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

DSC 291: ML Systems
Spring 2024

LLMs

Parallelization

Single-device Optimization

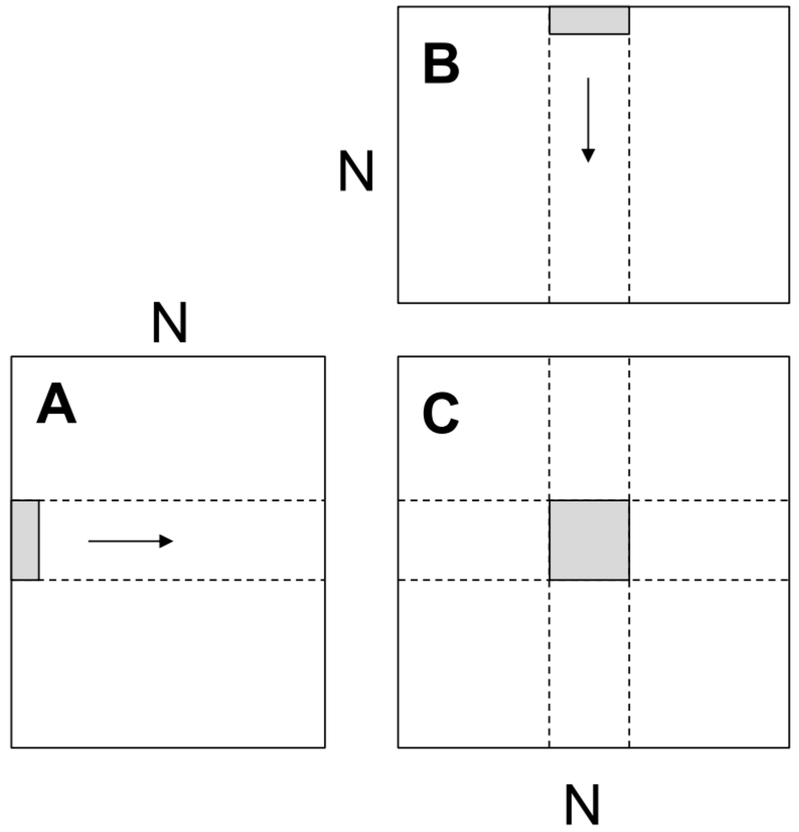
Basics

GPU and CUDA

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

Case study: GPU Matmul

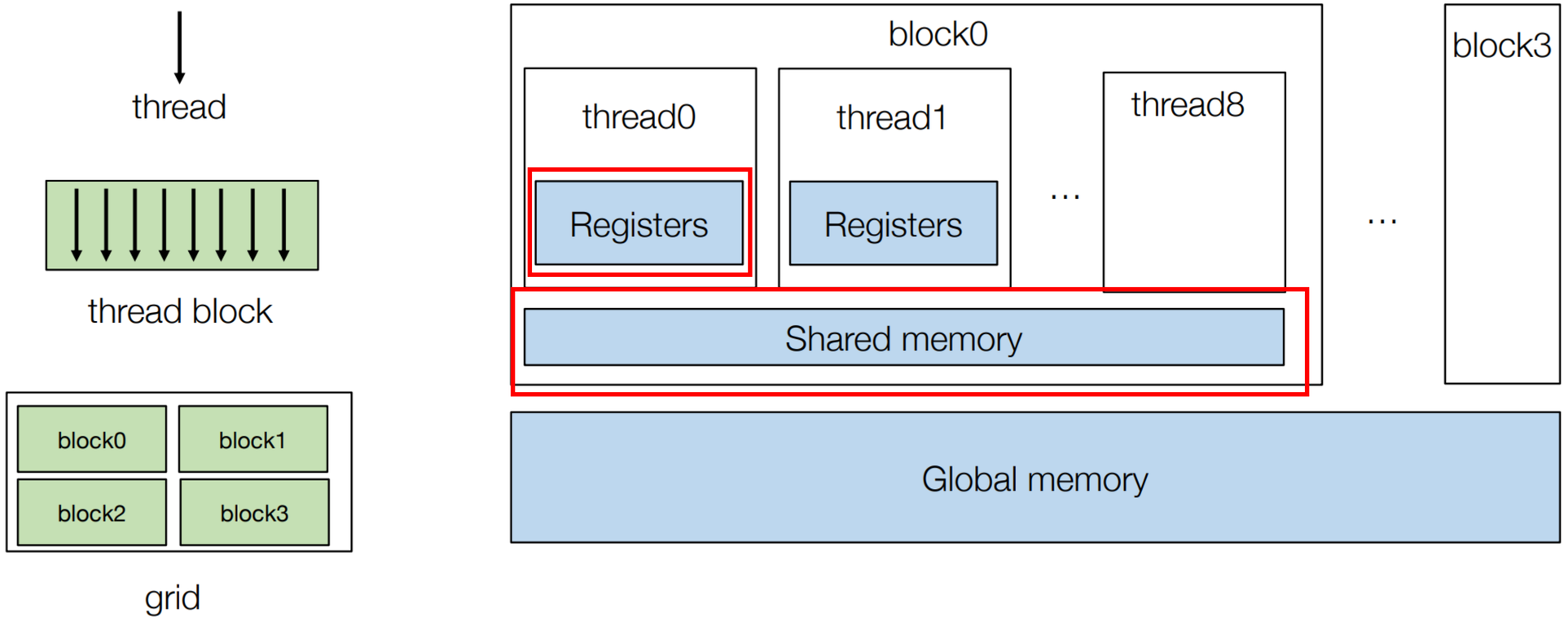
- Strawman solution:
  - C = A x B
  - Each thread computes one element




```
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;
}
```

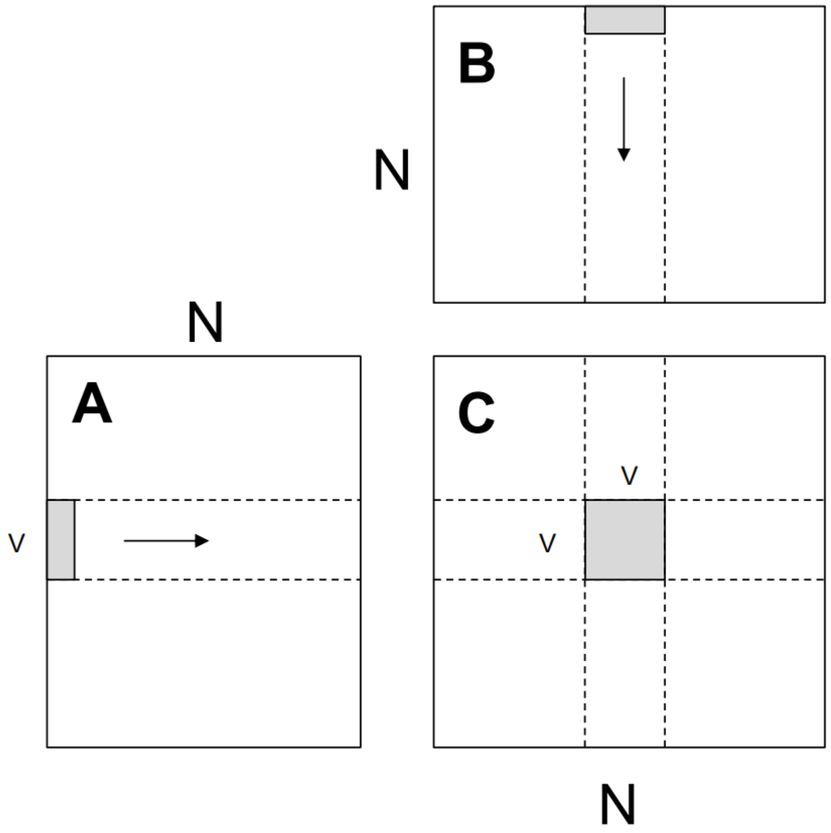
High-level Opt Idea: Recall Memory Hierarchy



Recall register tiling -> thread tiling

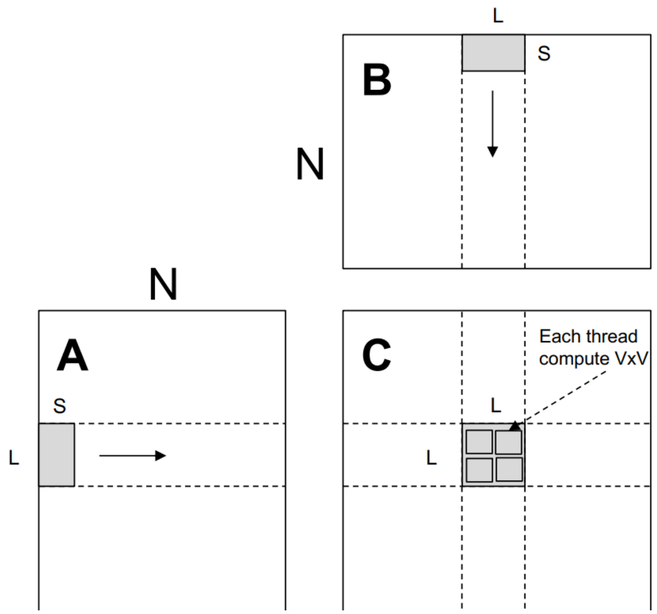
- Each thread computes a VxV submatrix

```
__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 x L submatrix
- Then a thread computes a V x 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[:];
}
```

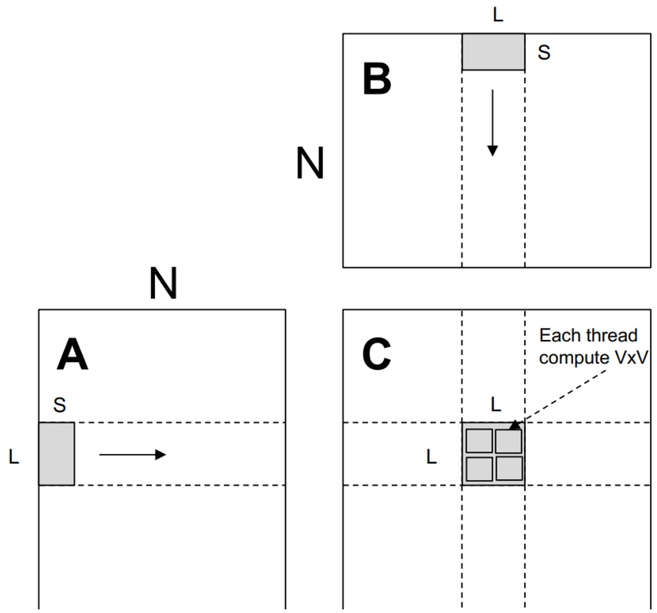
Memory overhead?

- 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

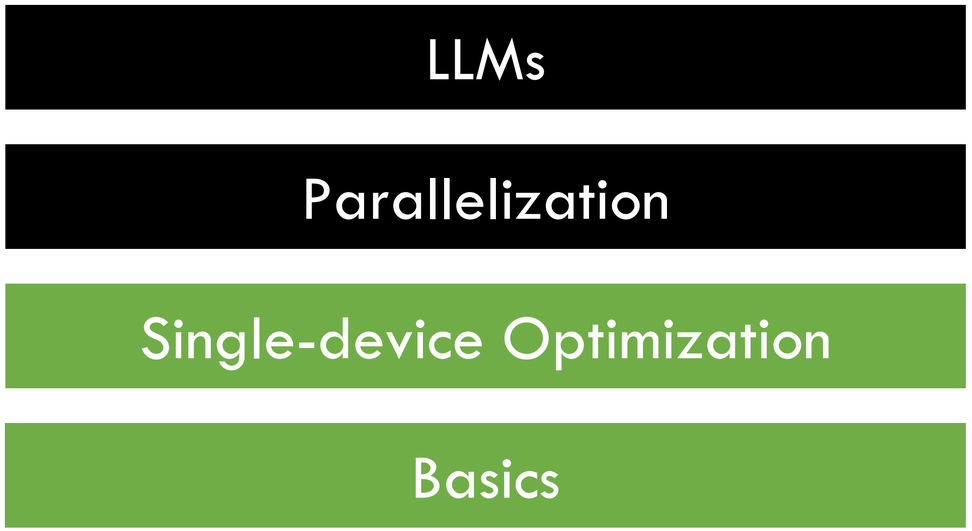



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {

  __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[:];
}
```

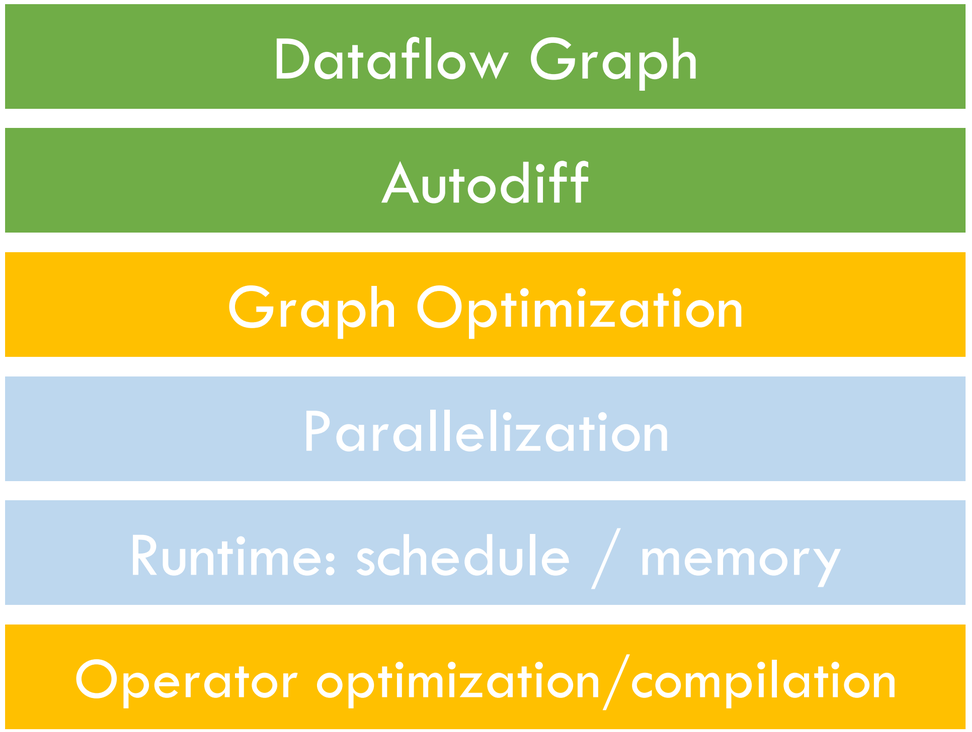
More GPU Optimizations

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

Next Topic:



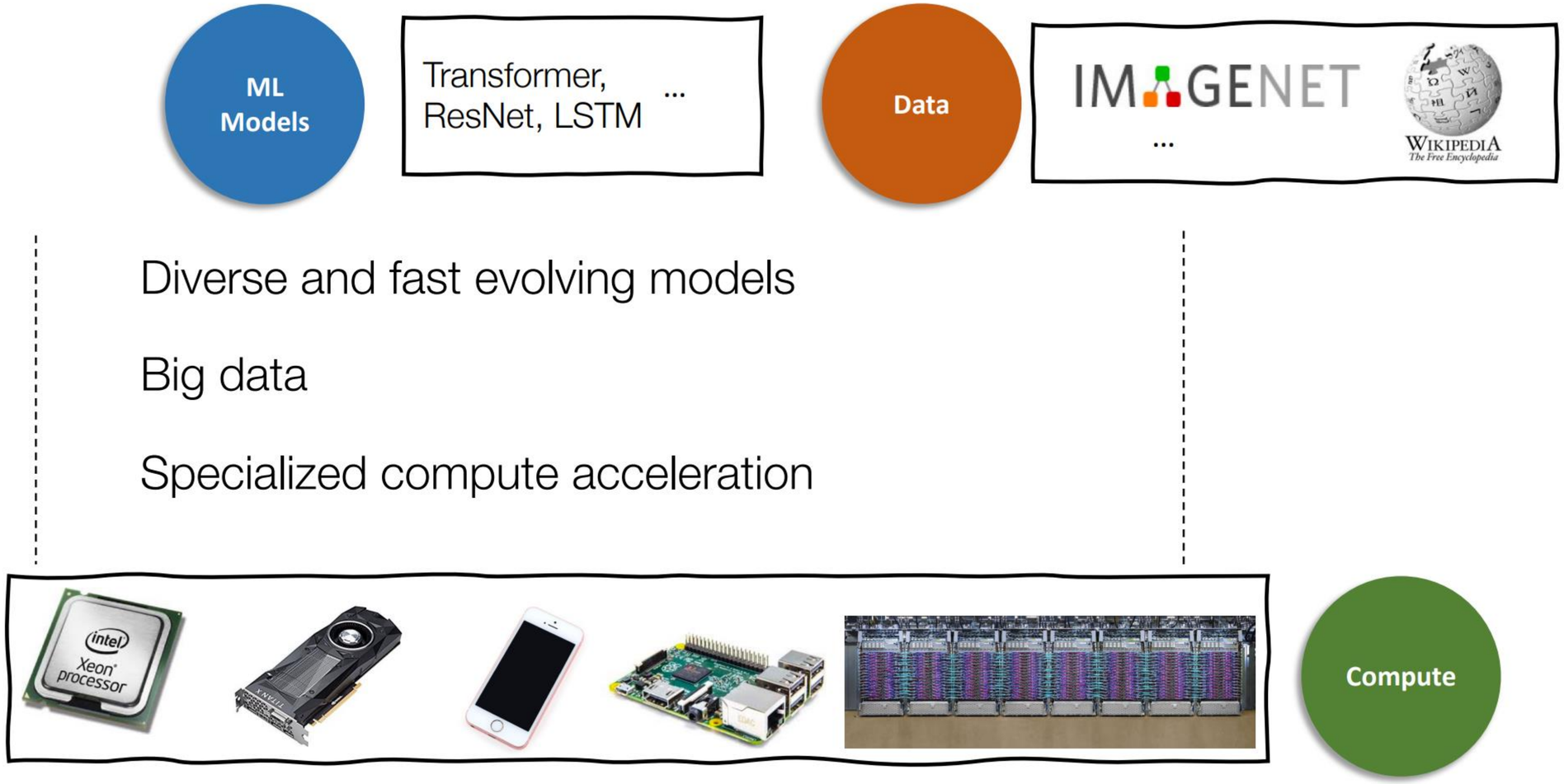
Orange are parts of ML Compilation



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



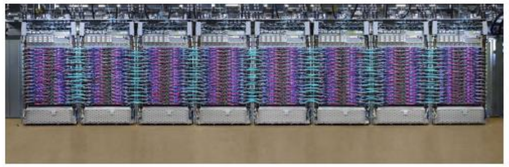
In Reality

MKL-DNN

cuDNN

ARM-Compute

TPU Backends



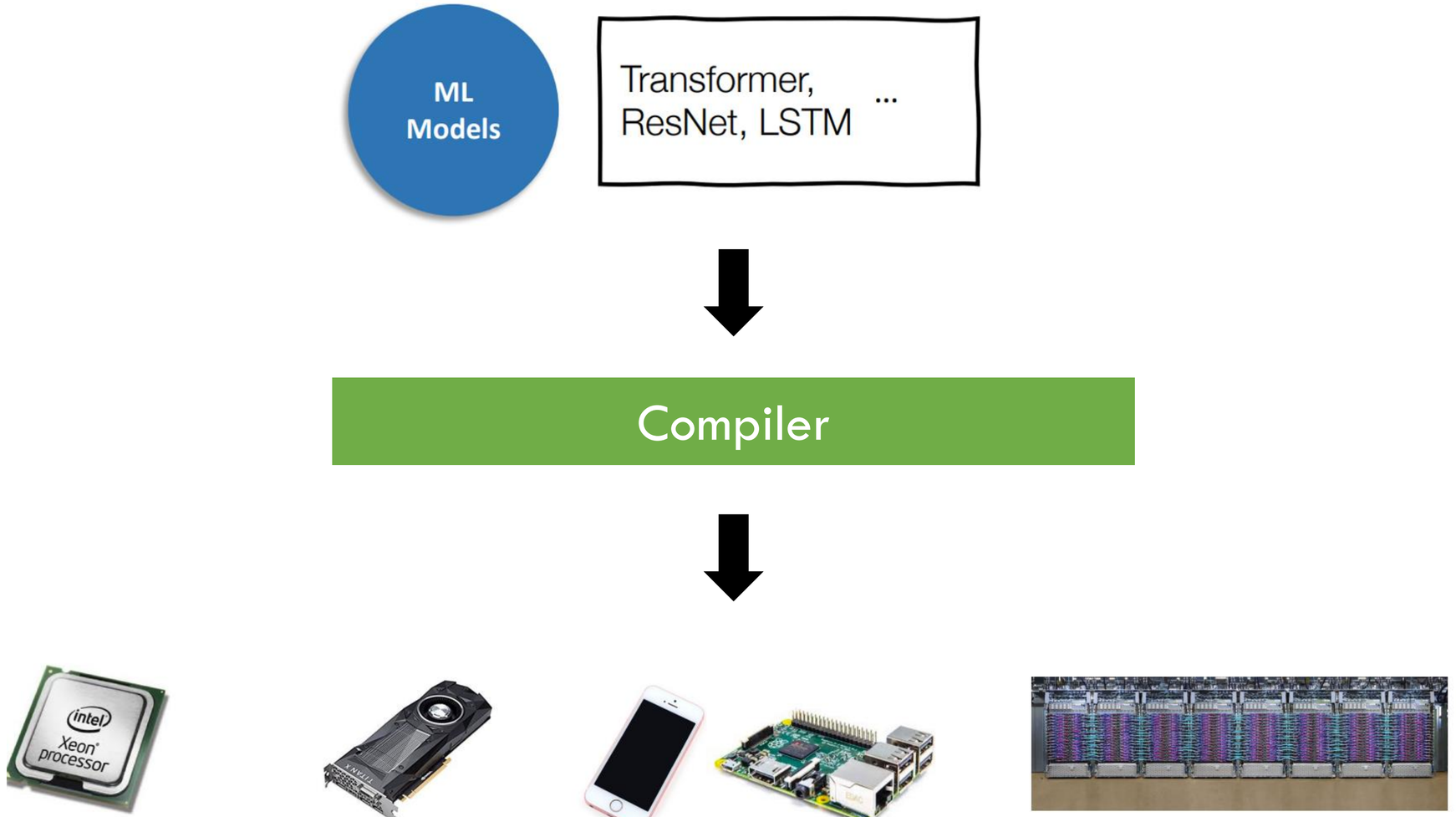
Goals

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

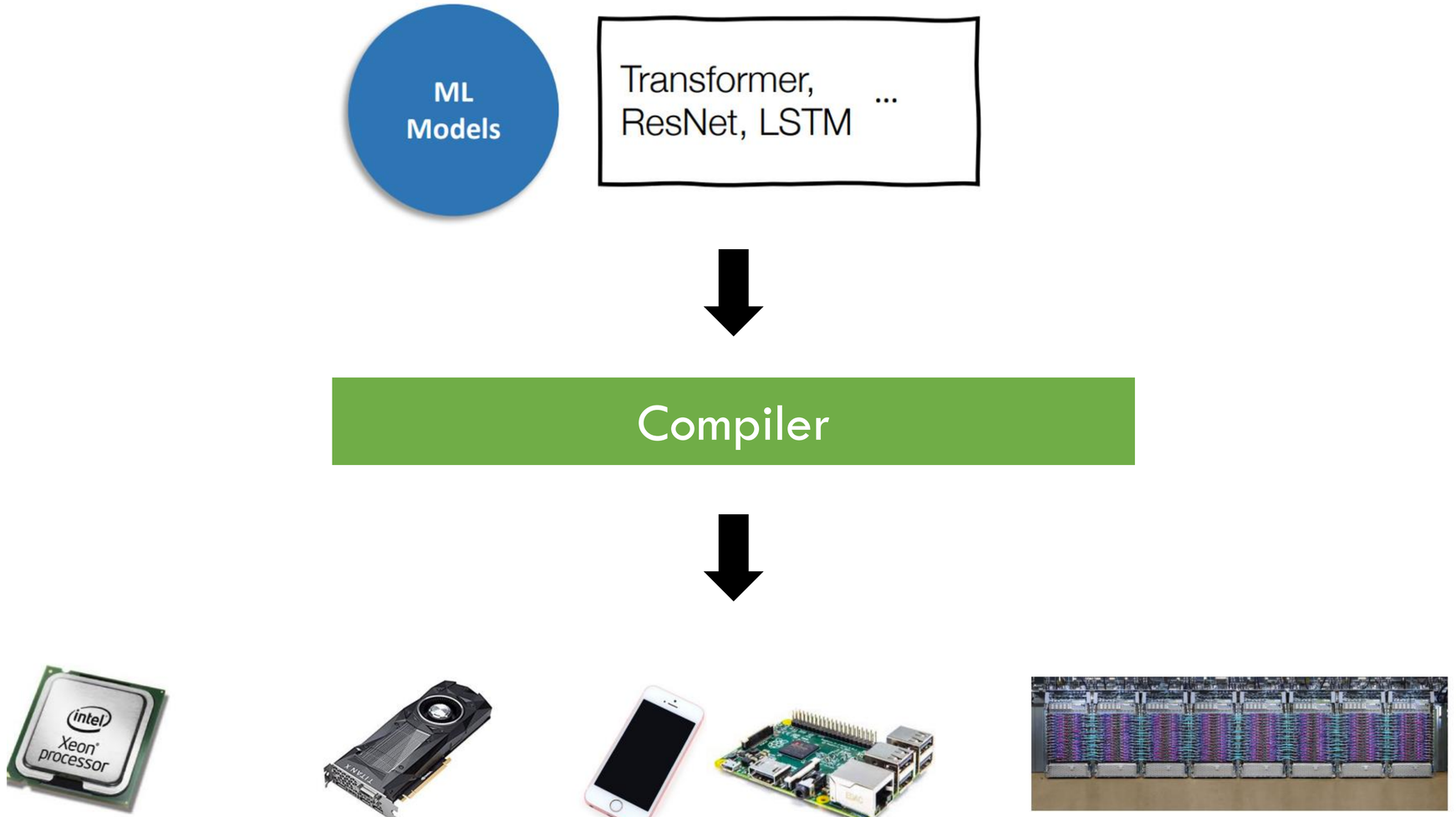
The common theme of MLC is optimization in different forms:

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

ML Compilation Goals

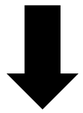


ML Compilation Goals

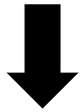


What is a Traditional Compiler?

Human
Code (e.g., cpp)



Compiler



Machine
code

ML
Models

Transformer,
ResNet, LSTM ...

Dataflow Graph

Transformed Dataflow Graph

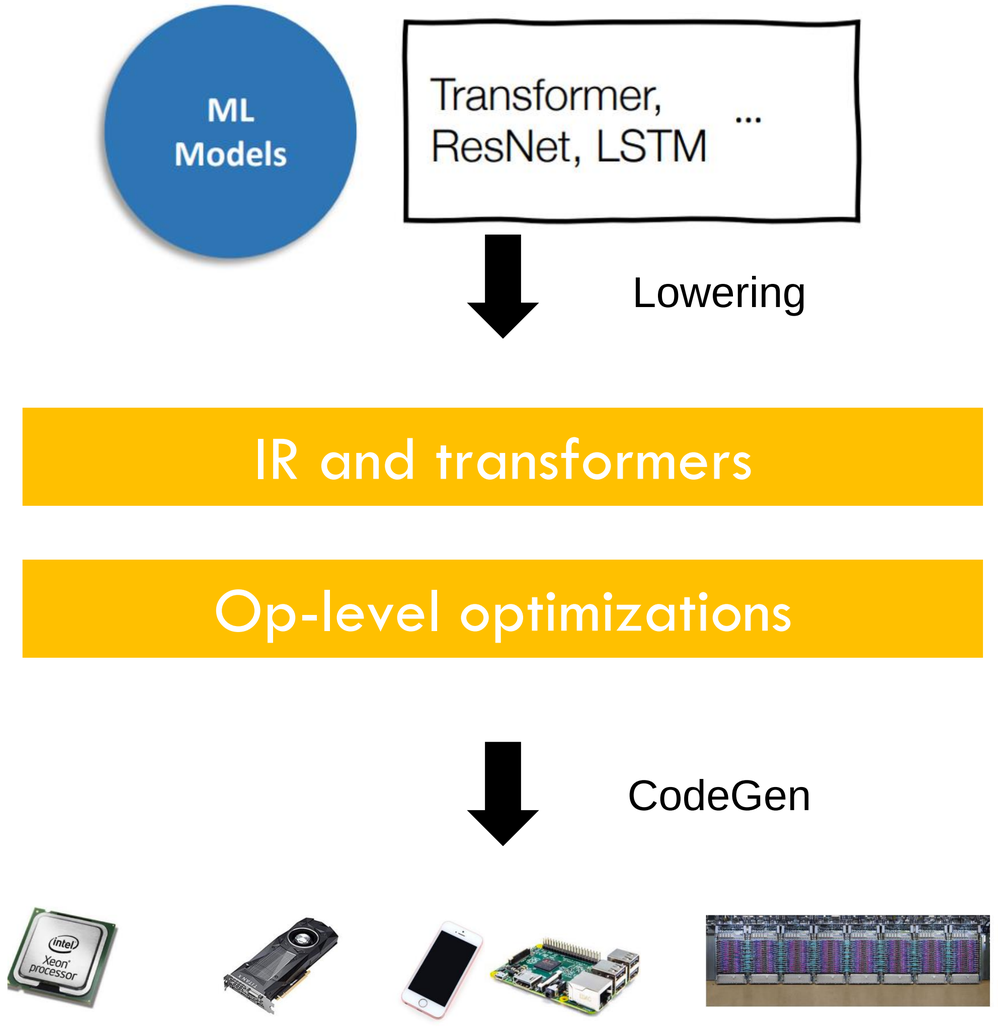
Efficient Kernel code

Machine code

Problems:

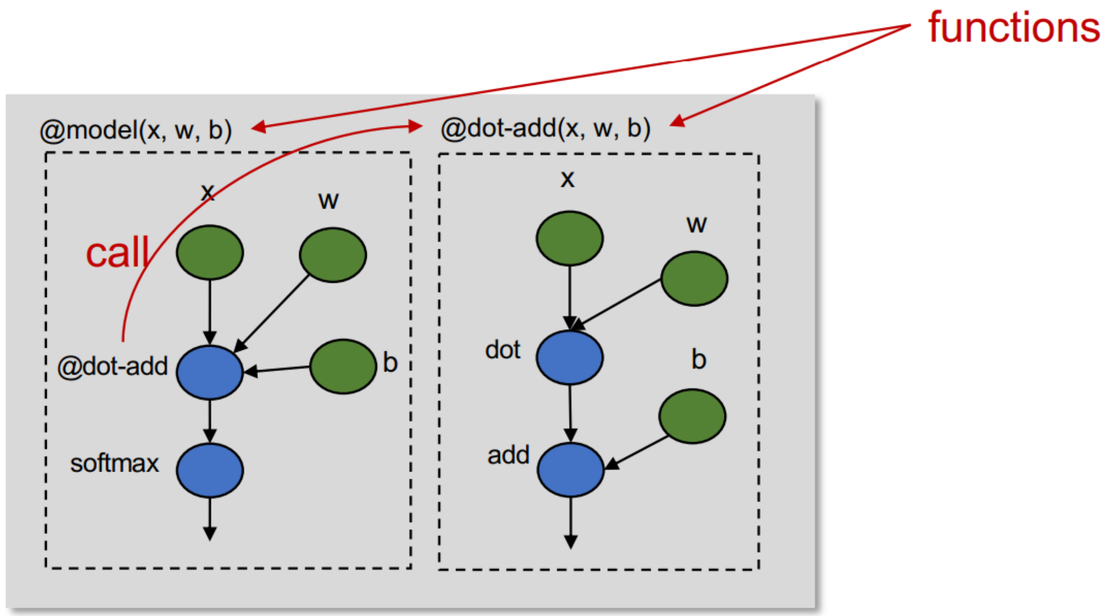
- Op-level: How to make operator fast on different hardware?
  - Tiling Based on register/cache/shared mem sizes
  - Use target device-specific accelerations
  - Generate the operator implementations automatically
- Graph-level: graph transformations to make it faster
- Programming-level:
  - How to 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 if interdependent functions

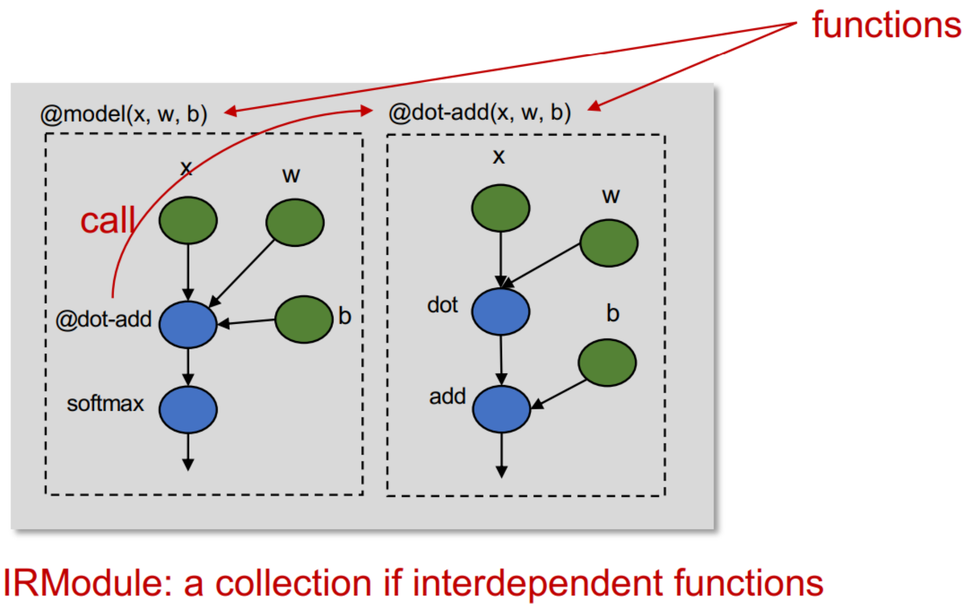
Notable Compilers

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

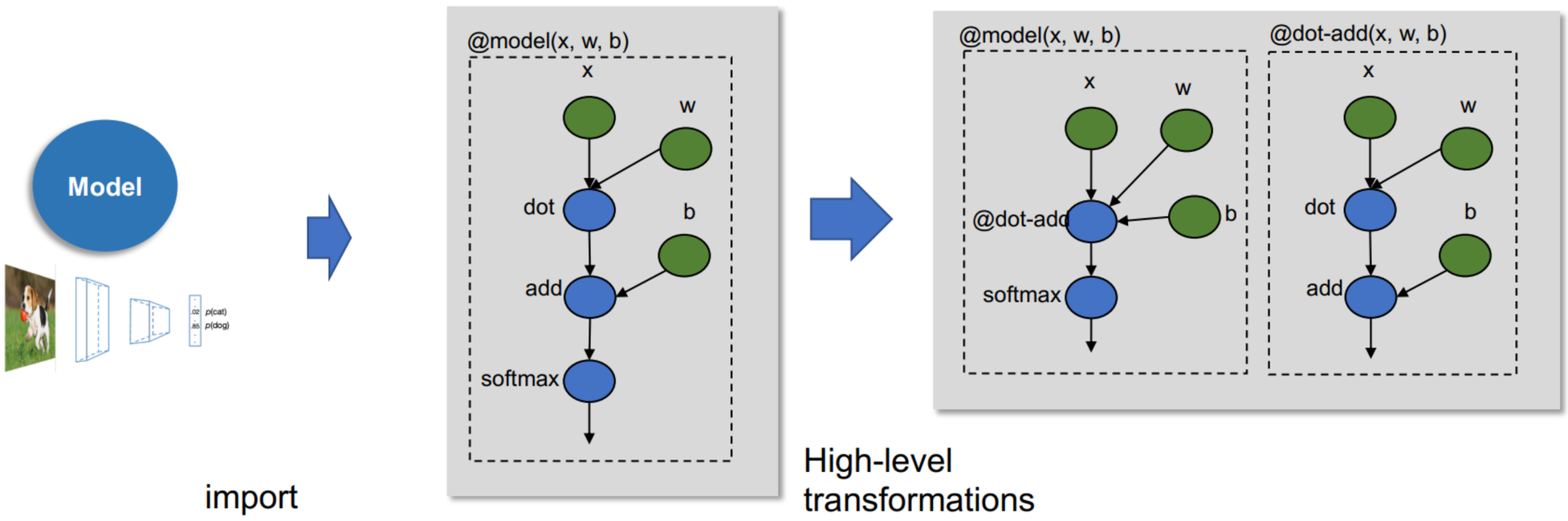
- What are potential challenges of user code parsing?



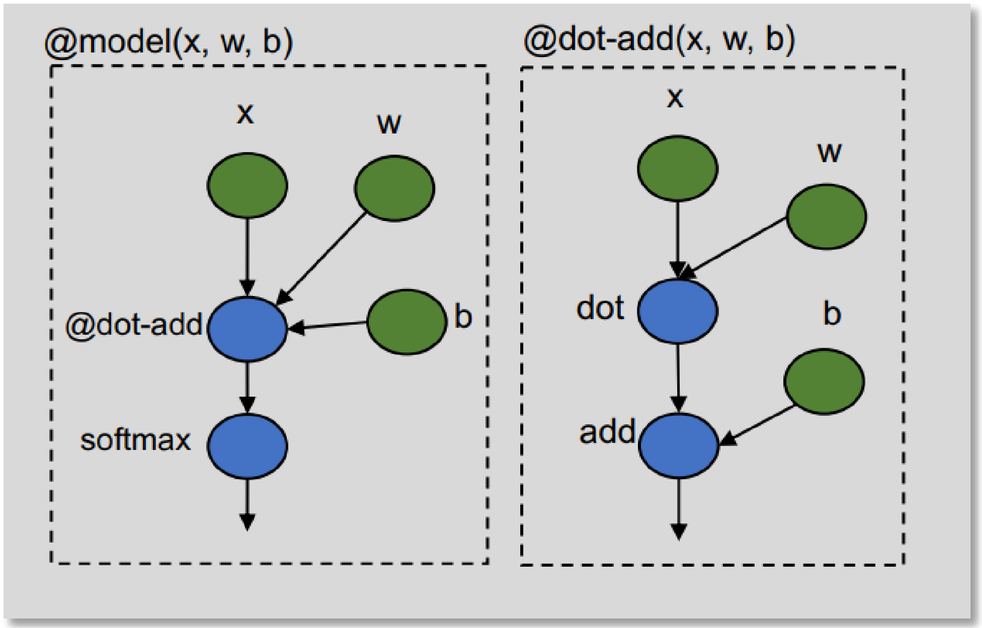
IRModule: a collection if interdependent functions

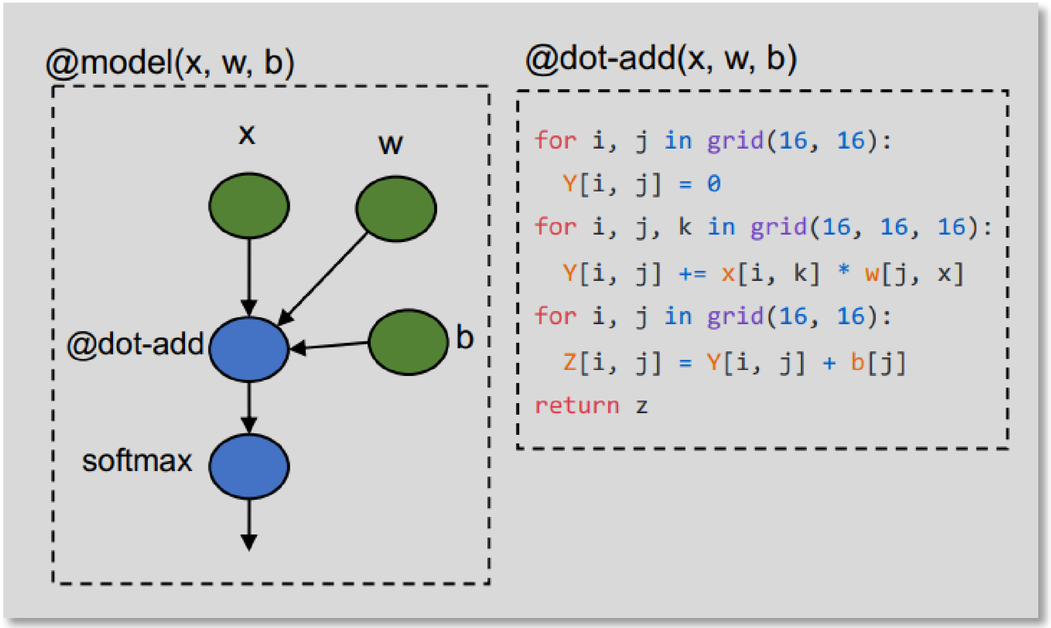
Example Compile flow: high-level transformations

- We'll talk about some techniques here next week

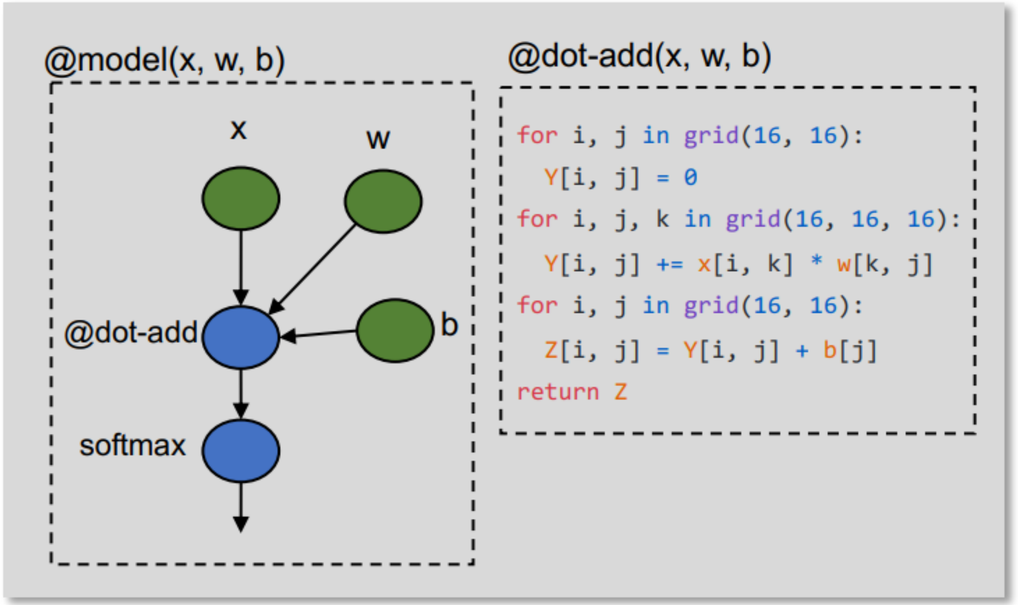


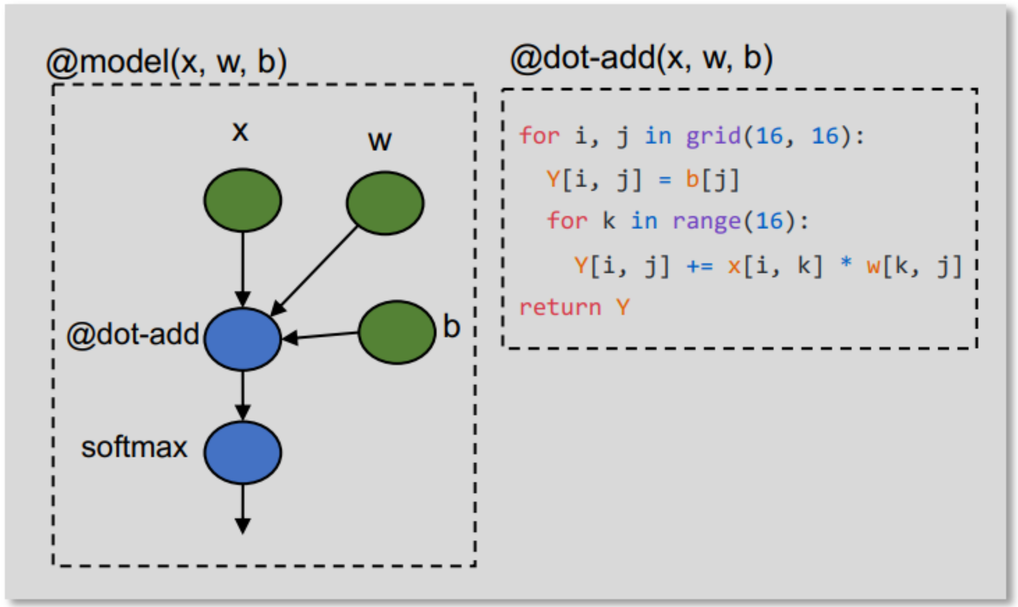
Example Compile flow: lowering to loop IR





Example Compile flow: Loop transformers

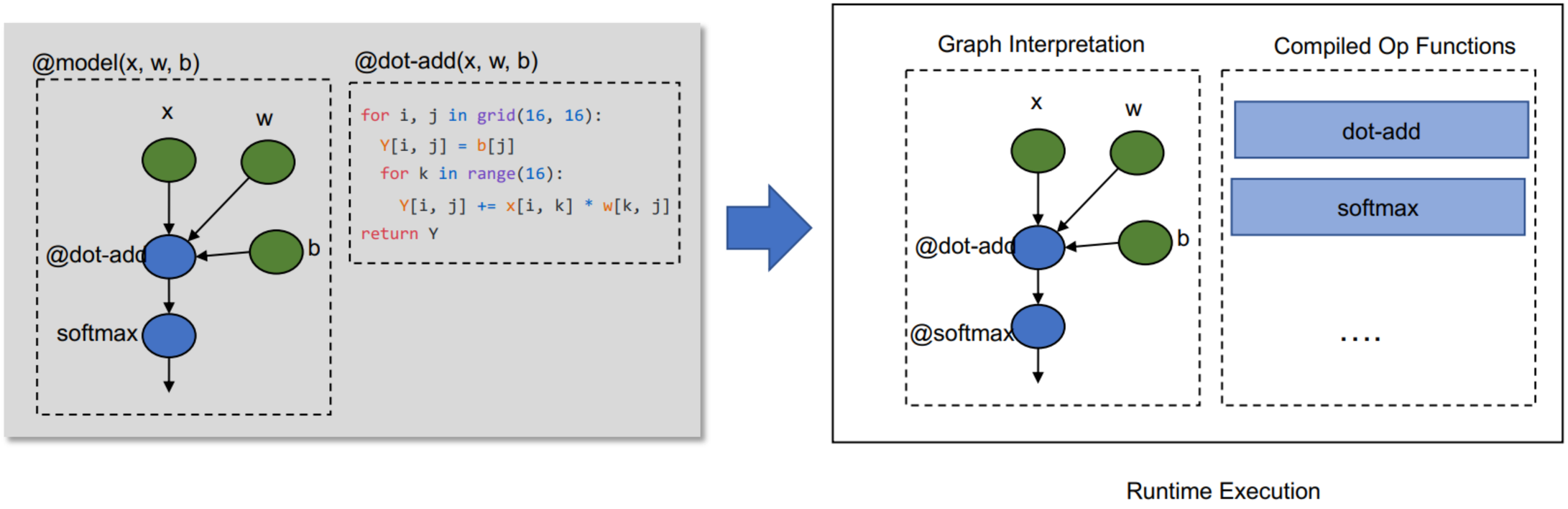




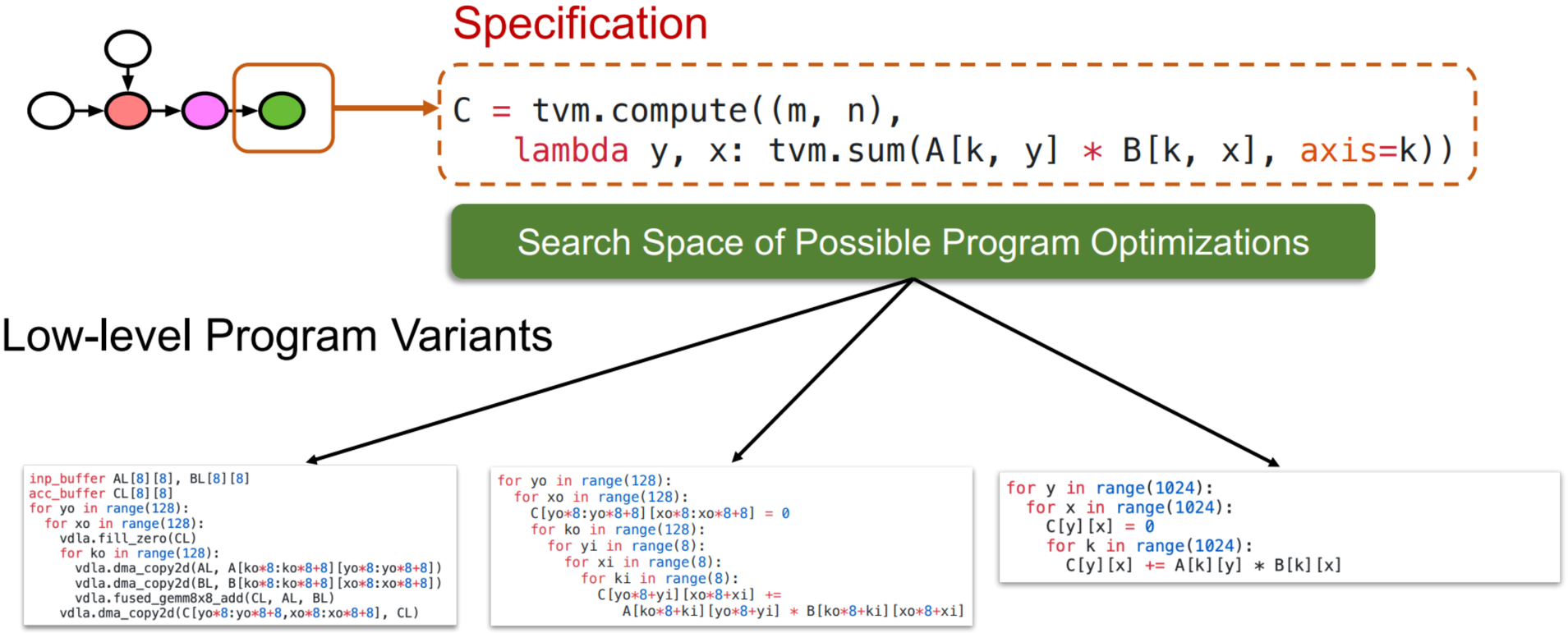
Low-level transformations

Example Compilation: CodeGen

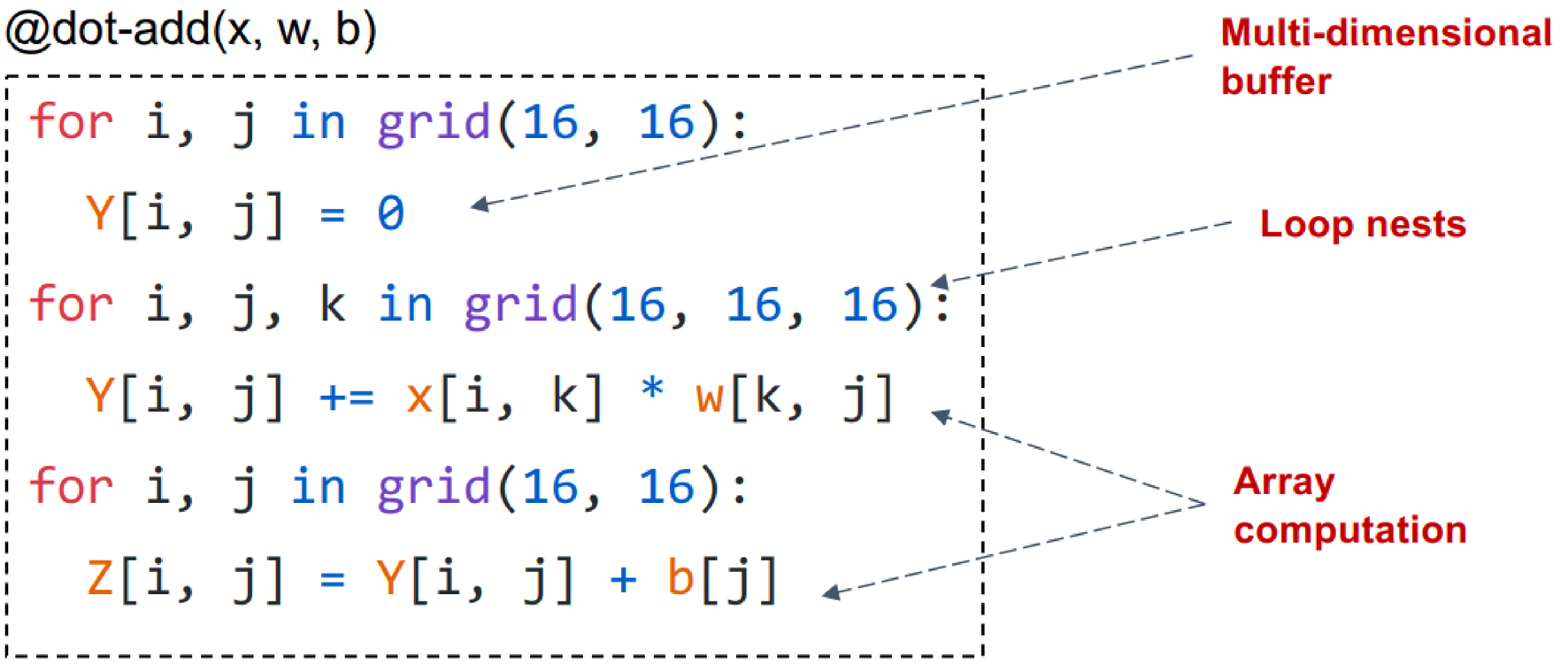
Eventually, we transform a user code into some binary artifacts



Lower-level code optimization



Low-level Loop Representation



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[threadId.x * 4 + blockIdx.x] = . . .
```

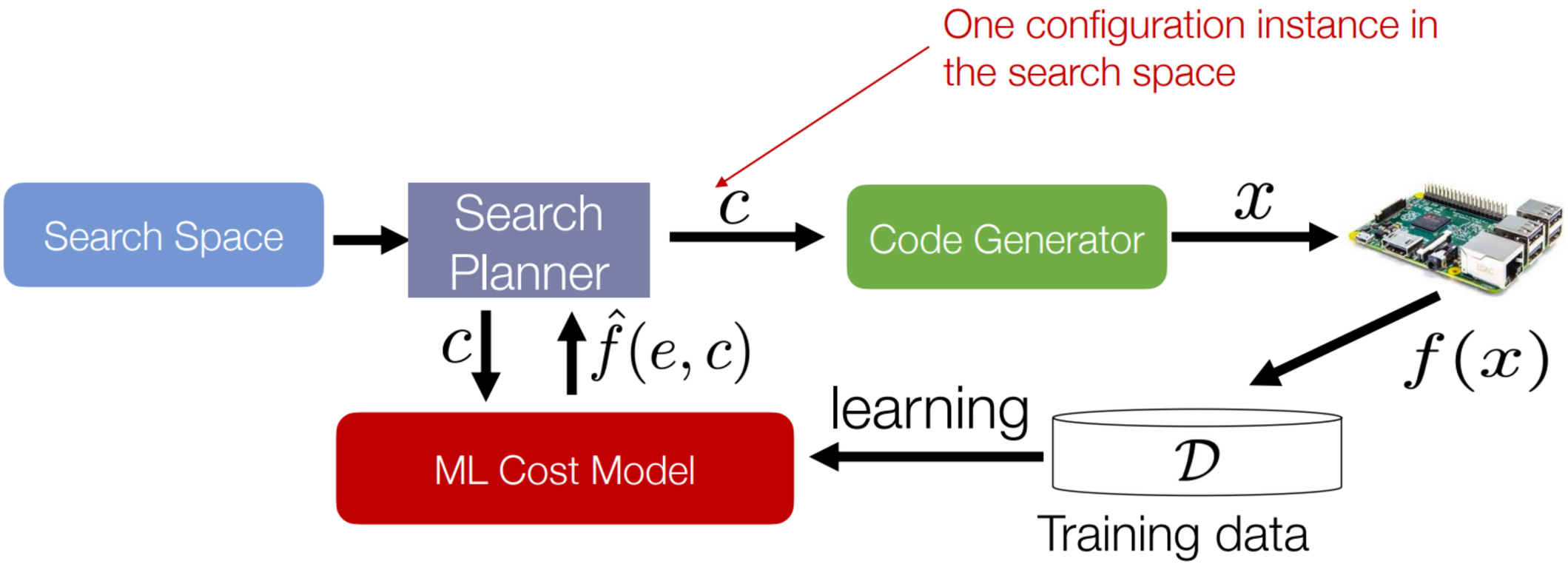
Transformation

```
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

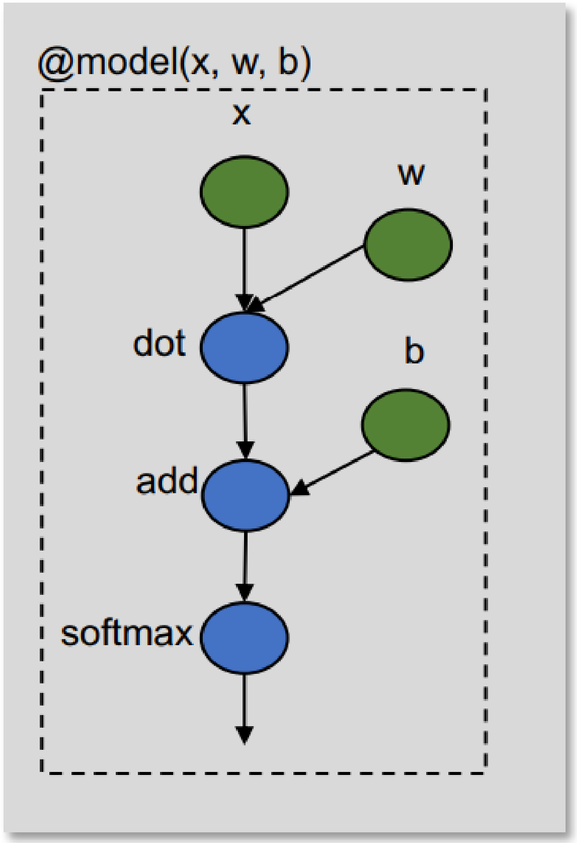
Search via Learned Cost Model



Elements of an automated ML Compiler

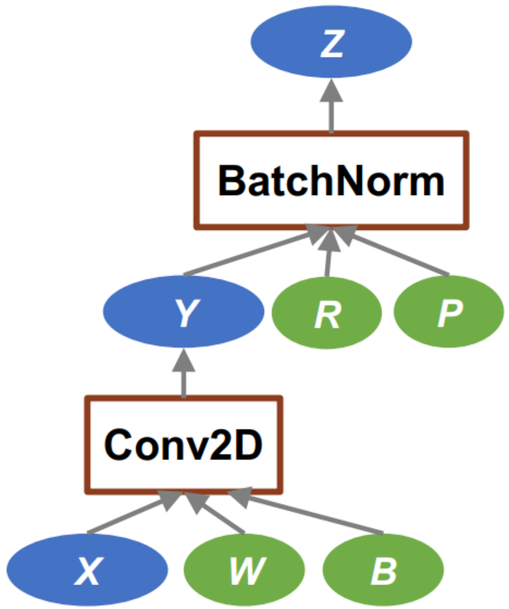
- Program abstraction
  - Represent the program/optimization of interest
- Build Search space through a set of transformations
  - Good coverage of common optimizations like tiling
- Effective Search
  - Accurate cost models
  - Transferability

High-level IR transformations

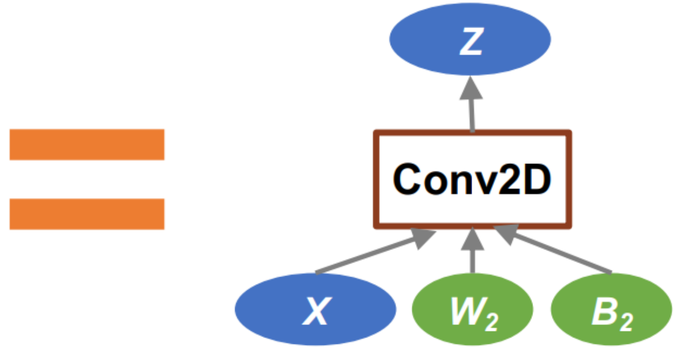


- Graph-like representation
- Each node is a tensor operator
- Can be transformed (e.g. fusion) and annotated (e.g., device placement)
- Most ML frameworks today have this layer

Recall: fusing conv and bn



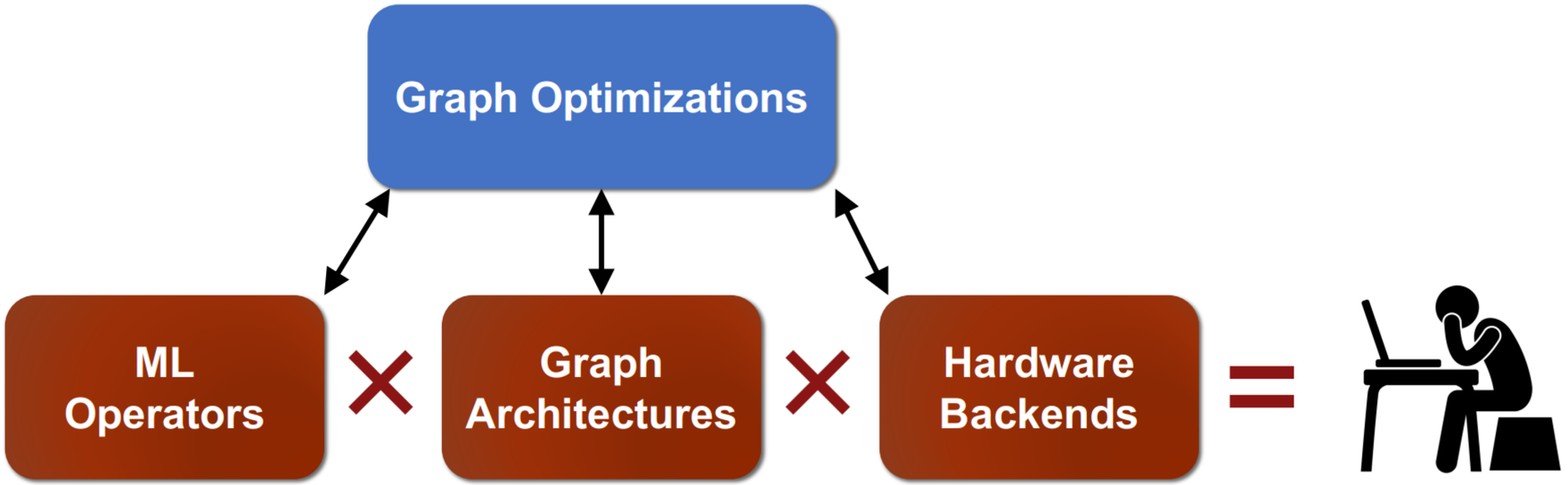
$$Z(n,c,h,w) = \left(\sum_{d,u,v} X(n,d,h+u,w+v) * W_2\,(c,d,u,v) \right) + B_2\,(n,c,h,w)$$



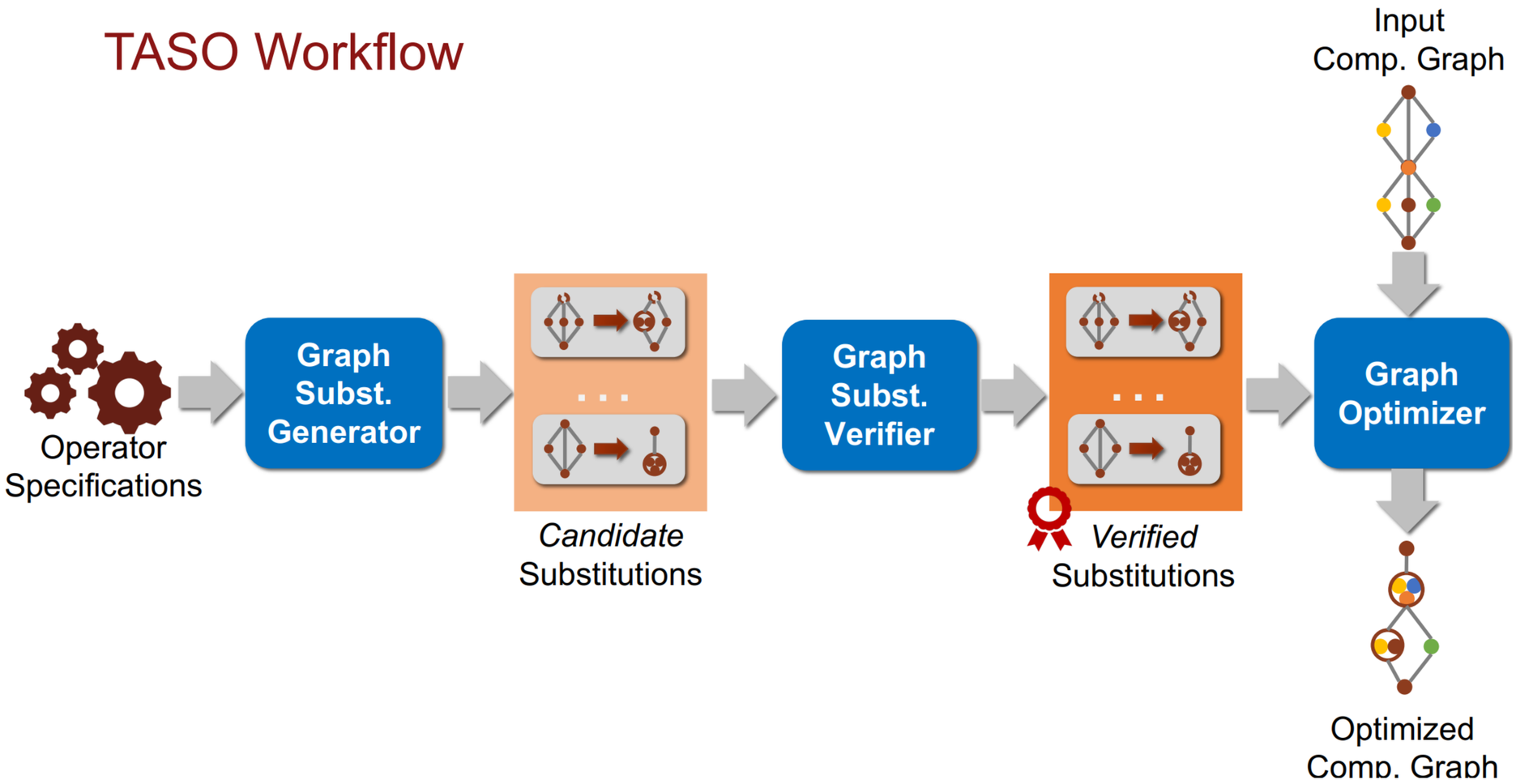
$$W_2(n,c,h,w) = W(n,c,h,w) * R(c)$$

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

Problems of High-level Graph Optimizations



Idea: Enumerate and Verify



Graph Substitution Generator