



<https://hao-ai-lab.github.io/dsc291-s24/>

DSC 291: ML Systems Spring 2024

LLMs

Parallelization

Single-device Optimization

Basics

Which of the following is false?

- A. Forward mode autodiff and backward mode autodiff give the same results
- B. Forward mode autodiff is slower than backward mode autodiff for common ML tasks
- C. Backward mode autodiff is adopted by ML tasks since the dimension of the output is usually smaller than that of the input
- D. Forward model autodiff could be efficient when n is very small and k is large in the following equation

$$f : R^n \rightarrow R^k$$

What is a “kernel” in the context of GPUs?

- A. A specific section of the CPU used for memory operations.
- B. A specific section of the GPU used for memory operations.
- C. A type of thread that operates on the GPU.
- D. A function that is executed simultaneously by tens of thousands of threads on GPU cores.

What type of parallelism is natural for GPU?

- A. Task Parallelism
- B. Data Parallelism
- C. Both data and task parallelism
- D. Neither data nor task parallelism

What is the significance of over-subscribing the GPU?

- A. It reduces the overall performance of the GPU.
- B. It ensures that there are more blocks than SMPs present on the device, helping to hide latencies and ensure high occupancy of the GPU.
- C. It leads to a memory overflow in the GPU.
- D. It ensures that there are more SMPs than blocks present on the device.

What is the arithmetic intensity for the following function:

```
# A, B are 2-D matrices of shape [2,2]
func(matrix A, matrix B):
    Load A
    Load B
    C = matmul(A,B)
```

- A. 0.334
- B. 2
- C. 1
- D. 1.5

Which of the following Tensor manipulations cannot benefit from strided representation

- A. Broadcast_to
- B. Slice
- C. Reshape
- D. Permute dimensions
- E. Transpose
- F. Continuous
- G. indexing like `t[:, 1:5]`

If we have tensor of shape $[2,9,1]$ stored contiguous in memory following row Major, what is its strides?

A. $(9,1,1)$

B. $(2,9,1)$

C. $(1,9,2)$

D. $(9,9,9)$

Which of the following about GPUs is false?

- A. SIMD requires all ALUs/Core to proceed in the same pace
- B. Threads are the smallest units to process a chunk of data on GPU
- C. Grid has a 3D shape like gridshape = (x, y, z)
- D. GPU has tons of ALUs that are specialized for massive data parallelism

Which of the following is True for Cache Tiling in Matmul

- A. It saves memory allocated in Cache
- B. It reduce the memory movement between Cache to Register
- C. It reuses memory movement between Dram and Cache
- D. It increases arithmetic intensity because it makes the computational faster

Which of the following is True about GPU Memory

- A. On H100, a CPU process can access an array stored on H100 GPU memory
- B. A thread in a threadblock can access its threadblock-level shared memory
- C. Pinned memory is a part of memory allocated on GPU
- D. `print(a)` function in C++ can print an array allocated via `a = cudaMalloc(..)`

GPU and CUDA

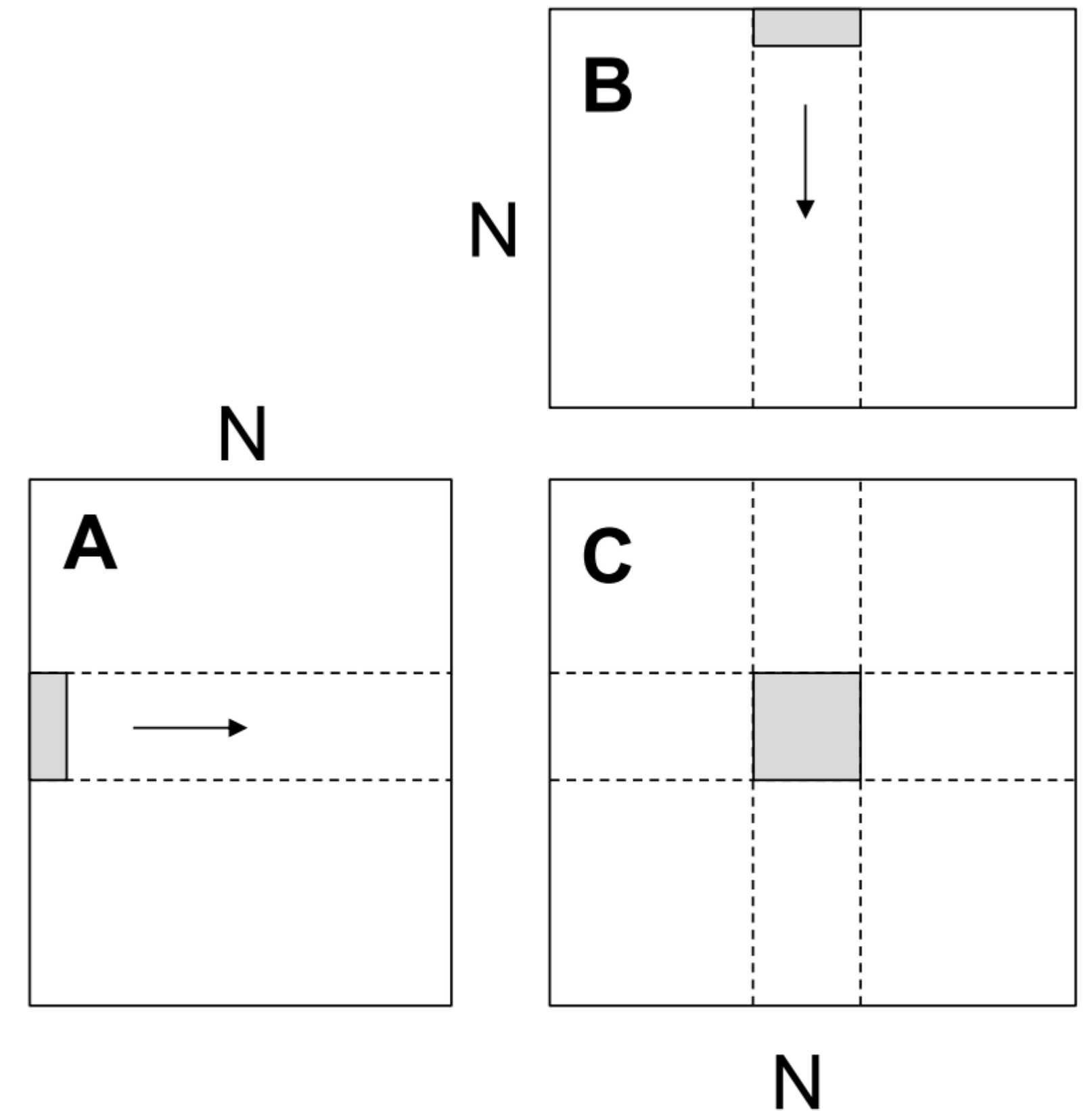
- Basic concepts and Architecture
 - Concepts
 - Execution Model
 - Memory
- Programming abstraction
- **Case study: Matmul**

Case study: GPU Matmul

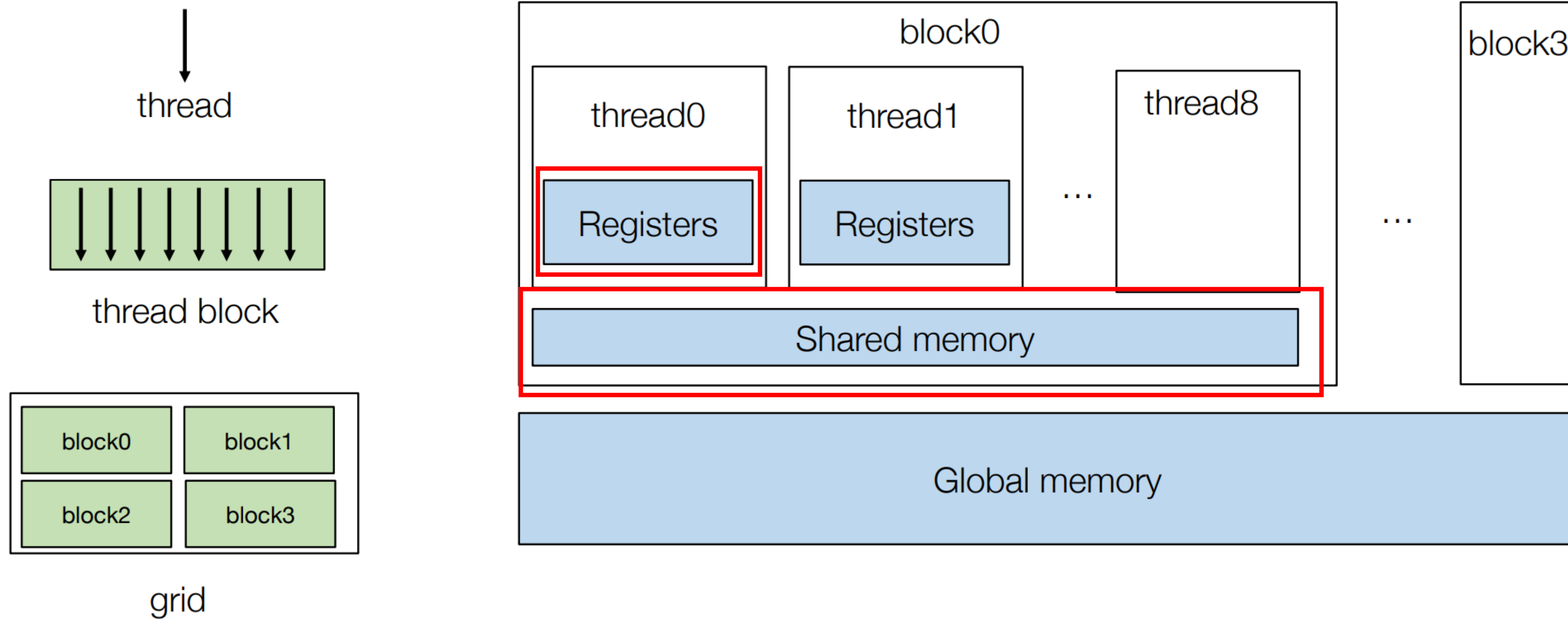
- Strawman solution:
 - $C = A \times B$
 - Each thread computes one element

```
int N = 1024;  
dim3 threadsPerBlock(32, 32, 1);  
dim3 numBlocks(N/32, N/32, 1);  
  
matmul<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {  
    int x = blockIdx.x * blockDim.x + threadIdx.x;  
    int y = blockIdx.y * blockDim.y + threadIdx.y;  
  
    result = 0;  
    for (int k = 0; k < N; ++k) {  
        result += A[x][k] * B[k][y];  
    }  
    C[x][y] = result;  
}
```



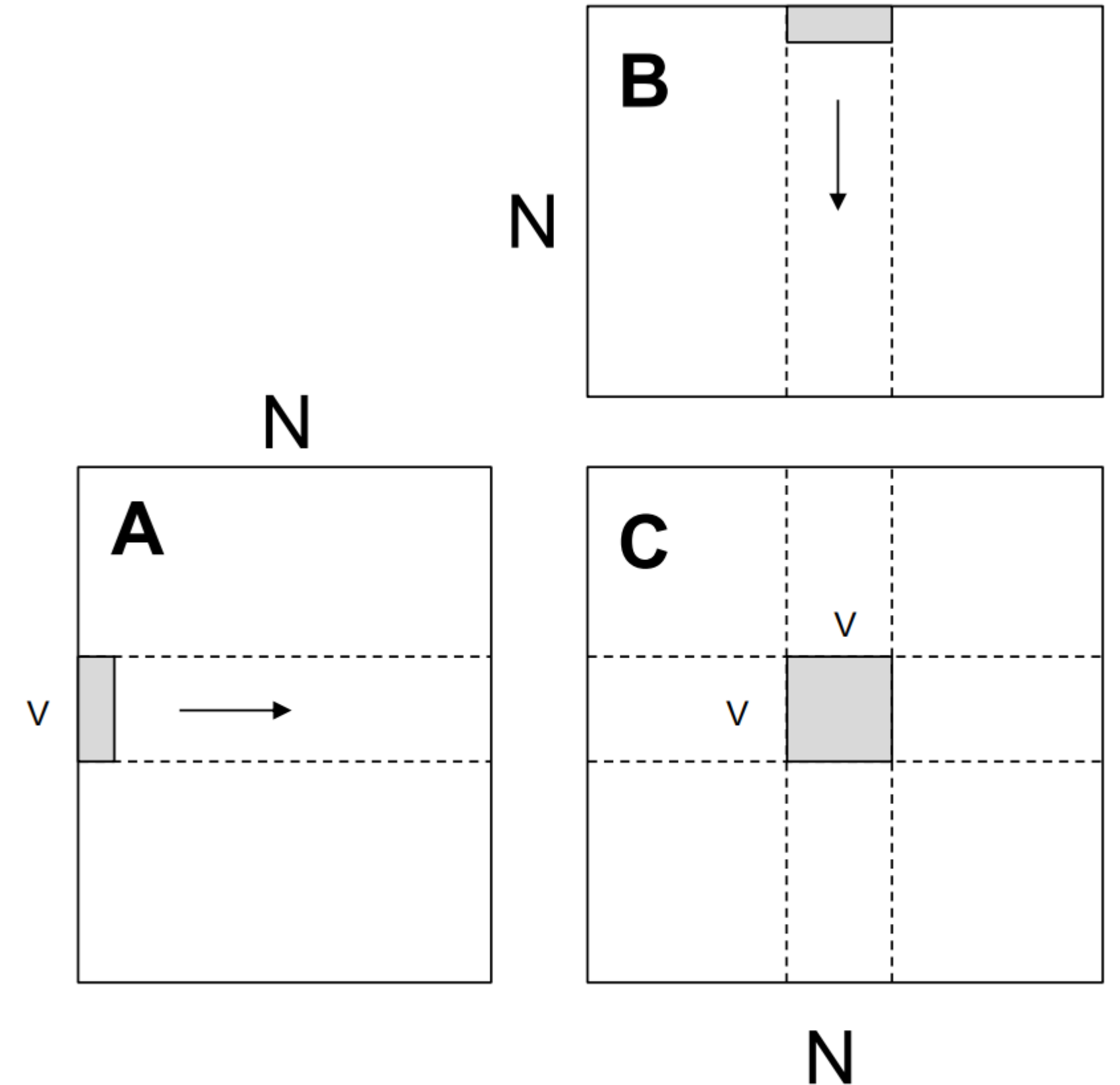
High-level Opt Idea: Recall Memory Hierarchy



Recall register tiling -> thread tiling

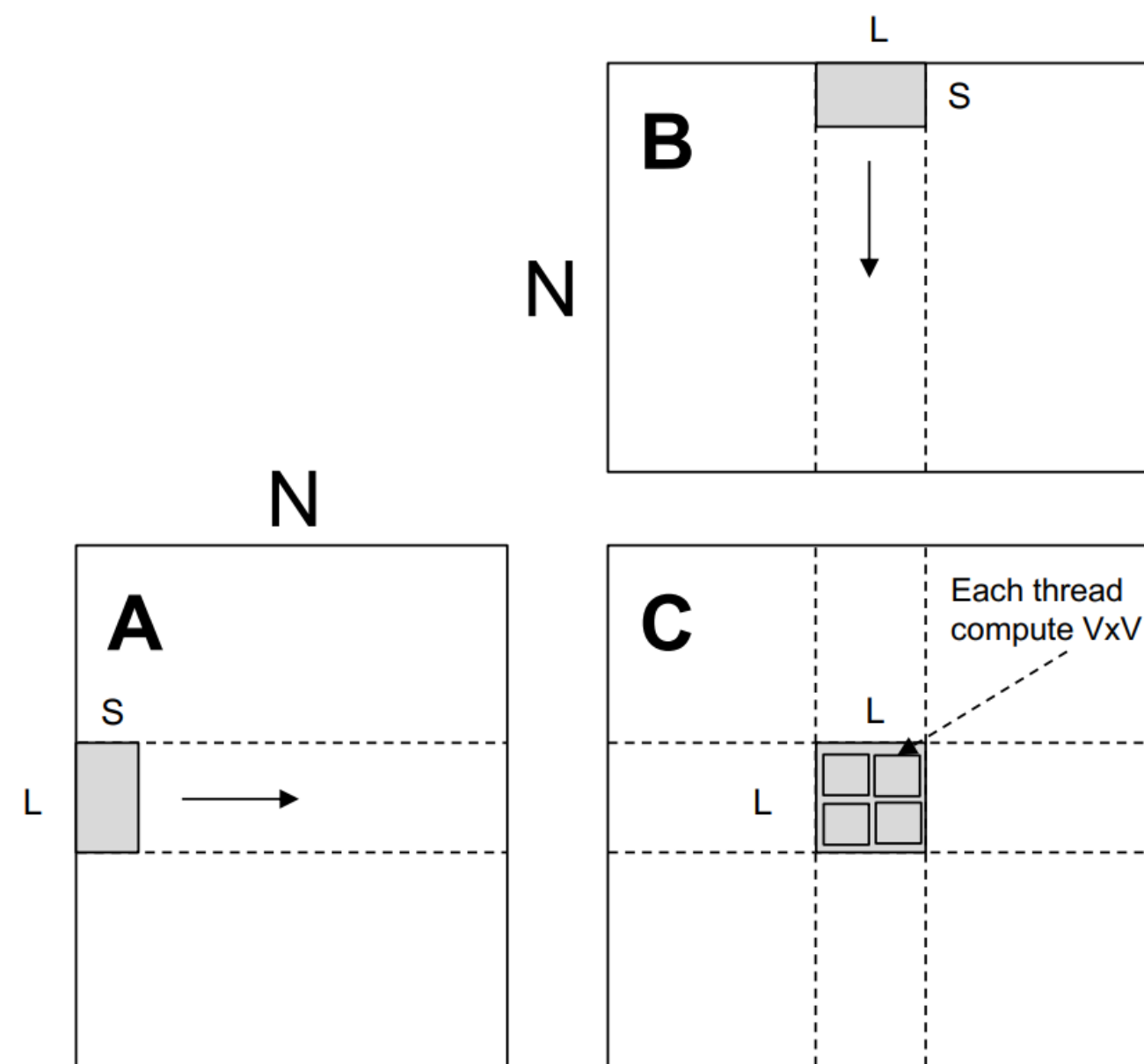
- Each thread computes a $V \times V$ submatrix

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {  
    int ybase = blockIdx.y * blockDim.y + threadIdx.y;  
    int xbase = blockIdx.x * blockDim.x + threadIdx.x;  
  
    float c[V][V] = {0};  
    float a[V], b[V];  
    for (int k = 0; k < N; ++k) {  
        a[:] = A[xbase*V : xbase*V + V, k];  
        b[:] = B[k, ybase*V : ybase*V + V];  
        for (int y = 0; y < V; ++y) {  
            for (int x = 0; x < V; ++x) {  
                c[x][y] += a[x] * b[y];  
            }  
        }  
    }  
    C[xbase * V : xbase*V + V, ybase*V : ybase*V + V] = c[:];  
}
```



Recall Cache-aware tiling -> block-level tiling

- Use block shared mem
- A block computes a $L \times L$ submatrix
- Then a thread computes a $V \times V$ submatrix and reuses the matrices in shared block memory



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {  
    __shared__ float sA[S][L], sB[S][L];  
    float c[V][V] = {0};  
    float a[V], b[V];  
    int yblock = blockIdx.y;  
    int xblock = blockIdx.x;  
  
    for (int ko = 0; ko < N; ko += S) {  
        __syncthreads();  
        // needs to be implemented by thread cooperative fetching  
        sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];  
        sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];  
        __syncthreads();  
        for (int ki = 0; ki < S; ++ki) {  
            a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];  
            b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];  
            for (int y = 0; y < V; ++y) {  
                for (int x = 0; x < V; ++x) {  
                    c[y][x] += a[y] * b[x];  
                }  
            }  
        }  
    }  
    int ybase = blockIdx.y * blockDim.y + threadIdx.y;  
    int xbase = blockIdx.x * blockDim.x + threadIdx.x;  
    C[ybase * V : ybase * V + V, xbase * V : xbase * V + V] = c[:];  
}
```

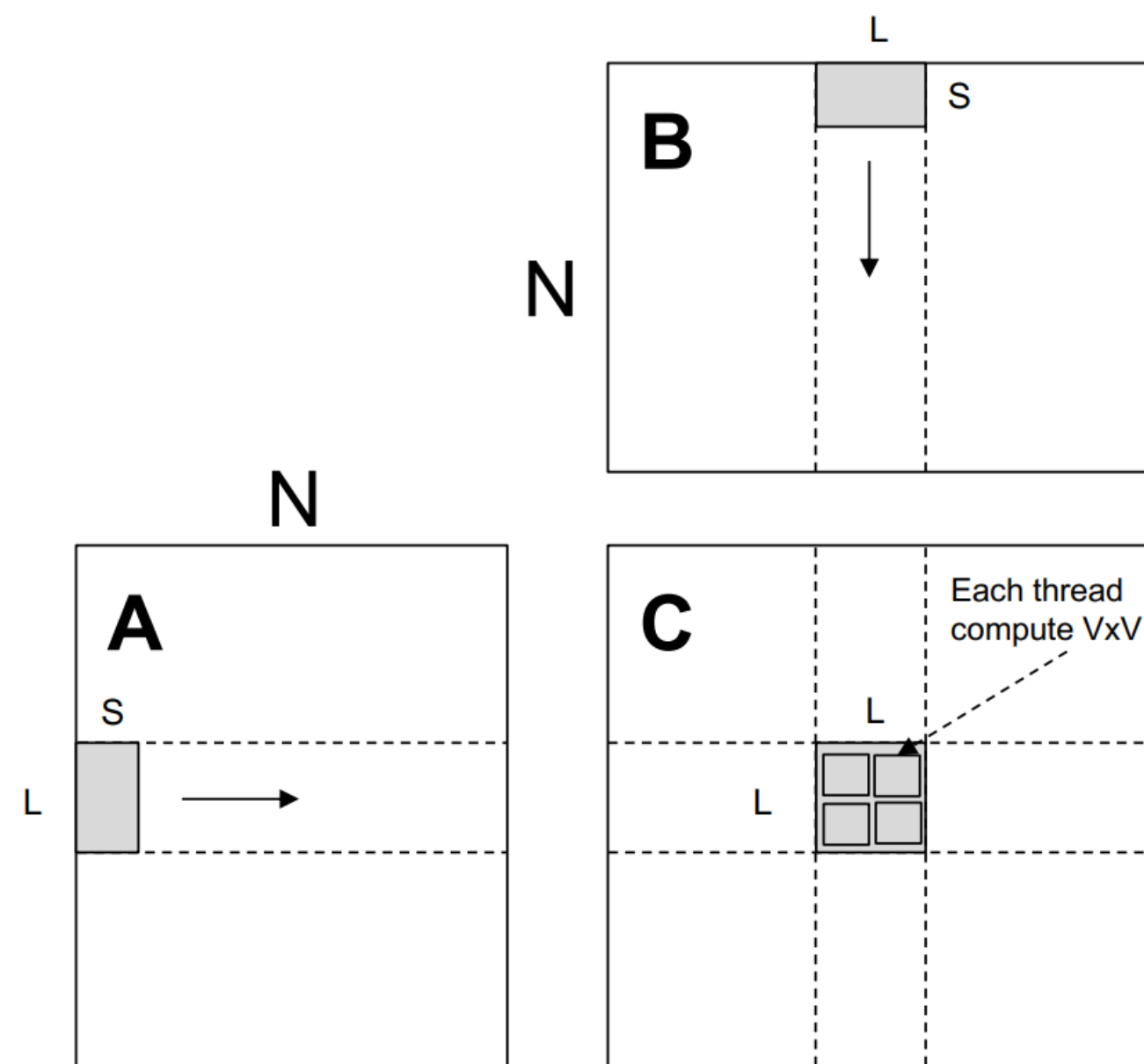

Memory overhead?

- Global memory access per threadblock
 - $2LN$
- Number of threadblocks:
 - N^2 / L^2
- Total global memory access:
 - $2N^3 / L$
- Shared memory access per thread:
 - $2VN$
- Number of threads
 - N^2 / V^2
- Total shared memory access:
 - $2N^3 / V$

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {  
    __shared__ float sA[S][L], sB[S][L];  
    float c[V][V] = {0};  
    float a[V], b[V];  
    int yblock = blockIdx.y;  
    int xblock = blockIdx.x;  
  
    for (int ko = 0; ko < N; ko += S) {  
        __syncthreads();  
        // needs to be implemented by thread cooperative fetching  
        sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];  
        sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];  
        __syncthreads();  
        for (int ki = 0; ki < S; ++ki) {  
            a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];  
            b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];  
            for (int y = 0; y < V; ++y) {  
                for (int x = 0; x < V; ++x) {  
                    c[y][x] += a[y] * b[x];  
                }  
            }  
        }  
    }  
    int ybase = blockIdx.y * blockDim.y + threadIdx.y;  
    int xbase = blockIdx.x * blockDim.x + threadIdx.x;  
    C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];  
}
```

Core Problems Here

- How to choose L/V? Tradeoffs:
 - #threads
 - #registers
 - Amount of shared memory



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
    __shared__ float sA[S][L], sB[S][L];
    float c[V][V] = {0};
    float a[V], b[V];
    int yblock = blockIdx.y;
    int xblock = blockIdx.x;

    for (int ko = 0; ko < N; ko += S) {
        __syncthreads();
        // needs to be implemented by thread cooperative fetching
        sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
        sB[:, :] = B[k : k + S, xblock * L : xblock * L + L];
        __syncthreads();
        for (int ki = 0; ki < S; ++ki) {
            a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
            b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];
            for (int y = 0; y < V; ++y) {
                for (int x = 0; x < V; ++x) {
                    c[y][x] += a[y] * b[x];
                }
            }
        }
    }
    int ybase = blockIdx.y * blockDim.y + threadIdx.y;
    int xbase = blockIdx.x * blockDim.x + threadIdx.x;
    C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
}
```

More GPU Optimizations

- Global memory continuous read
- Shared memory bank conflict
- Pipelining
- Tensor core
- Etc.

Next Topic:

LLMs

Parallelization

Single-device Optimization

Basics

Orange are parts of ML Compilation

Dataflow Graph

Autodiff

Graph Optimization

Parallelization

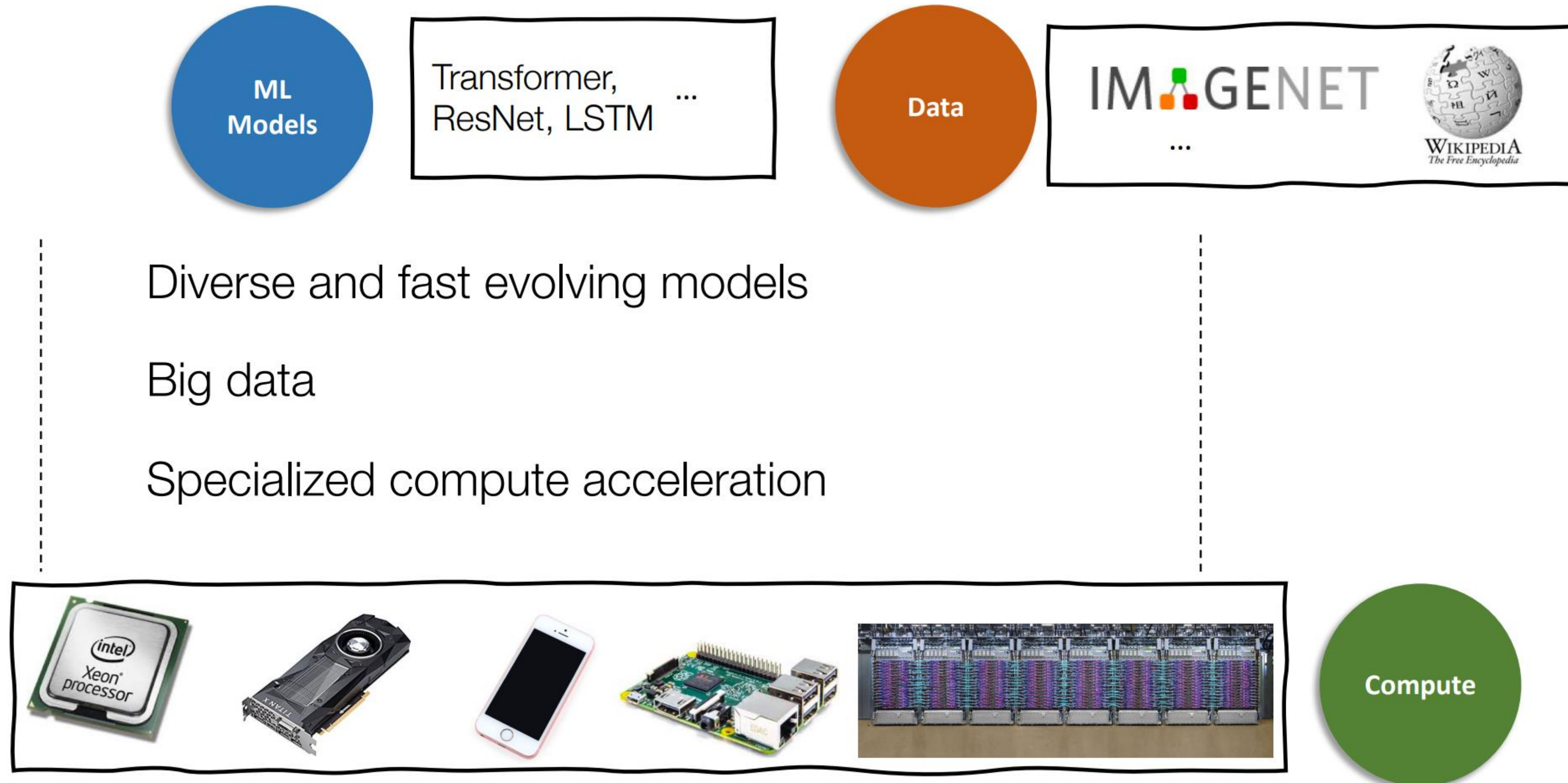
Runtime: schedule / memory

Operator optimization/compilation

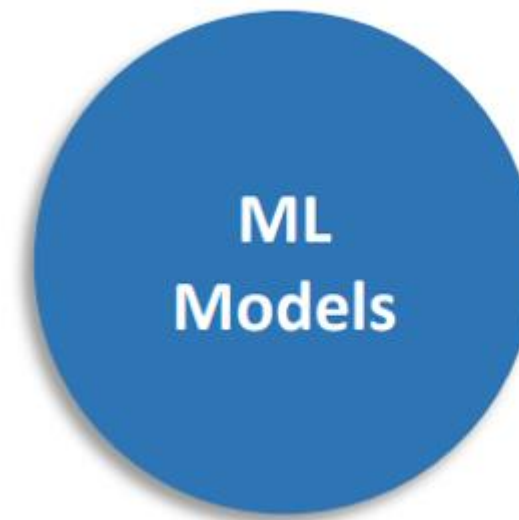
Agenda on this part

- ML Compilation Overview
 - Compiler
 - Graph optimization
- Memory Optimization
 - Activation checkpointing
 - Quantization and Mixed precision
- Two Guest Talks covering details in compilation, JIT, graph fusion, and beyond:
 - Meta PyTorch lead developer: Jason Ansel
 - Google JAX/XLA lead developer: Jinliang Wei

ML Compilation Overview



In Reality



Transformer,
ResNet, LSTM ...



MKL-DNN



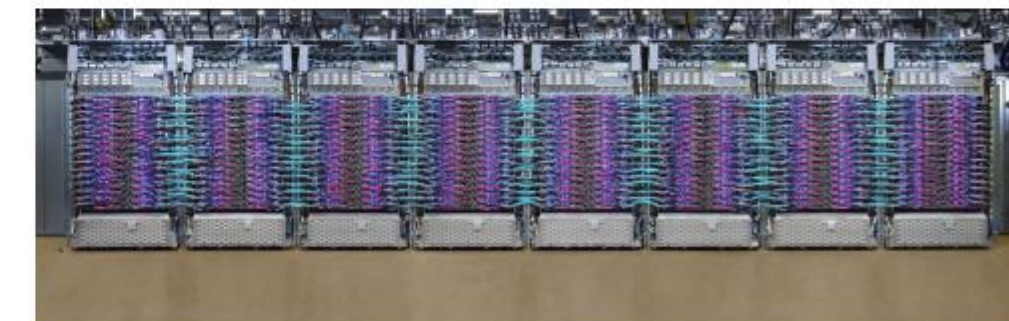
cuDNN



ARM-Compute



TPU Backends



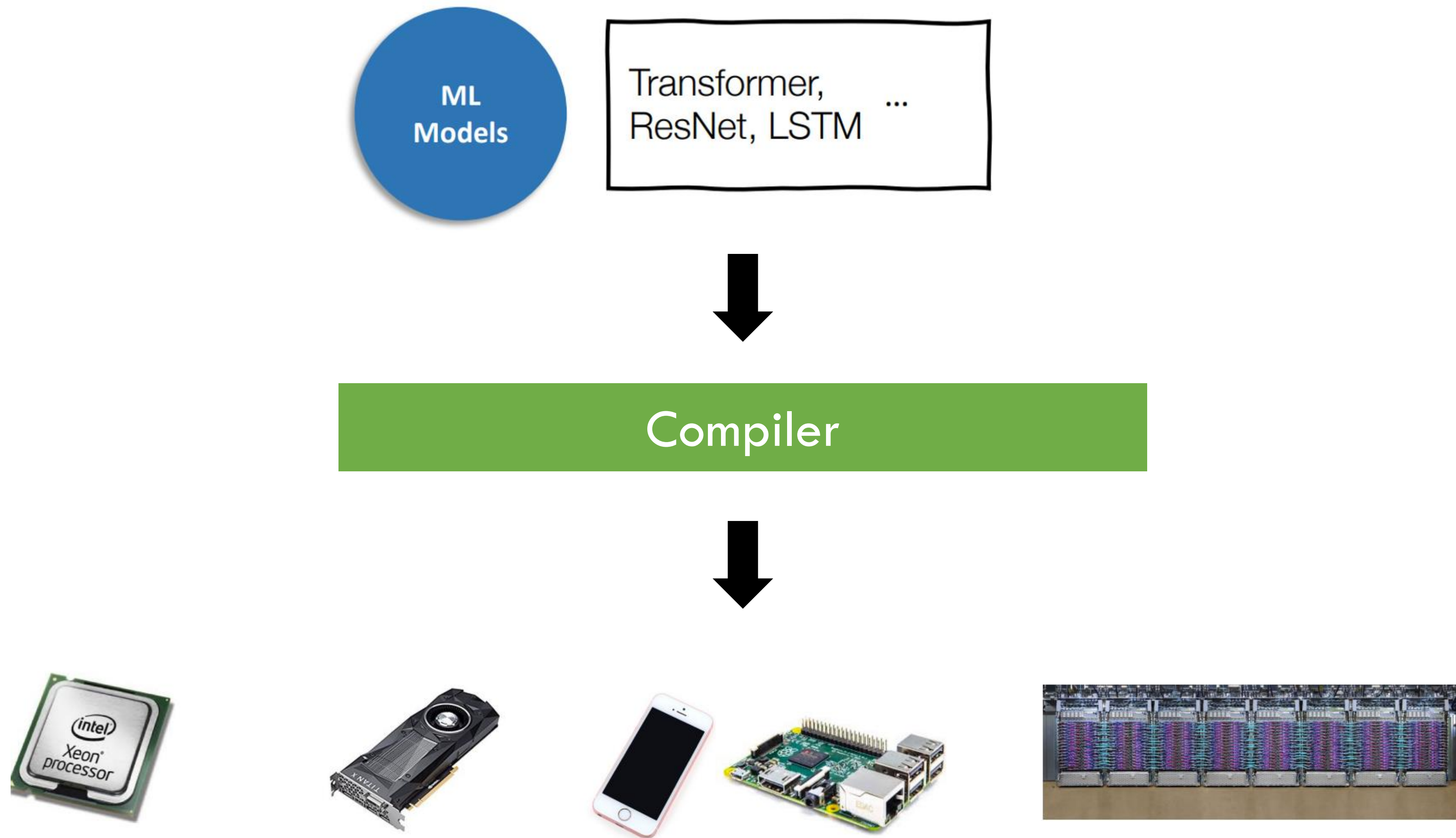
Goals

There are many equivalent ways to run the same model execution.

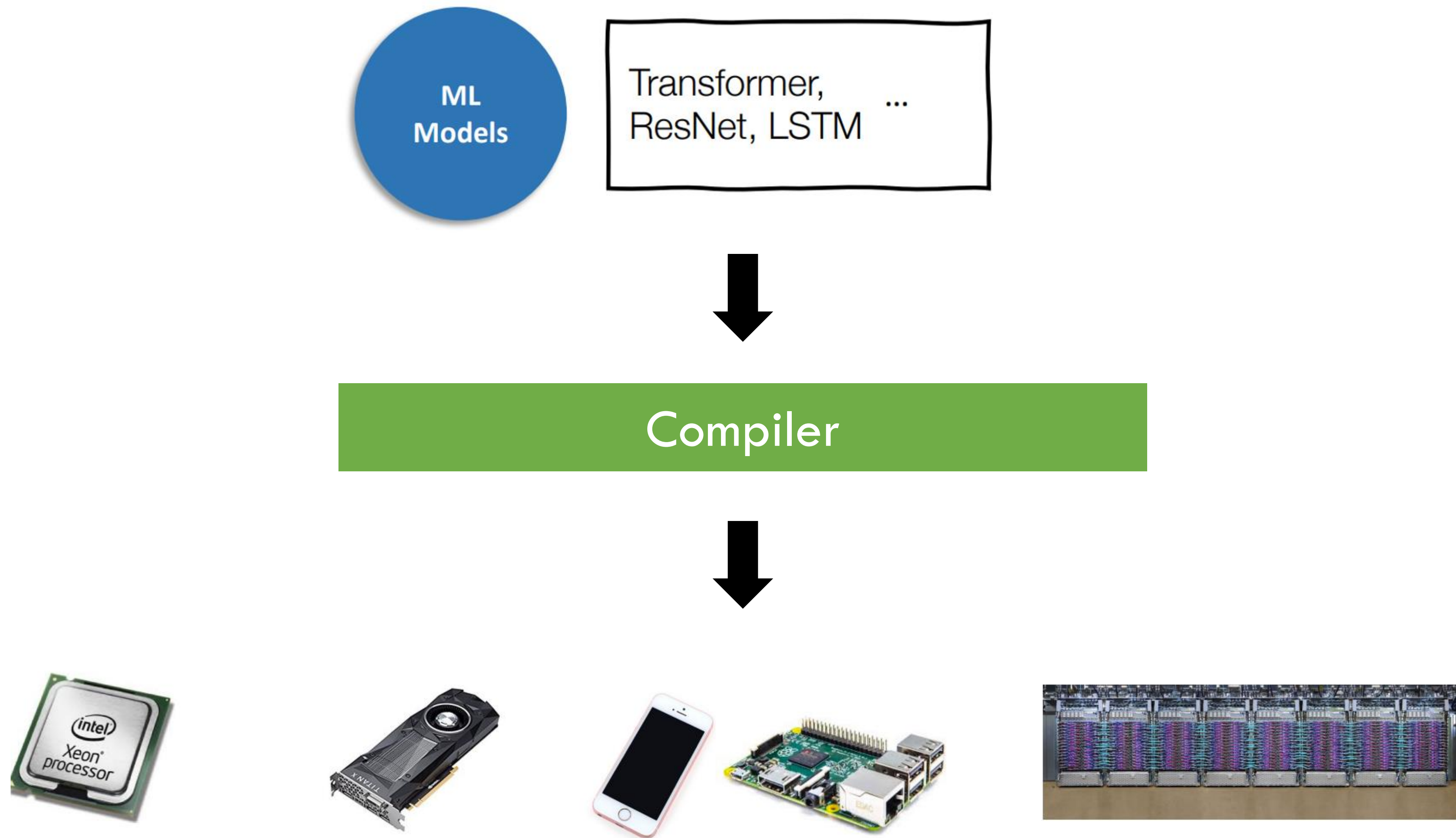
The common theme of MLC is optimization in different forms:

- Minimize memory usage
- Maximize execution efficiency
- Scaling to heterogeneous devices
- Minimize developer overhead

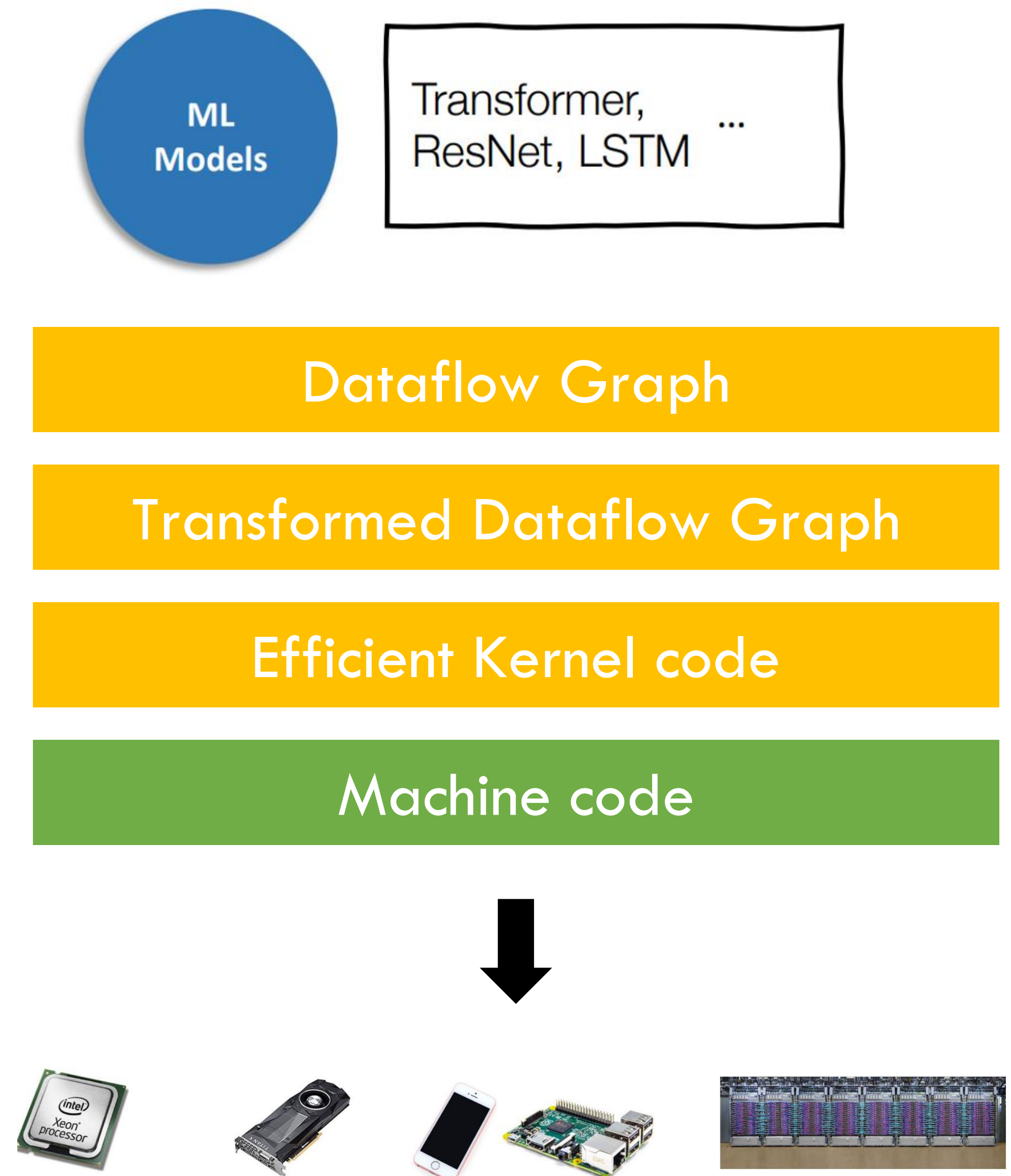
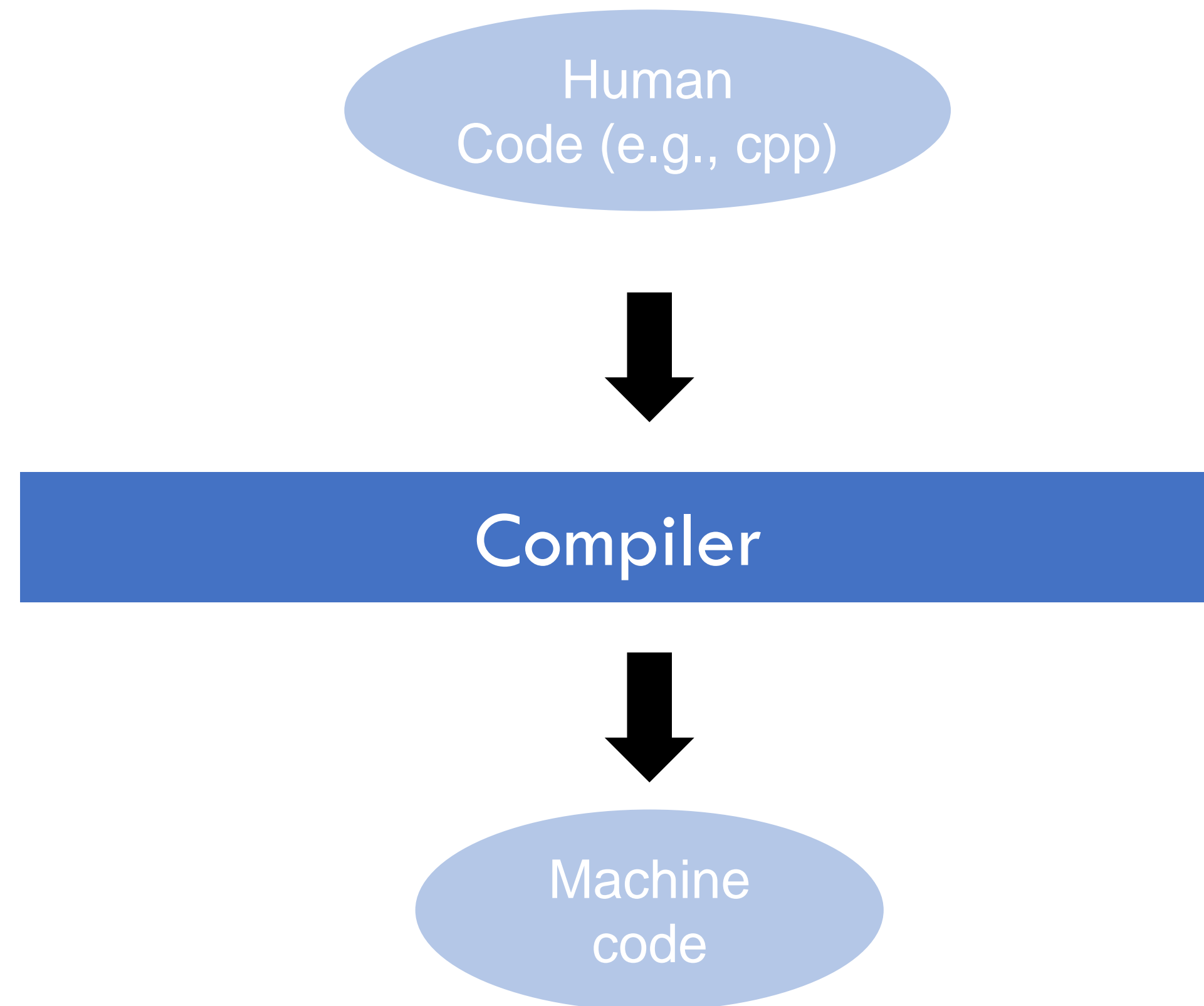
ML Compilation Goals



ML Compilation Goals



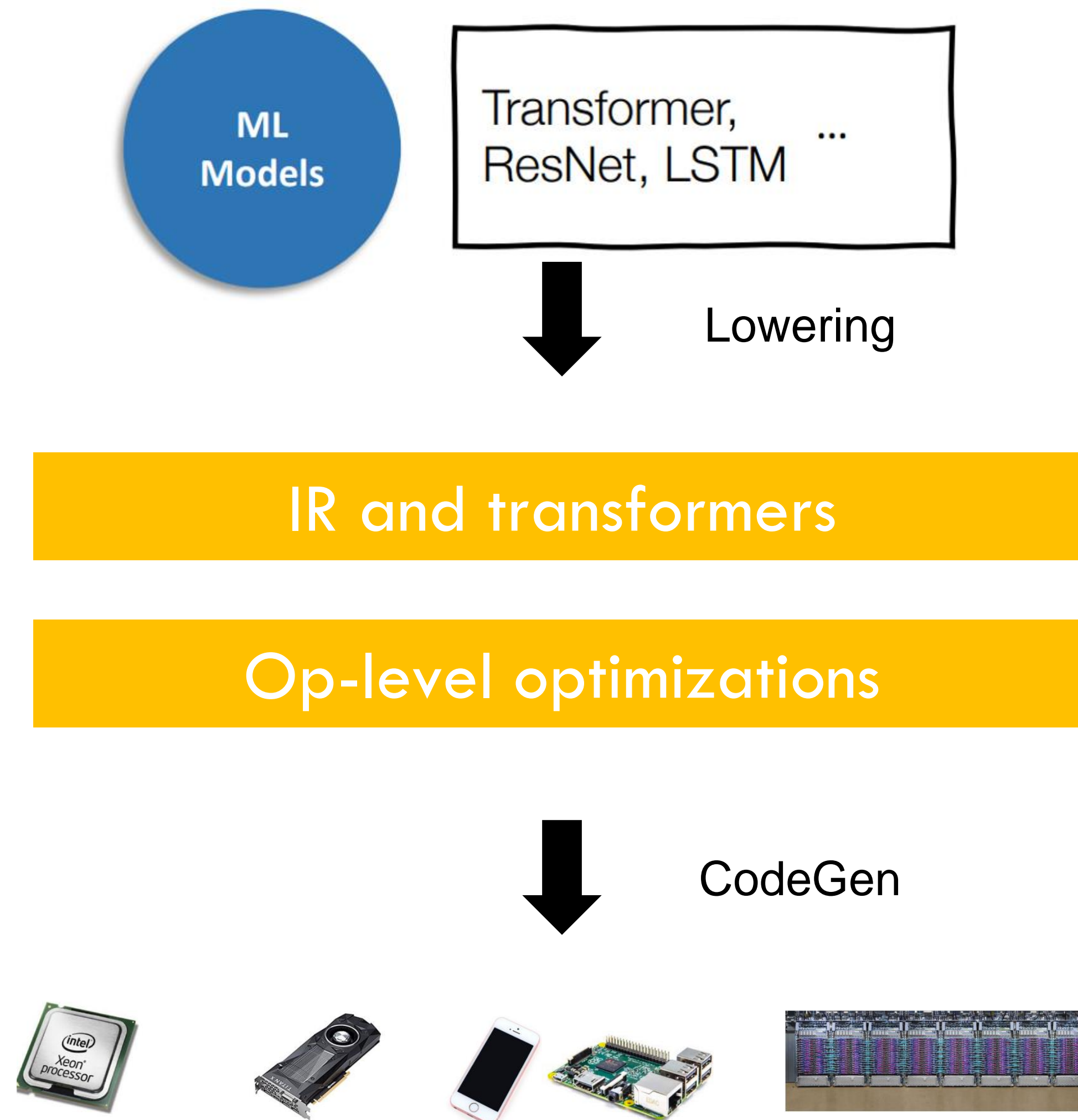
What is a Traditional Compiler?



Problems:

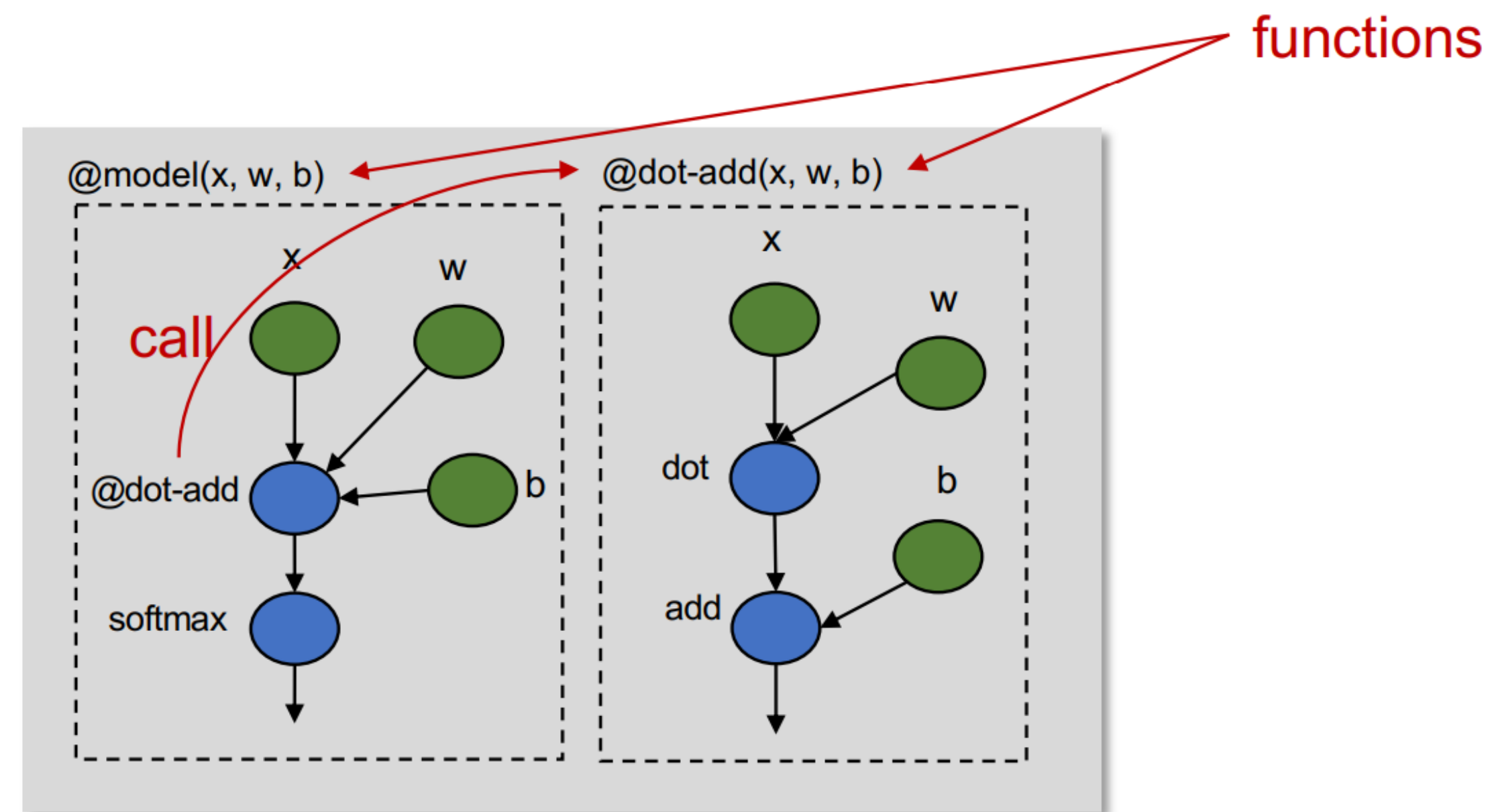
- Op-level: How to make operator fast on different hardware?
 - Tiling Based on register/cache/shared mem sizes
 - Use target device-specific accelerations
 - Generate the operator implementations automatically
- Graph-level: graph transformations to make it faster
- Programming-level:
 - How to transform an imperative code (by developers) into a compile-able code?

Compilation Process Today



IR: Intermediate representation

- What is the difference between this IR and the dataflow graph?



IRModule: a collection of interdependent functions

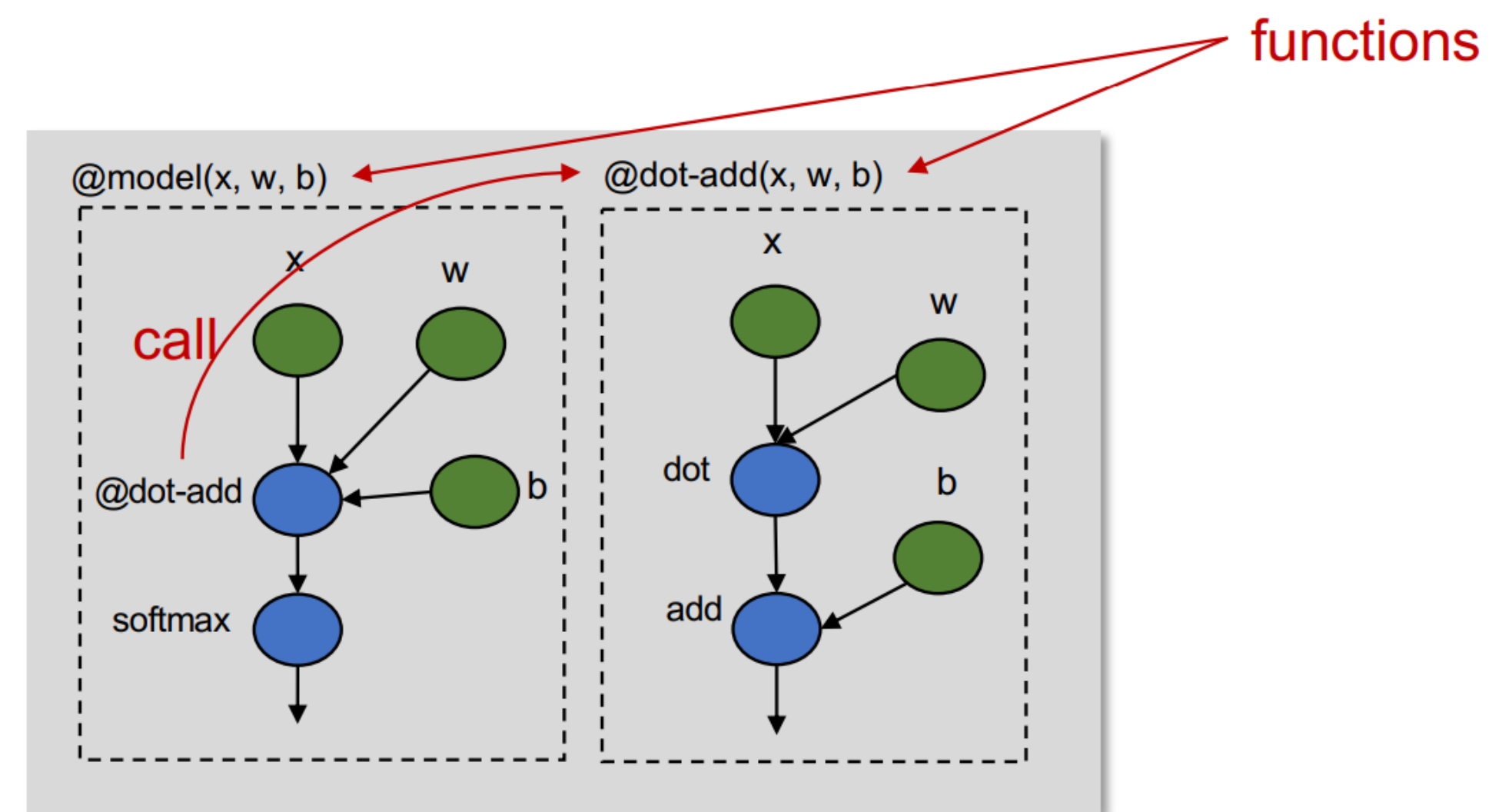
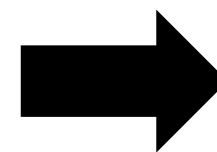
Notable Compilers

There are many different IRs by different compilers

- XLA: Accelerated Linear Algebra
 - HLO
- TVM: tensor virtual machine
 - IRModule (we used this on in class)
- Torch.compile: PyTorch
- Modular: Chris Lattner's startup

User Code transformations

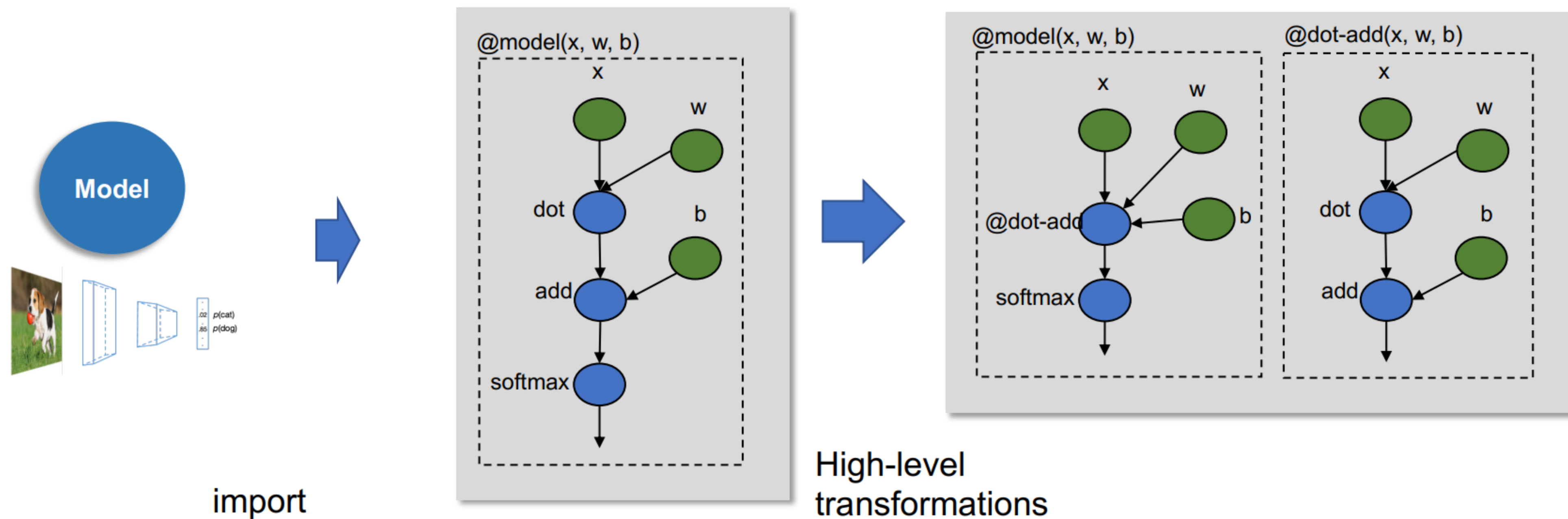
- What are potential challenges of user code parsing?



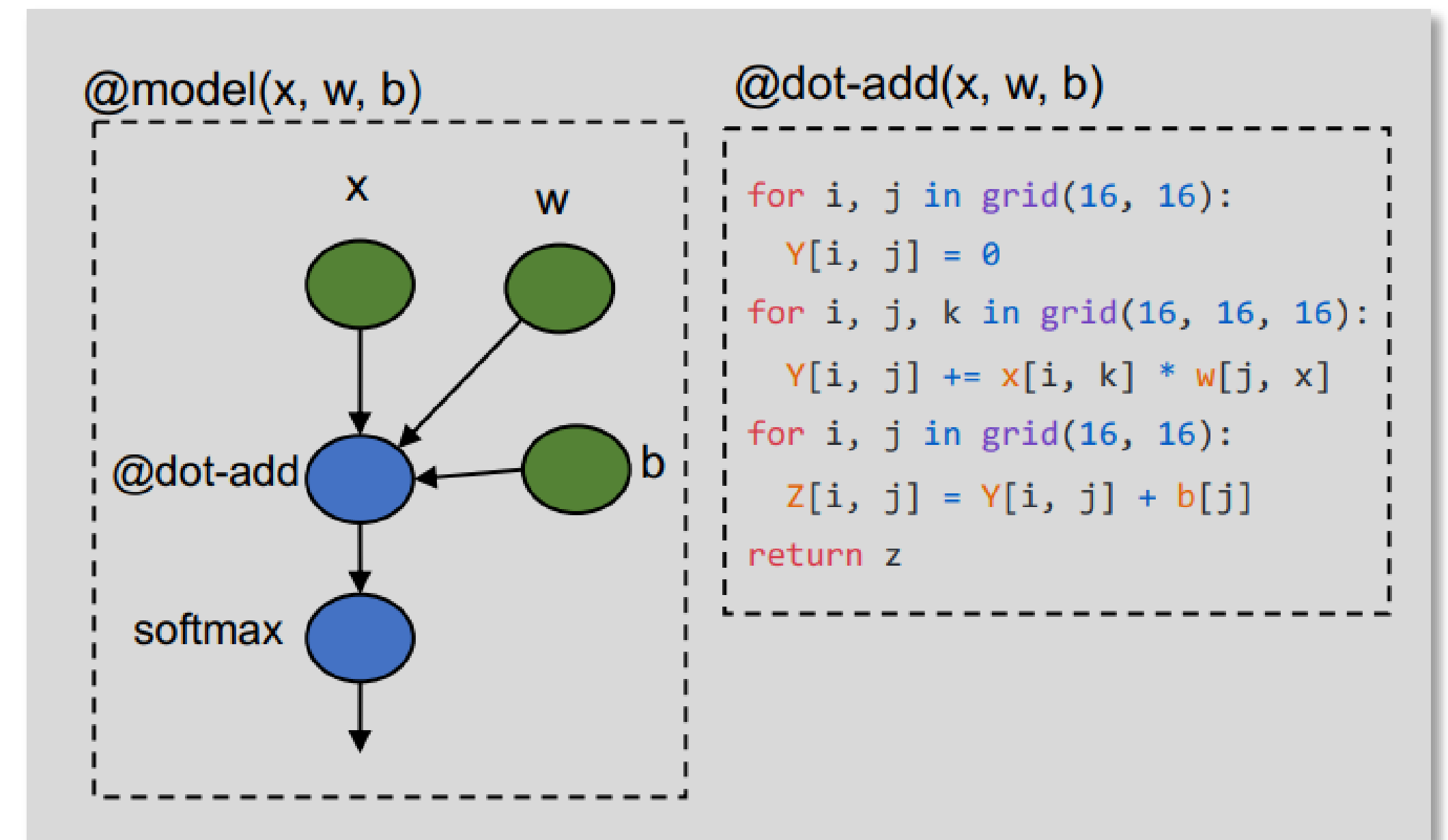
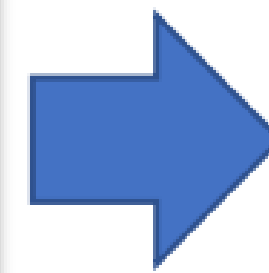
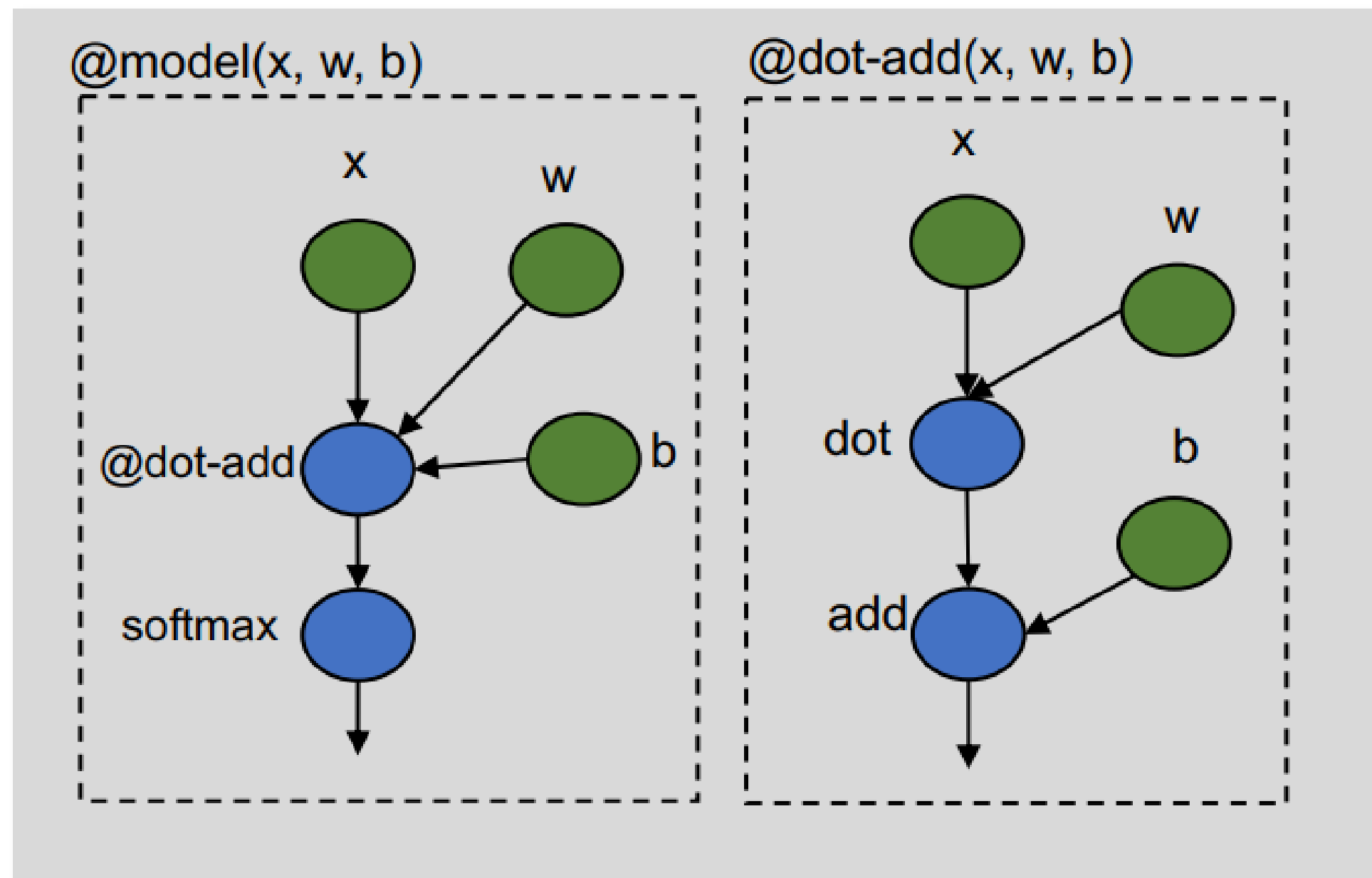
IRModule: a collection of interdependent functions

Example Compile flow: high-level transformations

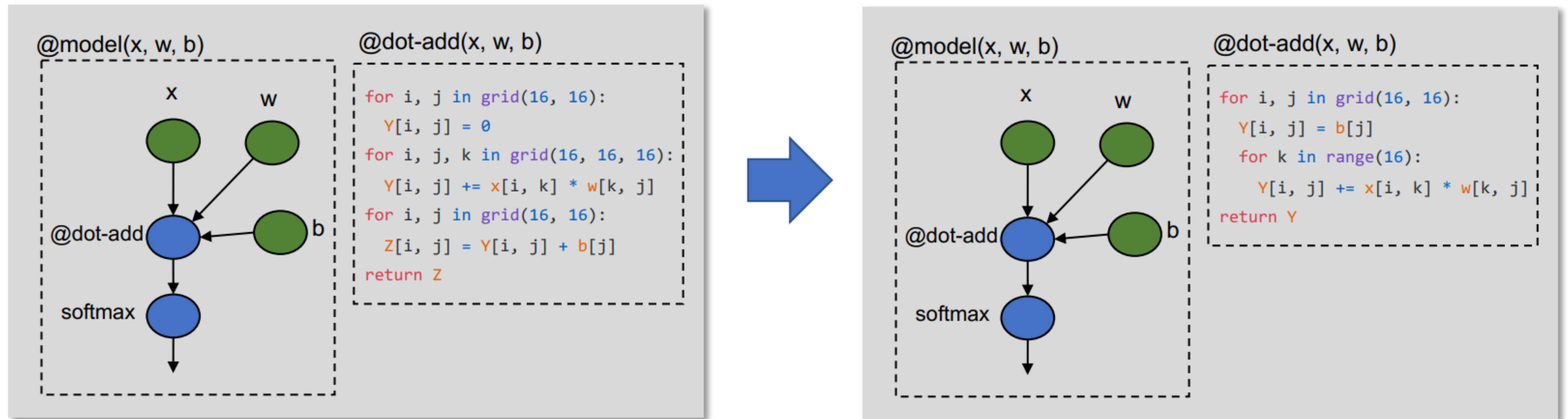
- We'll talk about some techniques here next week



Example Compile flow: lowering to loop IR



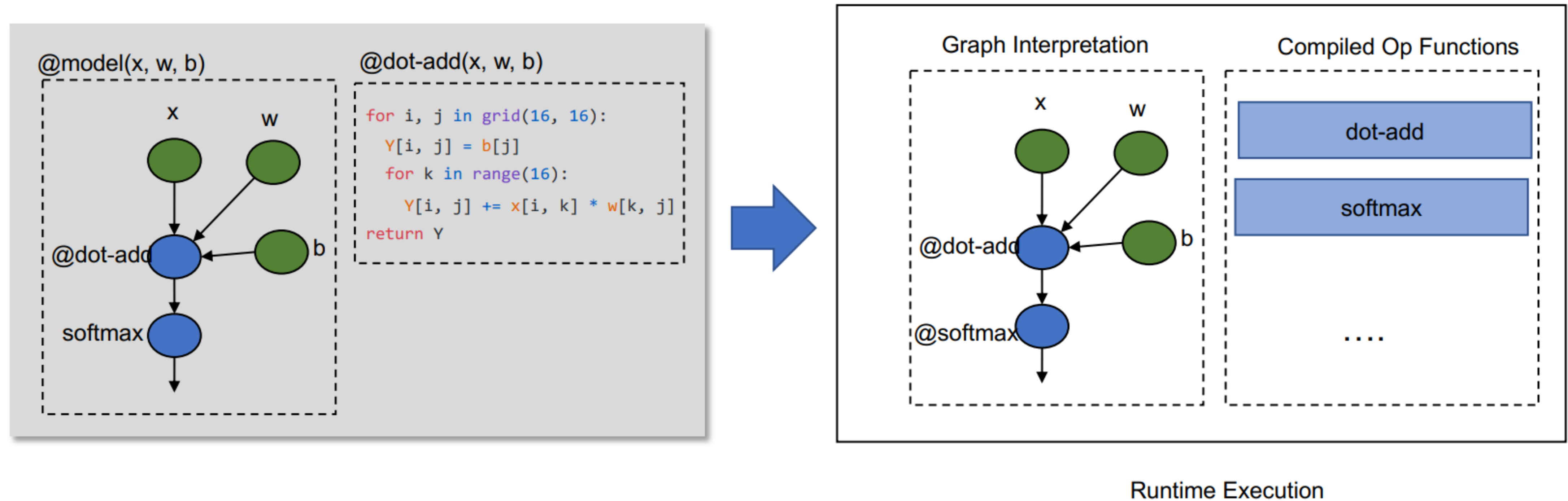
Example Compile flow: Loop transformers



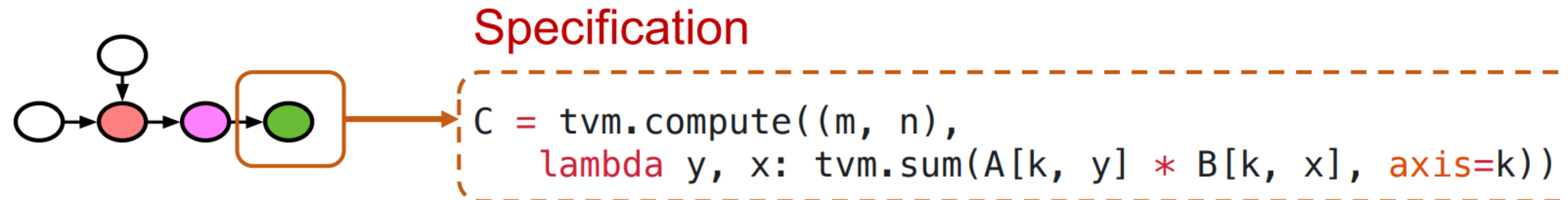
Low-level transformations

Example Compilation: CodeGen

Eventually, we transform a user code into some binary artifacts



Lower-level code optimization



Search Space of Possible Program Optimizations

Low-level Program Variants

```
inp_buffer AL[8][8], BL[8][8]  
acc_buffer CL[8][8]  
for yo in range(128):  
    for xo in range(128):  
        vdlc.fill_zero(CL)  
        for ko in range(128):  
            vdlc.dma_copy2d(AL, A[ko*8:ko*8+8][yo*8:yo*8+8])  
            vdlc.dma_copy2d(BL, B[ko*8:ko*8+8][xo*8:xo*8+8])  
            vdlc.fused_gemm8x8_add(CL, AL, BL)  
            vdlc.dma_copy2d(C[yo*8:yo*8+8,xo*8:xo*8+8], CL)
```

```
for yo in range(128):  
    for xo in range(128):  
        C[yo*8:yo*8+8][xo*8:xo*8+8] = 0  
        for ko in range(128):  
            for yi in range(8):  
                for xi in range(8):  
                    for ki in range(8):  
                        C[yo*8+yi][xo*8+xi] +=  
                            A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

```
for y in range(1024):  
    for x in range(1024):  
        C[y][x] = 0  
        for k in range(1024):  
            C[y][x] += A[k][y] * B[k][x]
```

Low-level Loop Representation

@dot-add(x, w, b)

```
for i, j in grid(16, 16):  
    Y[i, j] = 0  
for i, j, k in grid(16, 16, 16):  
    Y[i, j] += x[i, k] * w[k, j]  
for i, j in grid(16, 16):  
    Z[i, j] = Y[i, j] + b[j]
```

Multi-dimensional
buffer

Loop nests

Array
computation

Transforming Loops: Loop Splitting

Code

```
for x in range(128):  
    C[x] = A[x] + B[x]
```



```
for xo in range(32):  
    for xi in range(4):  
        C[xo * 4 + xi]  
        = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")  
xo, xi = split(x, 4)
```


Transforming Loops: Loop Reorder

Code

```
for xo in range(32):  
    for xi in range(4):  
        C[xo * 4 + xi]  
        = A[xo * 4 + xi] + B[xo * 4 + xi]
```



```
for xi in range(4):  
    for xo in range(32):  
        C[xo * 4 + xi]  
        = A[xo * 4 + xi] + B[xo * 4 + xi]
```

Transformation

```
x = get_loop("x")  
xo, xi = split(x, 4)  
reorder(xi, xo)
```

Transforming Loops: Thread Binding

Code

```
for xi in range(4):  
    for xo in range(32):  
        C[xo * 4 + xi]  
        = A[xo * 4 + xi] + B[xo * 4 + xi]
```



```
def gpu_kernel():  
    C[threadIdx.x * 4 + blockIdx.x] = . . .
```

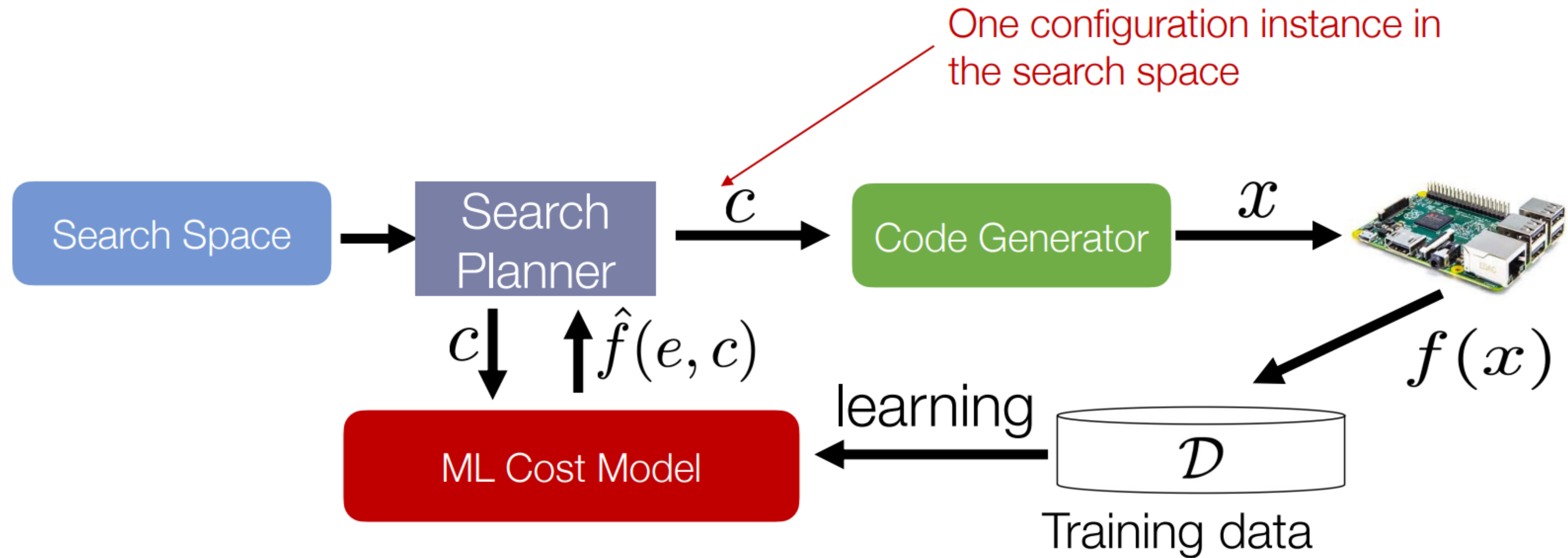
Transformation

```
x = get_loop("x")  
xo, xi = split(x, 4)  
reorder(xi, xo)  
bind_thread(xo, "threadIdx.x")  
bind_thread(xi, "blockIdx.x")
```

Problems

- We need to enumerate so many possibilities
- We need to fit with each device (register/cache sizes)
- We need to apply this to so many operators

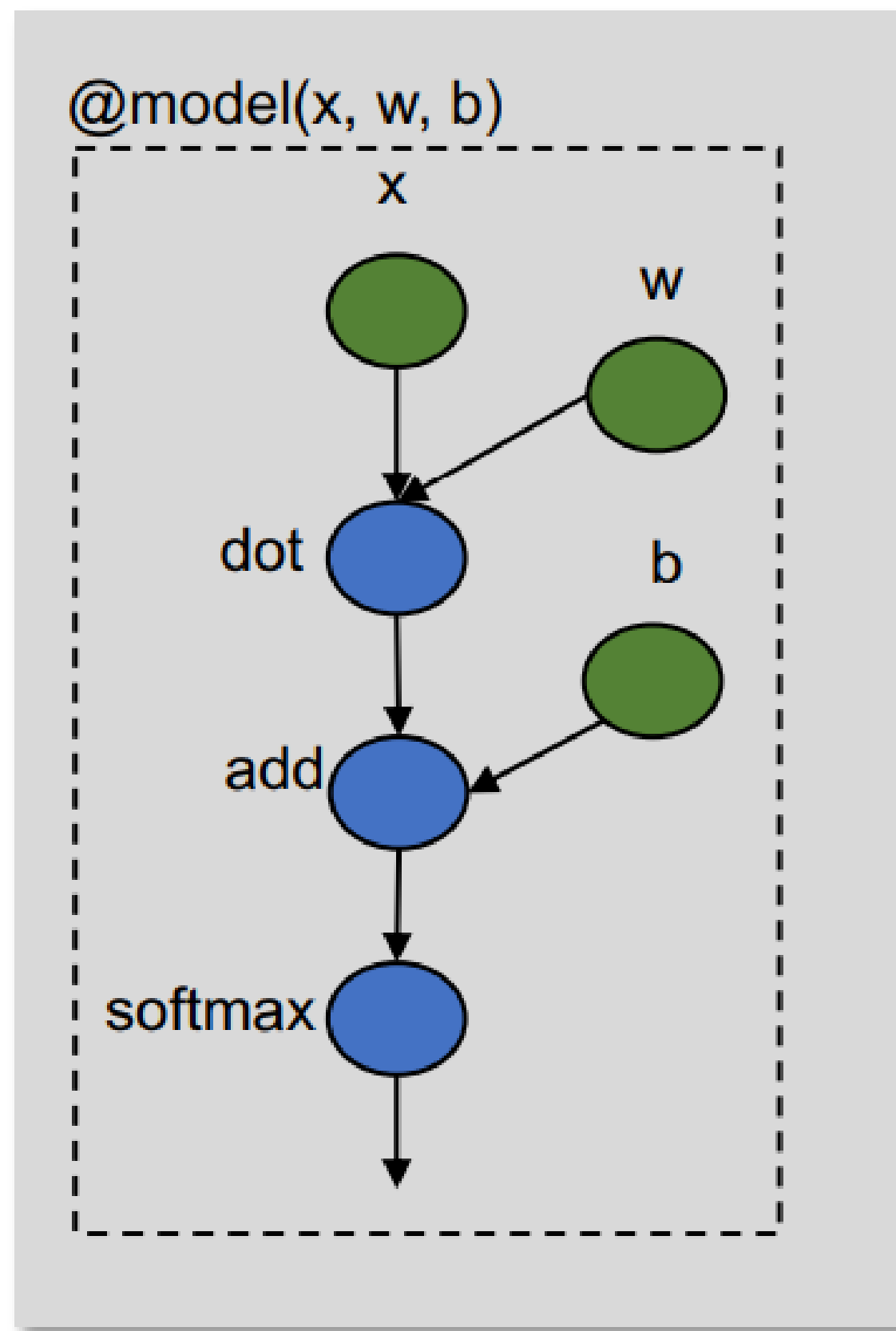
Search via Learned Cost Model



Elements of an automated ML Compiler

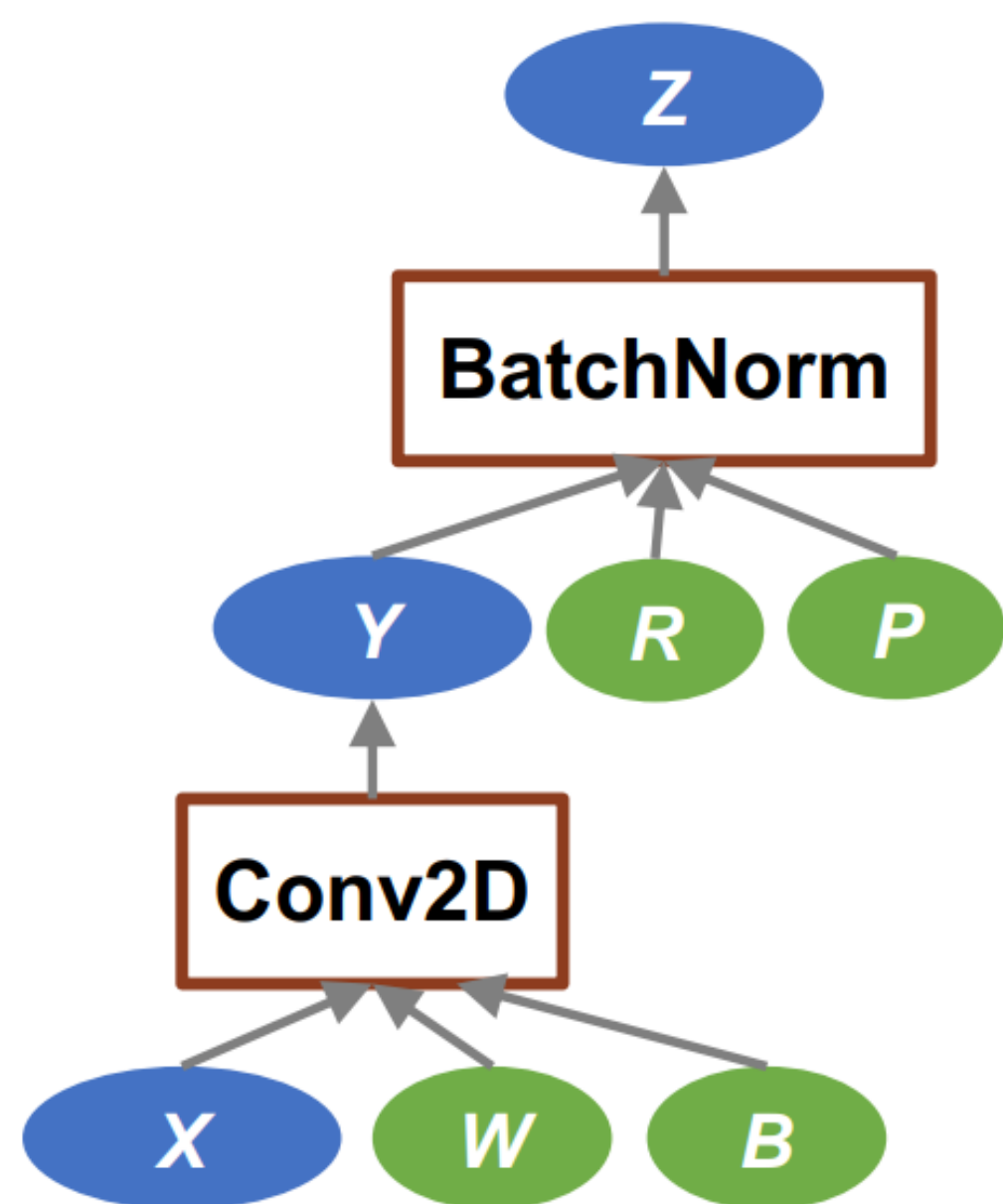
- Program abstraction
 - Represent the program/optimization of interest
- Build Search space through a set of transformations
 - Good coverage of common optimizations like tiling
- Effective Search
 - Accurate cost models
 - Transferability

High-level IR transformations



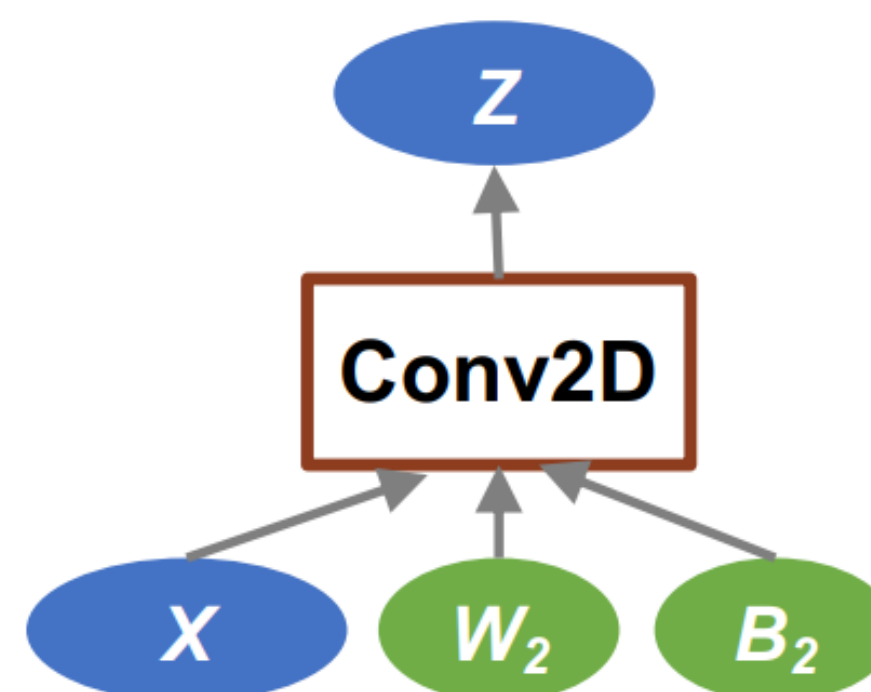
- Graph-like representation
- Each node is a tensor operator
- Can be transformed (e.g. fusion) and annotated (e.g., device placement)
- Most ML frameworks today have this layer

Recall: fusing conv and bn



$$Z(n, c, h, w) = \left(\sum_{d,u,v} X(n, d, h + u, w + v) * W_2(c, d, u, v) \right) + B_2(n, c, h, w)$$

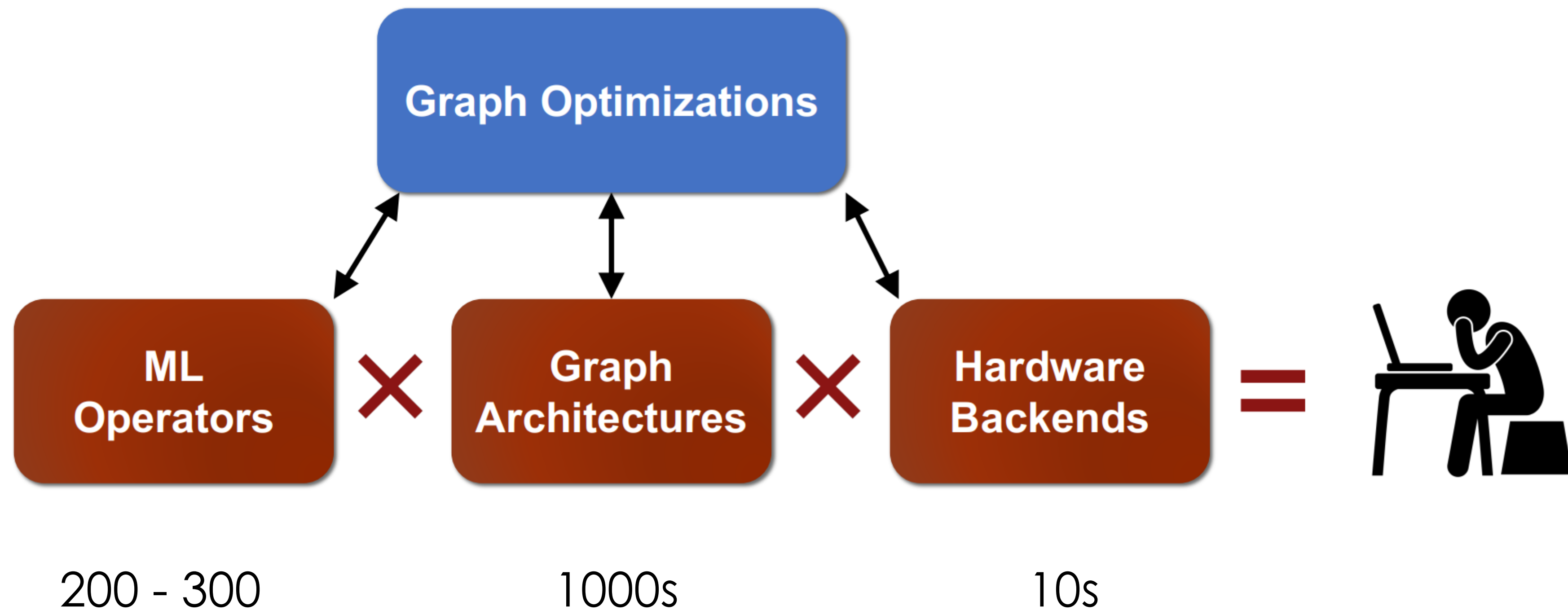
=



$$W_2(n, c, h, w) = W(n, c, h, w) * R(c)$$

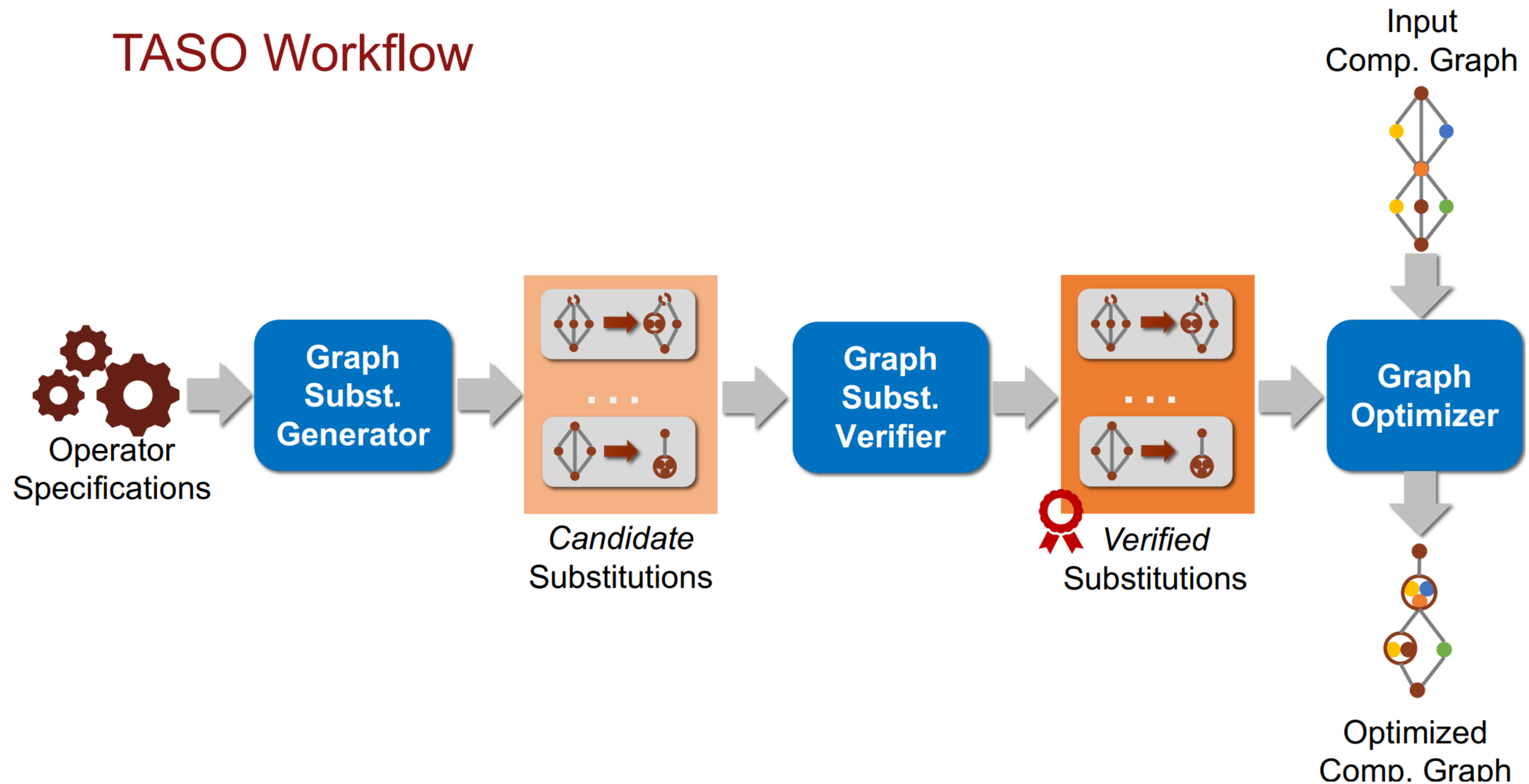
$$B_2(n, c, h, w) = B(n, c, h, w) * R(c) + P(c)$$

Problems of High-level Graph Optimizations



Idea: Enumerate and Verify

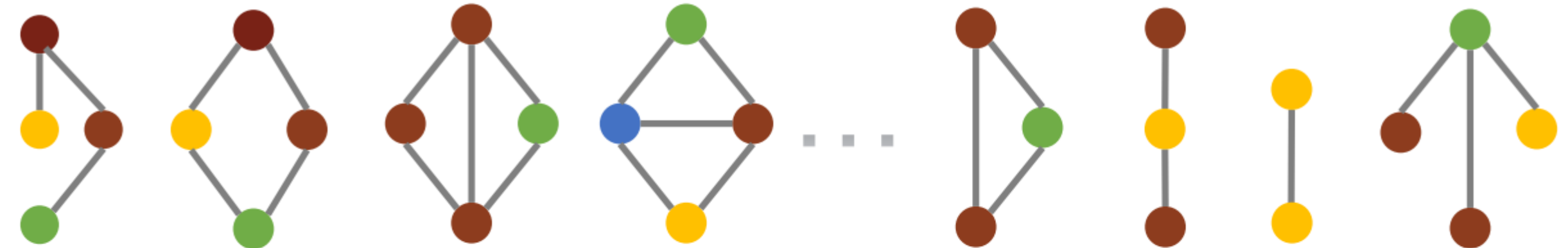
TASO Workflow



Graph Substitution Generator



Operators supported by
hardware backend



Enumerate all possible graphs up to a
fixed size using available operators