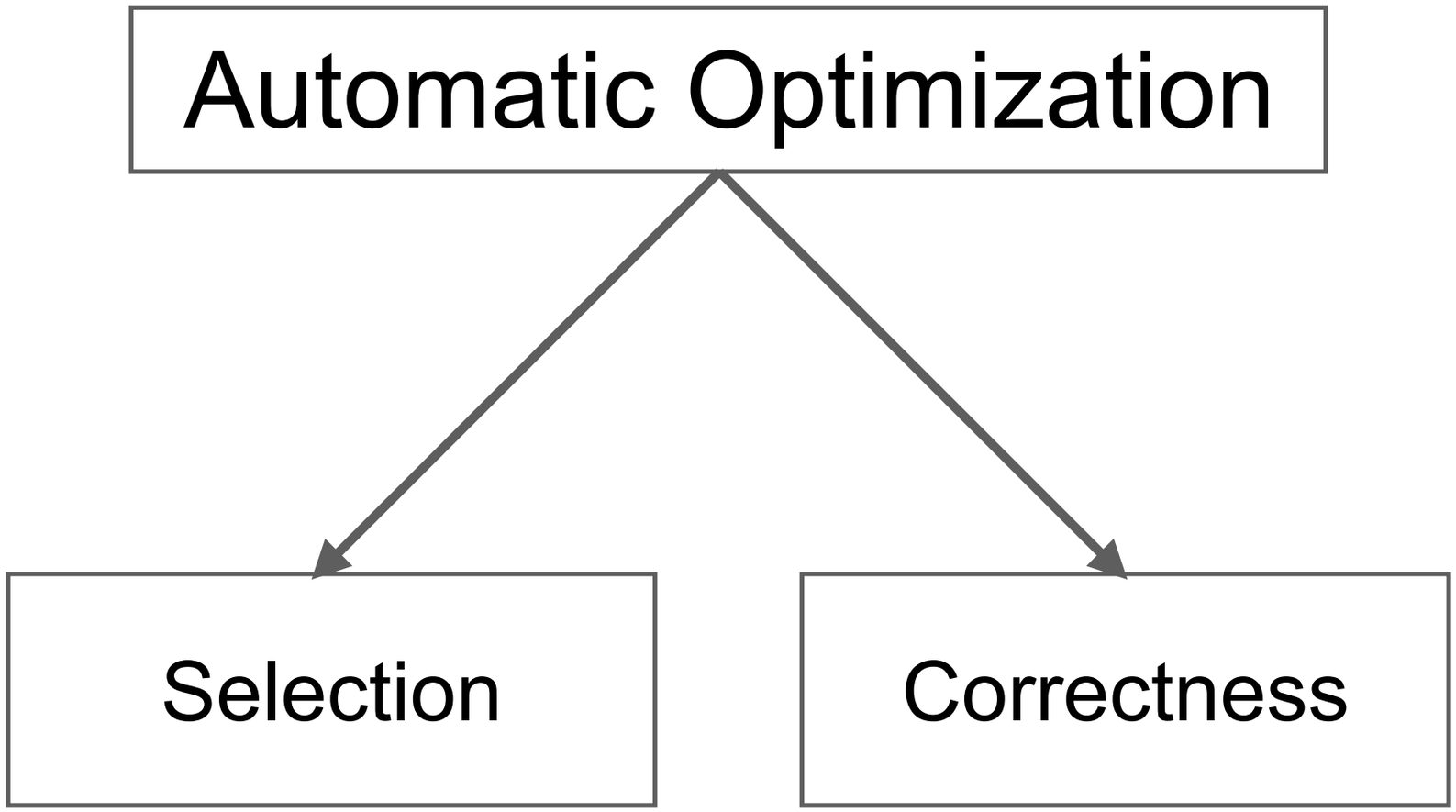
- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization

Outline

- **Phase I**
  - Tiramisu Overview
  - Example
  - RNNs
  - Sparse DNNs

- **Phase II**
  - Automatic Optimization


Automatic Optimization
Selection
Correctness

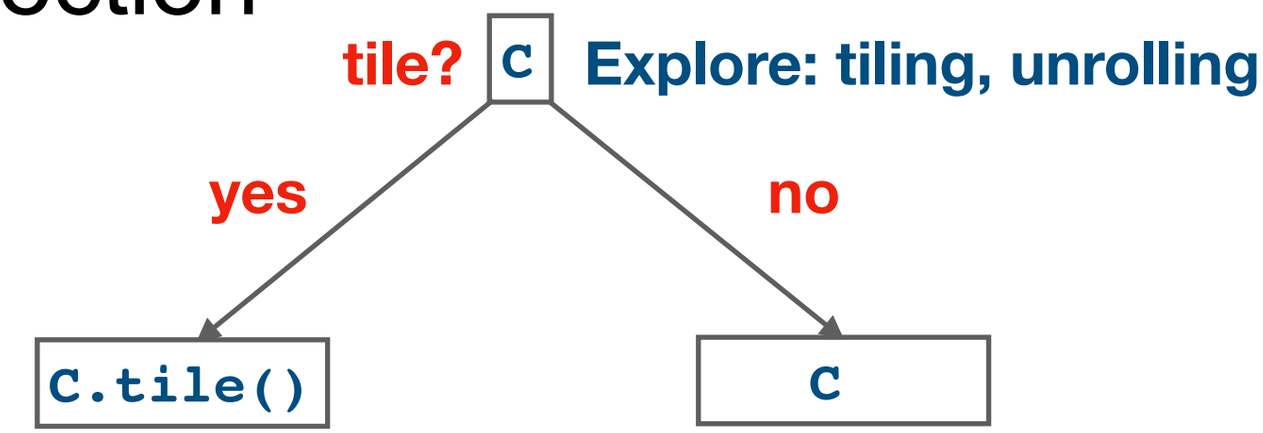
Optimization Selection

C **Explore: tiling, unrolling**

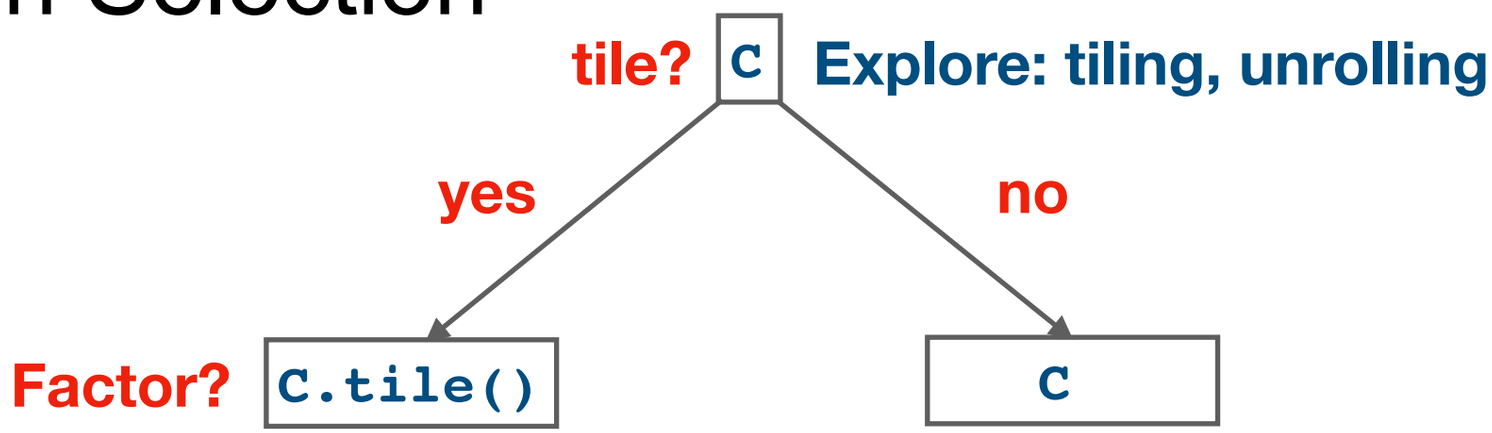
Optimization Selection

tile? C Explore: tiling, unrolling

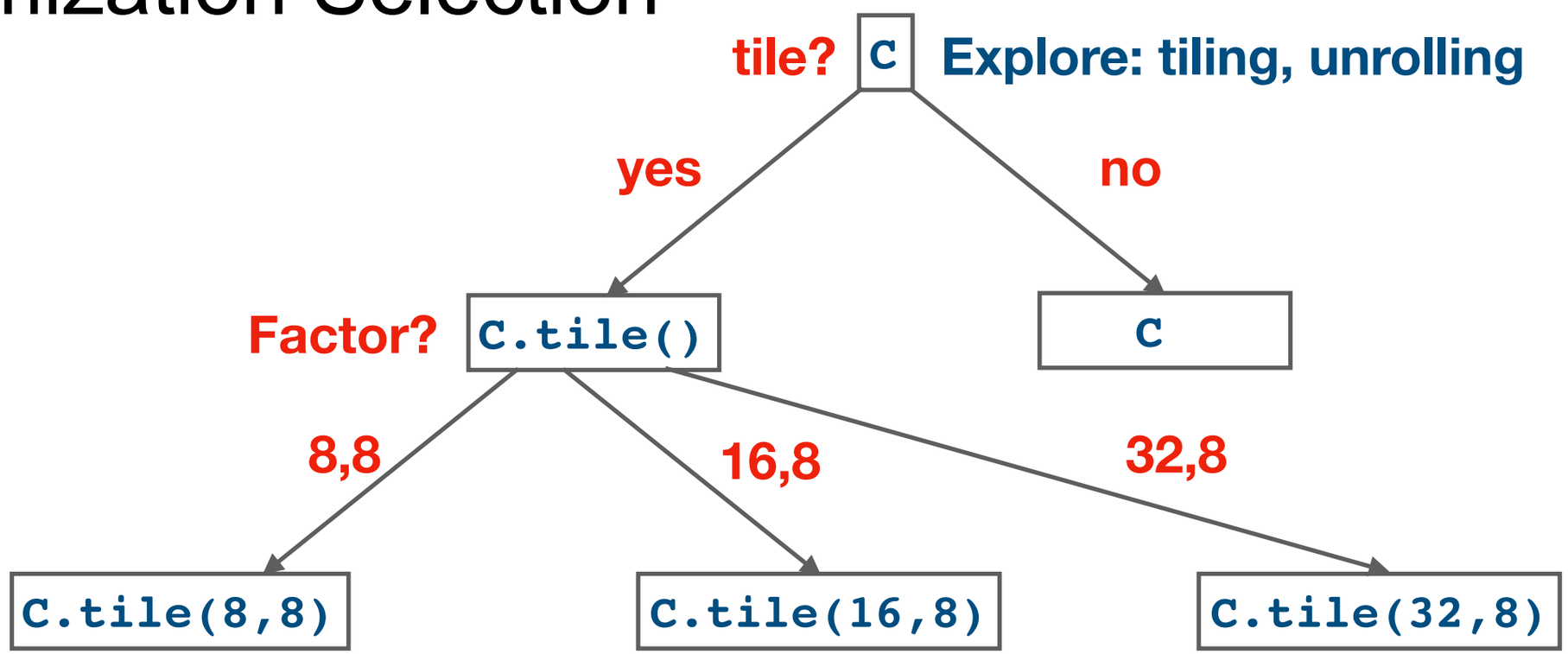
Optimization Selection



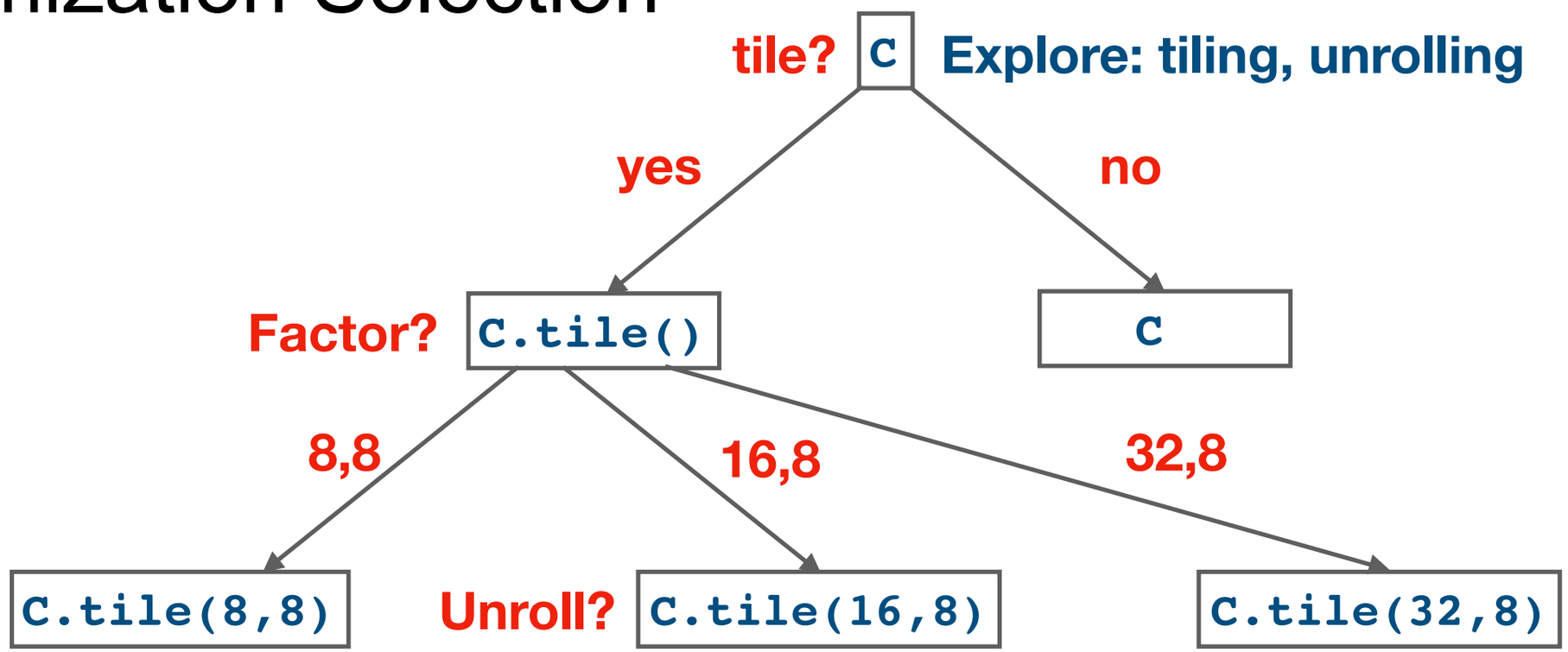
Optimization Selection



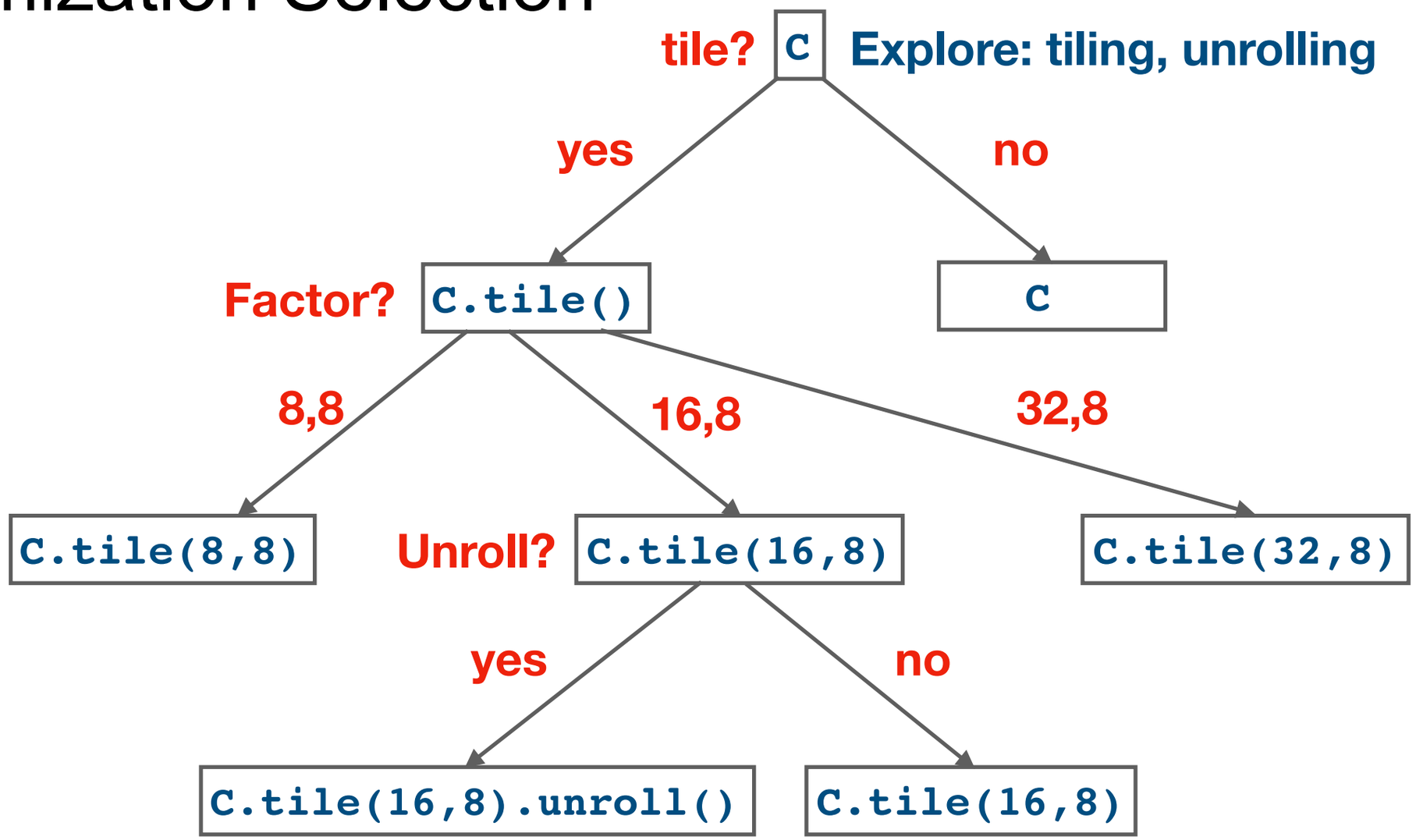
Optimization Selection



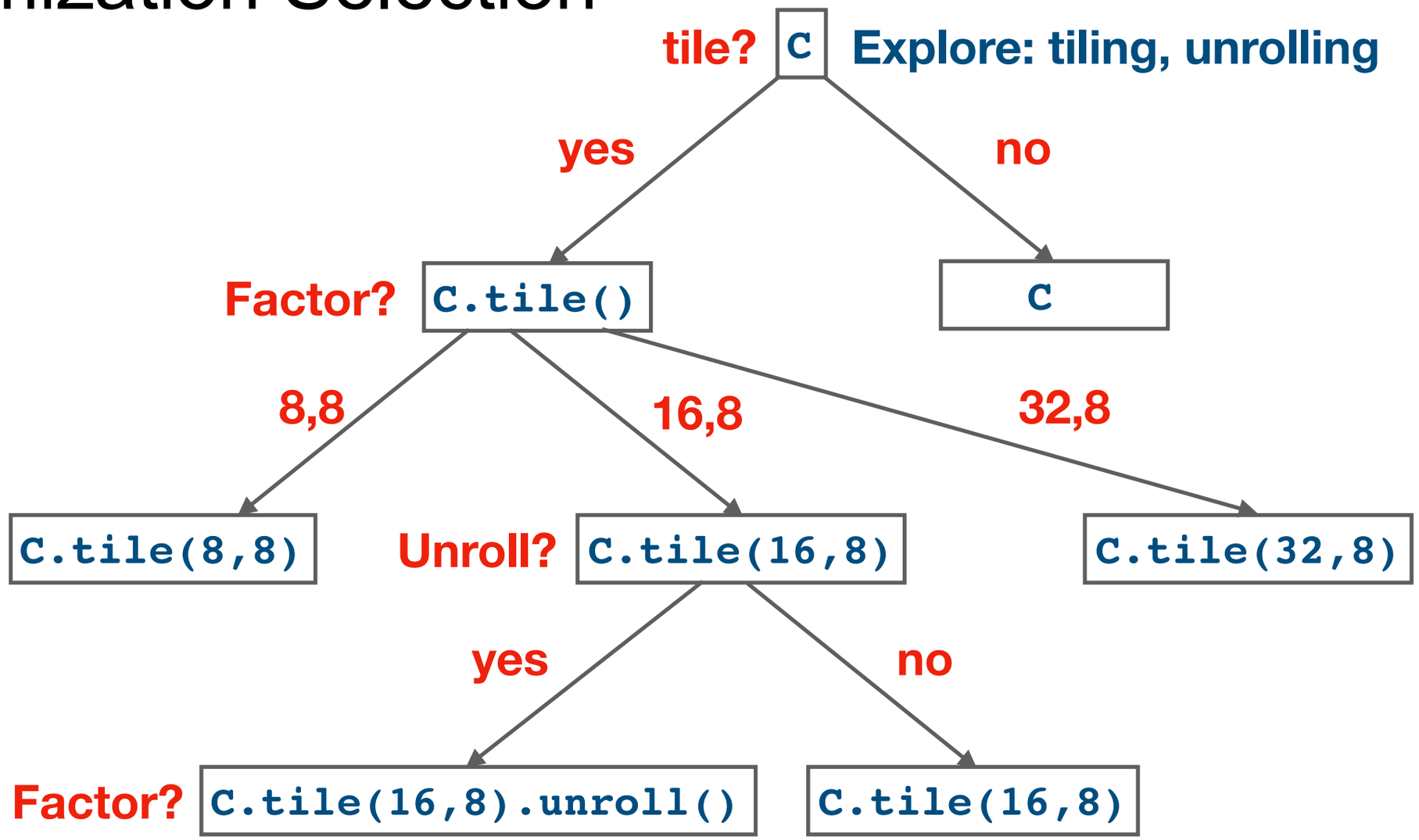
Optimization Selection



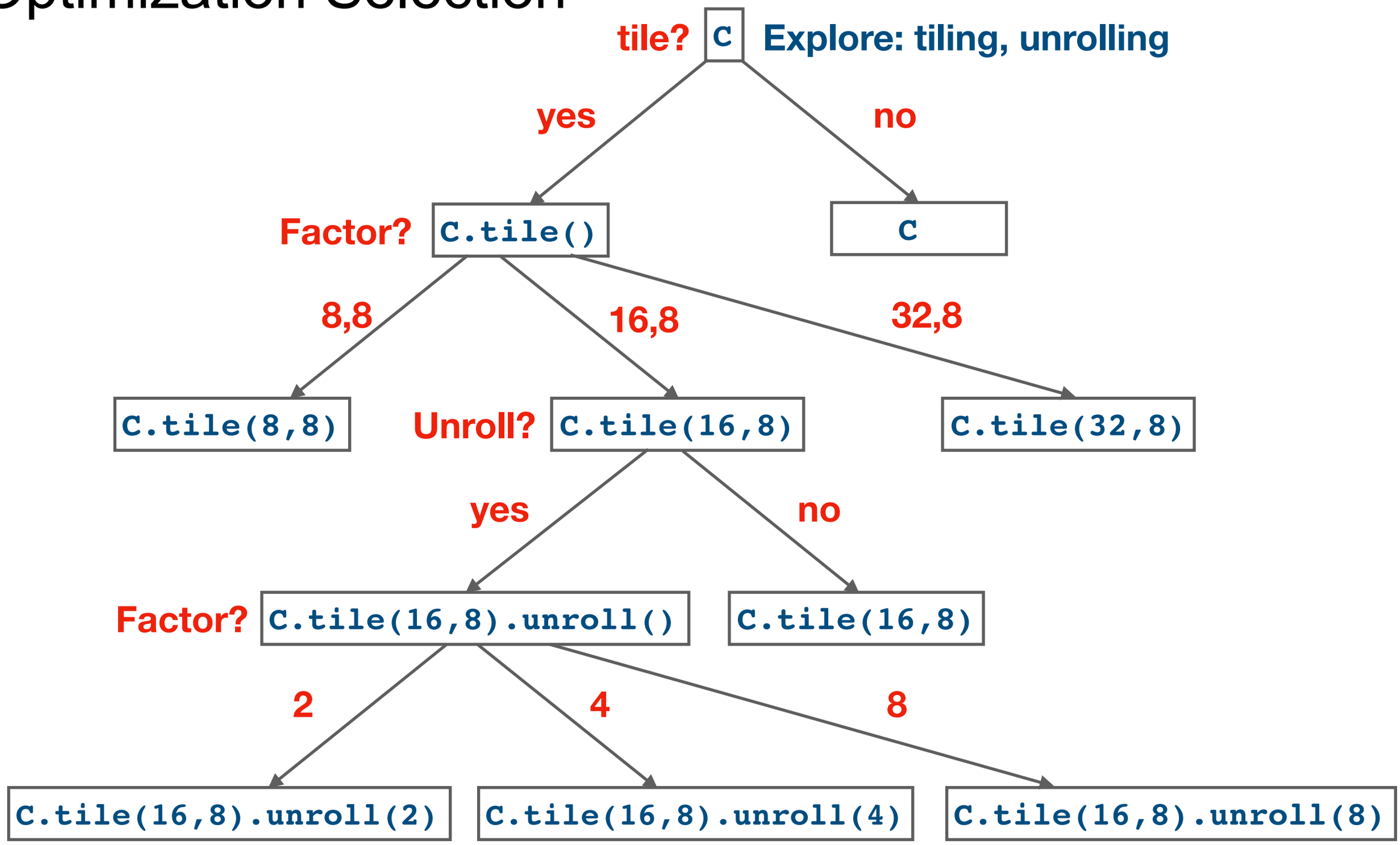
Optimization Selection

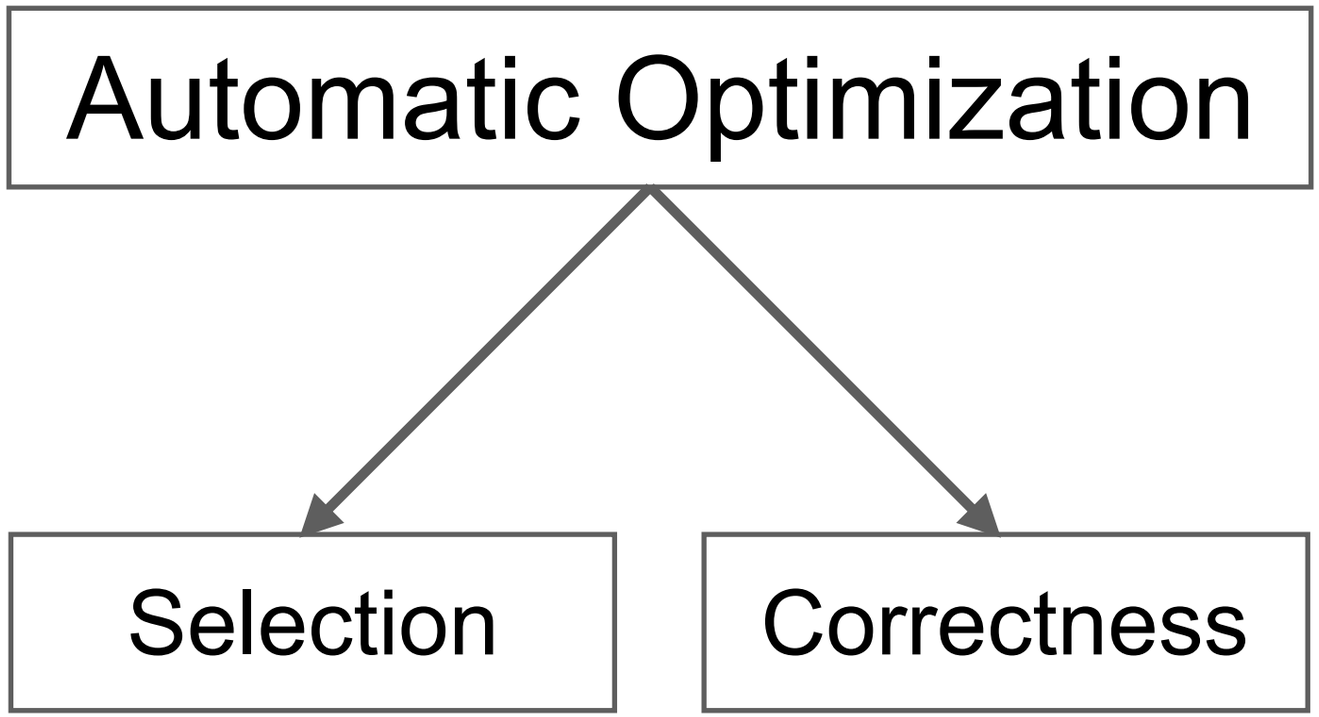


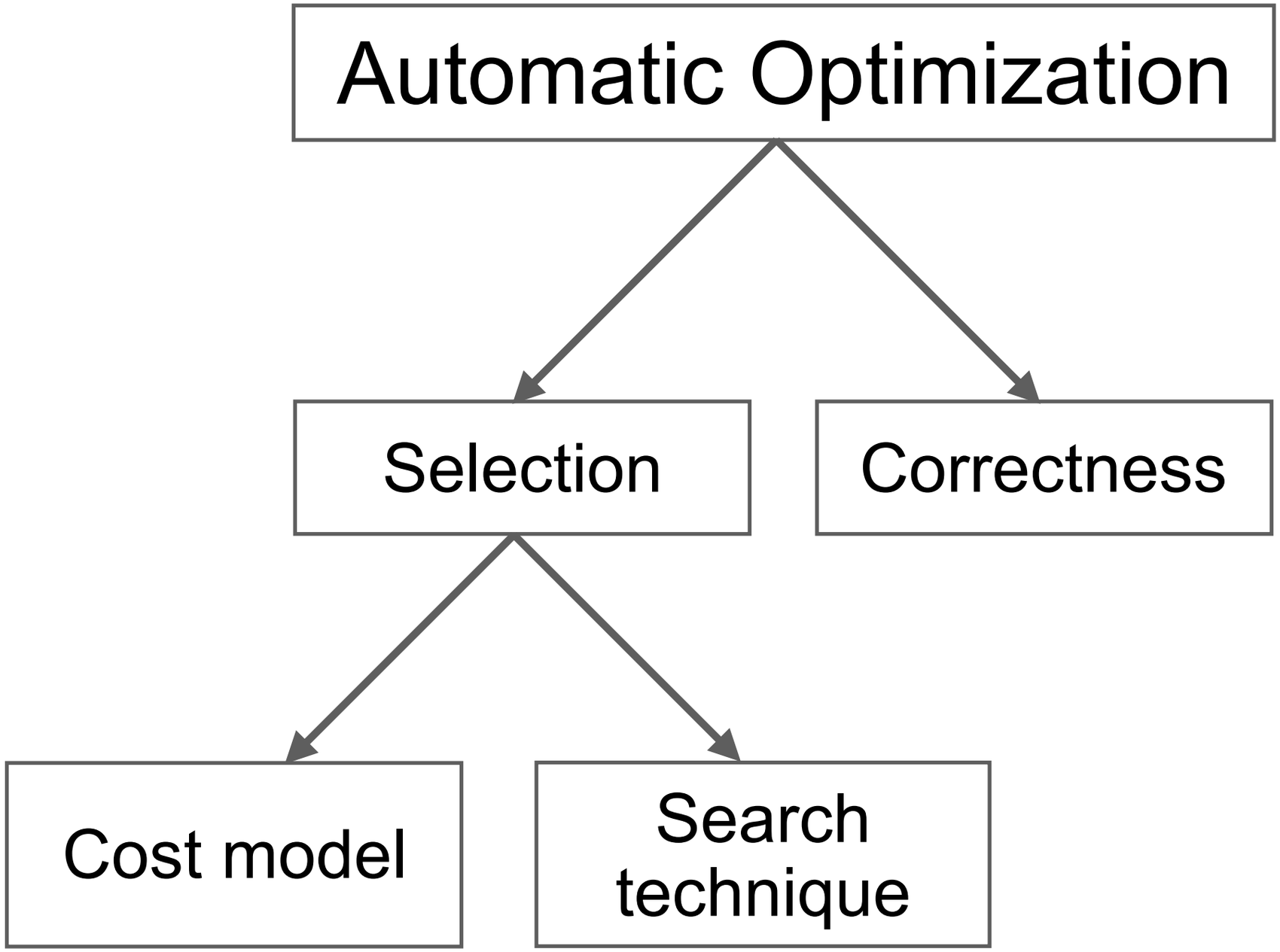
Optimization Selection

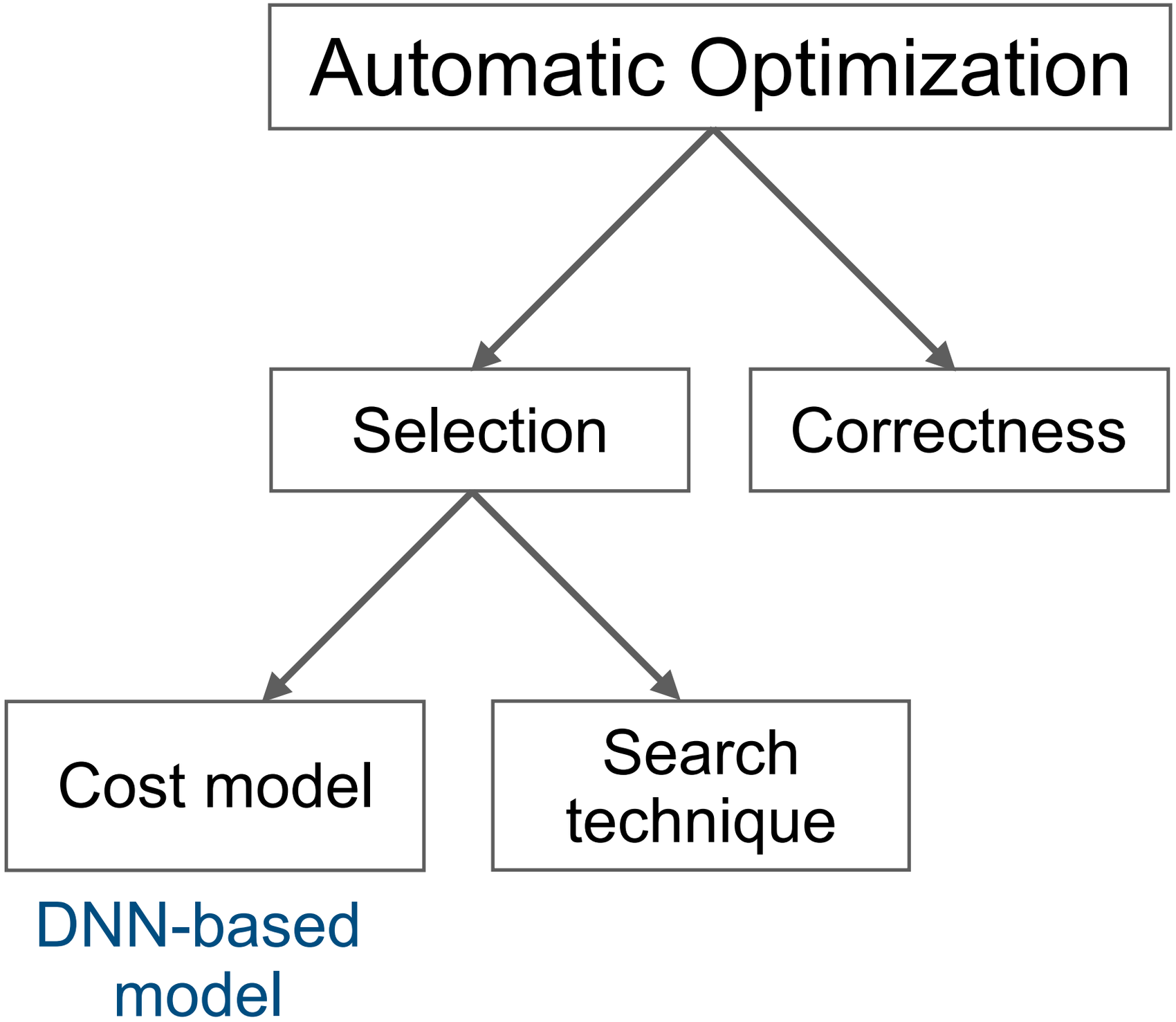


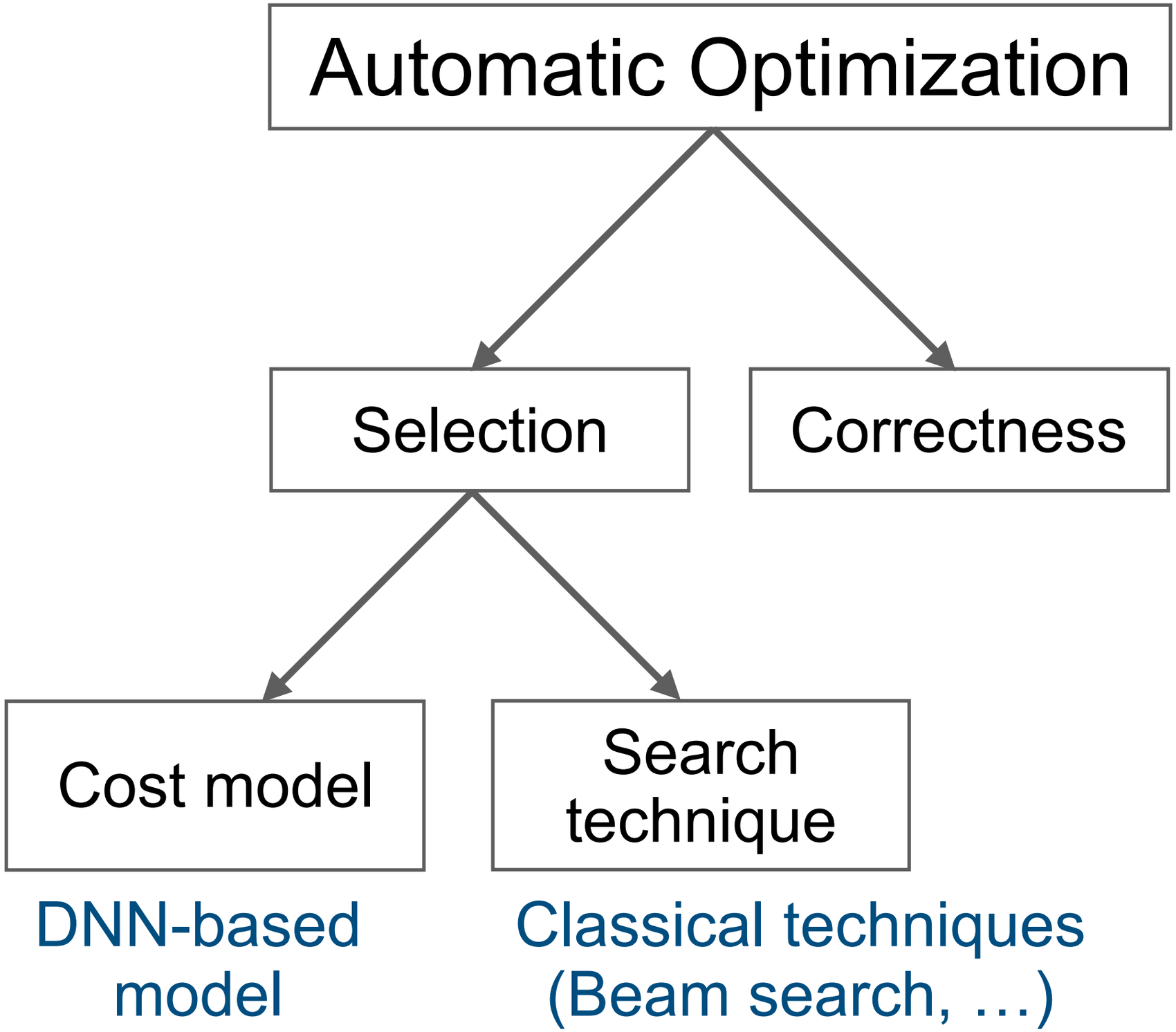
Optimization Selection

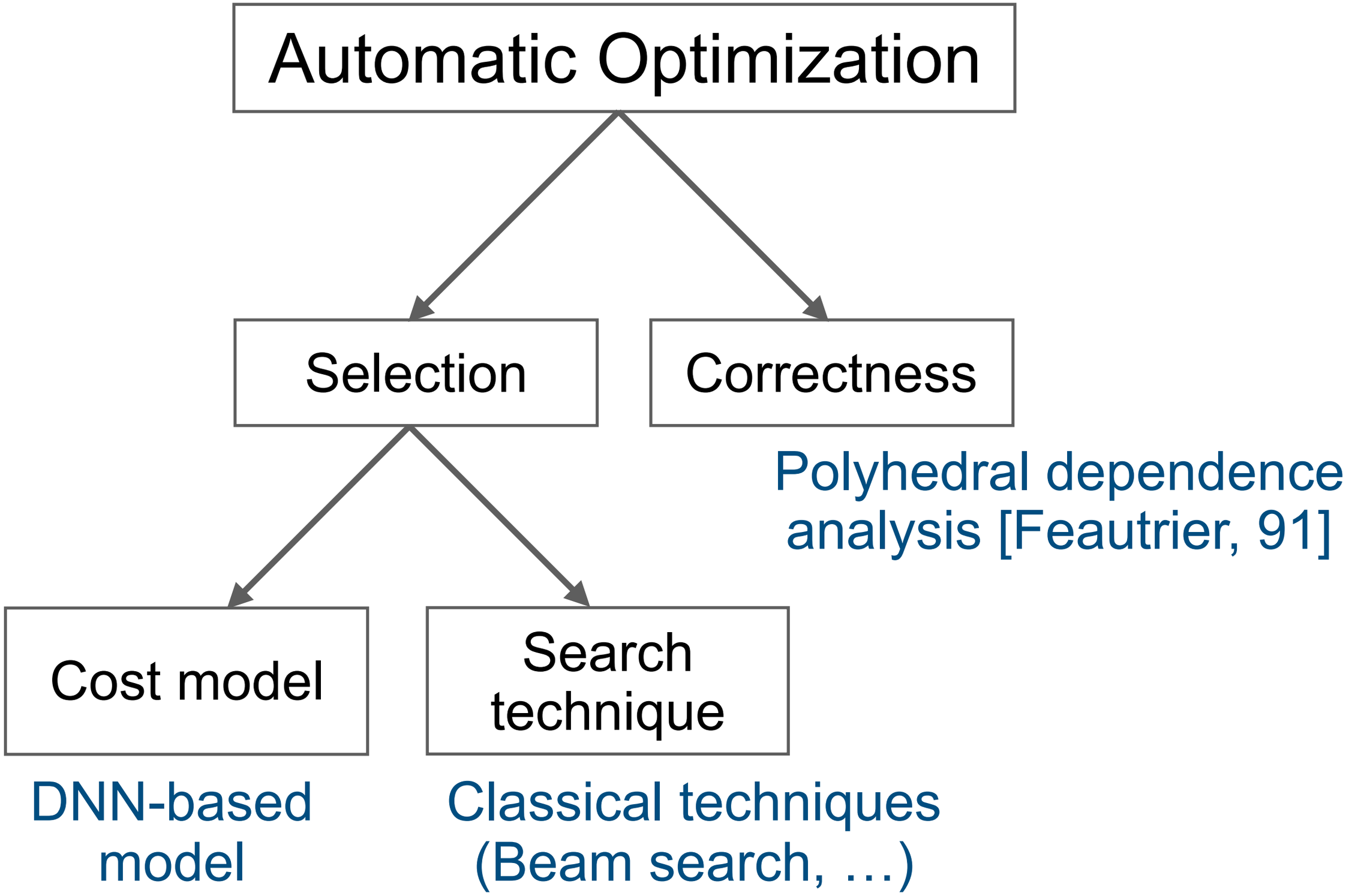


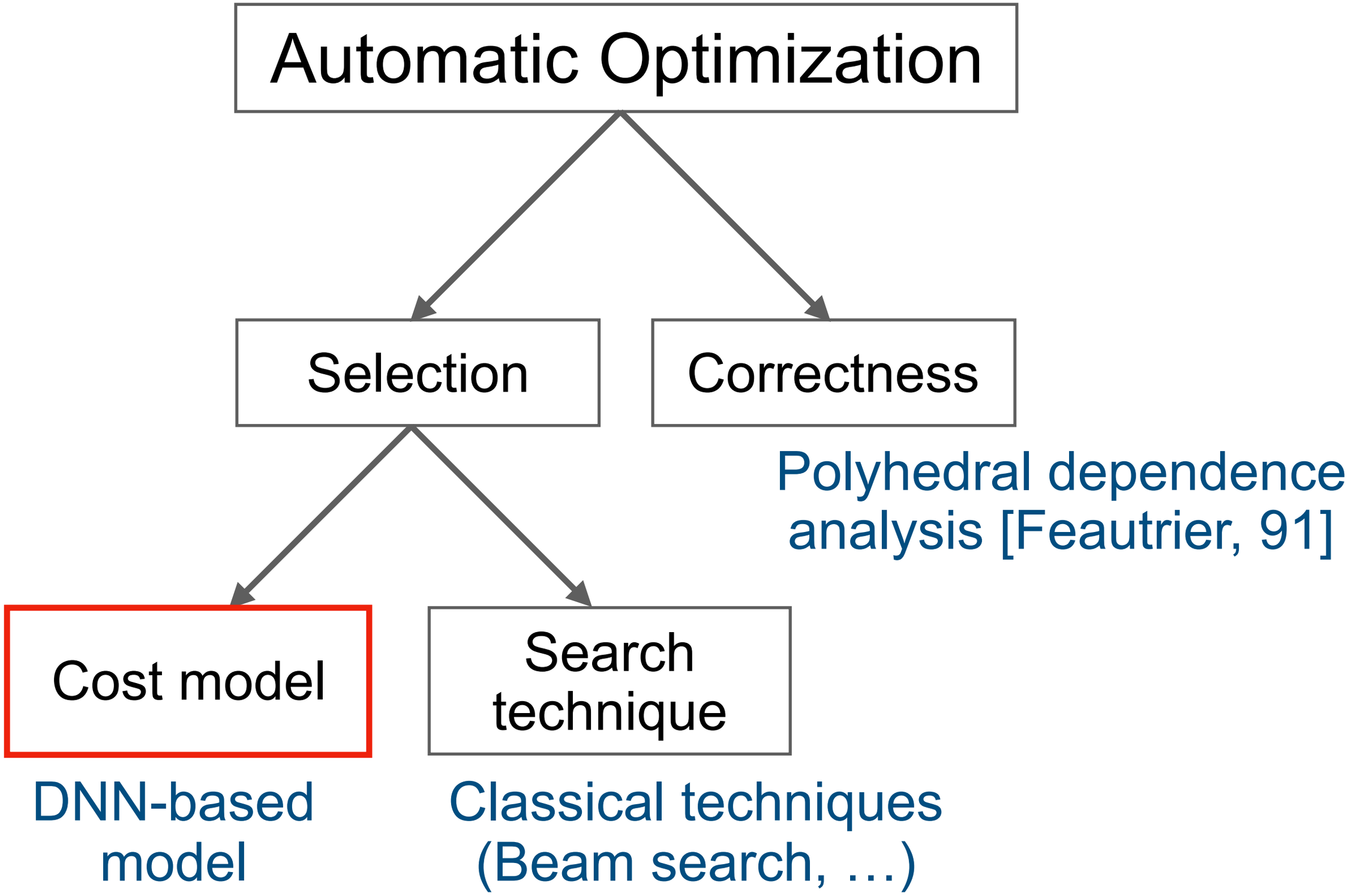

Automatic Optimization
Selection
Correctness


Automatic Optimization
Selection
Correctness
Cost model
Search technique

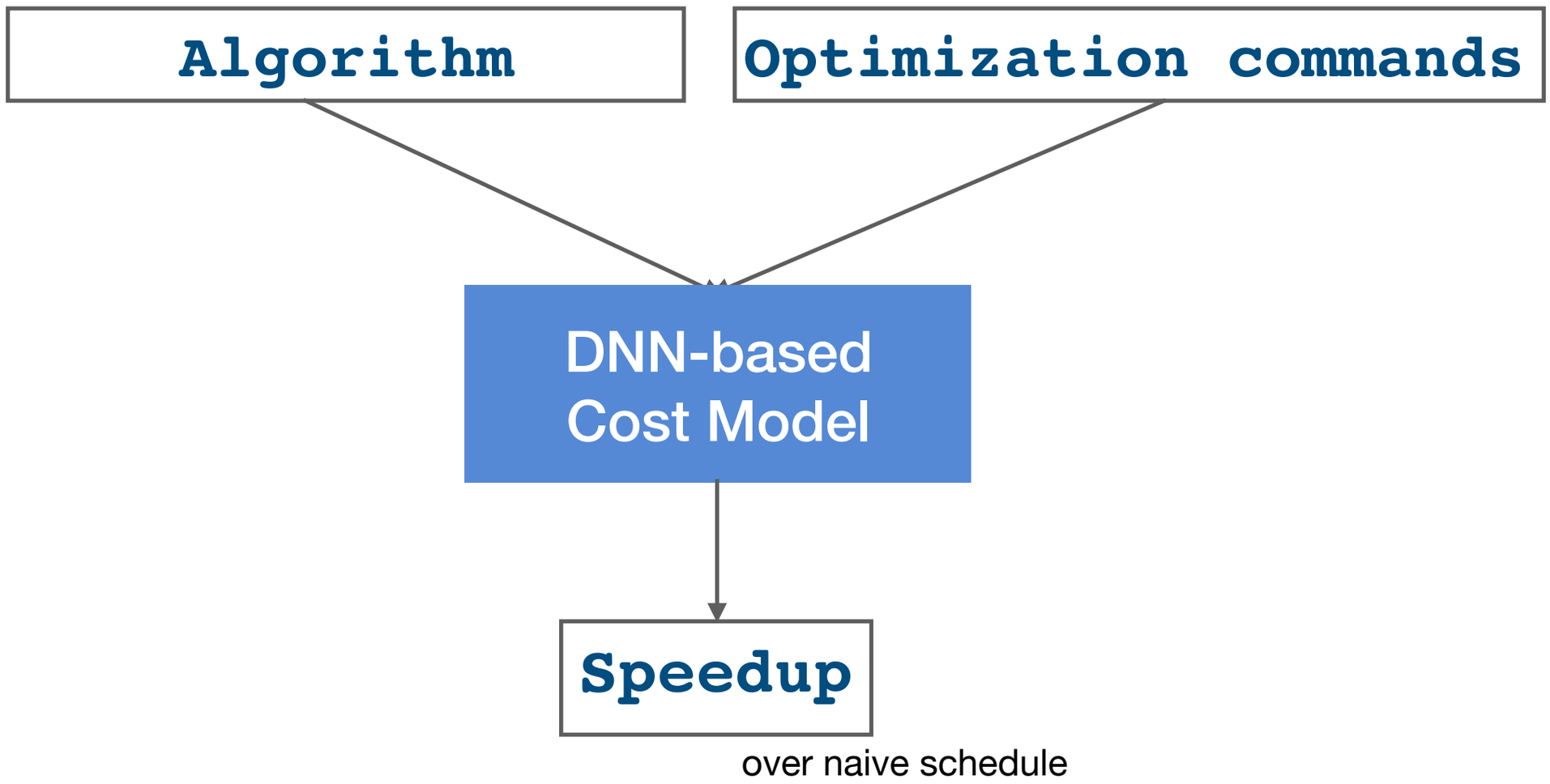

Automatic Optimization
Selection
Correctness
Cost model
Search technique
DNN-based model


Automatic Optimization
Selection
Correctness
Cost model
Search technique
DNN-based model
Classical techniques (Beam search, …)


Automatic Optimization
Selection
Correctness
Polyhedral dependence analysis [Feautrier, 91]
Cost model
Search technique
DNN-based model
Classical techniques (Beam search, …)


Automatic Optimization
Selection
Correctness
Polyhedral dependence analysis [Feautrier, 91]
Cost model
Search technique
DNN-based model
Classical techniques (Beam search, …)

DNN-based Cost Model



DNN-based Cost Model

DNN-based Cost Model

- Training data:
  - Generate random algorithms/schedules
  - Use patterns found in real code
  - Generate 1M data points

DNN-based Cost Model

- Training data:
  - Generate random algorithms/schedules
  - Use patterns found in real code
  - Generate 1M data points
- No feature engineering

DNN-based Cost Model

- Training data:
  - Generate random algorithms/schedules
  - Use patterns found in real code
  - Generate 1M data points
- No feature engineering

DNN-based Cost Model

- Training data:
  - Generate random algorithms/schedules
  - Use patterns found in real code
  - Generate 1M data points
- No feature engineering
- Error rates: 24.95%

Summary

Summary

- Tiramisu has 3 unique features:
  - Optimize sparse DNNs
  - Optimize RNNs
  - Generate code for distributed architectures

Summary

- Tiramisu has 3 unique features:
  - Optimize sparse DNNs
  - Optimize RNNs
  - Generate code for distributed architectures

- Two main design choices
  - Polyhedral representation
  - Separates algorithm from optimization commands

Summary

- Tiramisu has 3 unique features:
  - Optimize sparse DNNs
  - Optimize RNNs
  - Generate code for distributed architectures

- Two main design choices
  - Polyhedral representation
  - Separates algorithm from optimization commands

- Uses deep learning to build a cost model for automatic optimization

Summary

- Tiramisu has 3 unique features:
  - Optimize sparse DNNs
  - Optimize RNNs
  - Generate code for distributed architectures

- Two main design choices
  - Polyhedral representation
  - Separates algorithm from optimization commands

- Uses deep learning to build a cost model for automatic optimization

- Web: http://tiramisu-compiler.org

Summary

- Tiramisu has 3 unique features:
  - Optimize sparse DNNs
  - Optimize RNNs
  - Generate code for distributed architectures

- Two main design choices
  - Polyhedral representation
  - Separates algorithm from optimization commands

- Uses deep learning to build a cost model for automatic optimization

- Web: http://tiramisu-compiler.org

- Interested in building a community around Tiramisu