https://hao-ai-lab.github.io/cse234-w25/

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

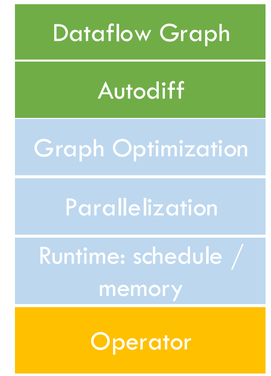
MLSys Basics

Recap of Last Lecture

- GPU Execution: thread hierarchy
  - Bulk launch of many threads
  - Two-level hierarchy: threads are grouped into thread blocks
- Distributed address space
  - Built-in memcpy primitives to copy between host and device address spaces (cudamalloc, cudamemcpy, pinned memory)
- Three different types of device address spaces
  - Per thread, per block (“shared”, SRAM), or per device (“global”, HBM)
- Barrier synchronization primitive for threads in thread block and cpu <-> gpu
- First GPU program: window average (== conv1d)

Today's Learning Goal

- **Case study: Matmul on GPU**
- Operator Compilation
- High-level DSL for CUDA: Triton
- Graph Optimization Starter



Develop the Thought Process when CUDA-ing

Convert your brain to be SIMD:

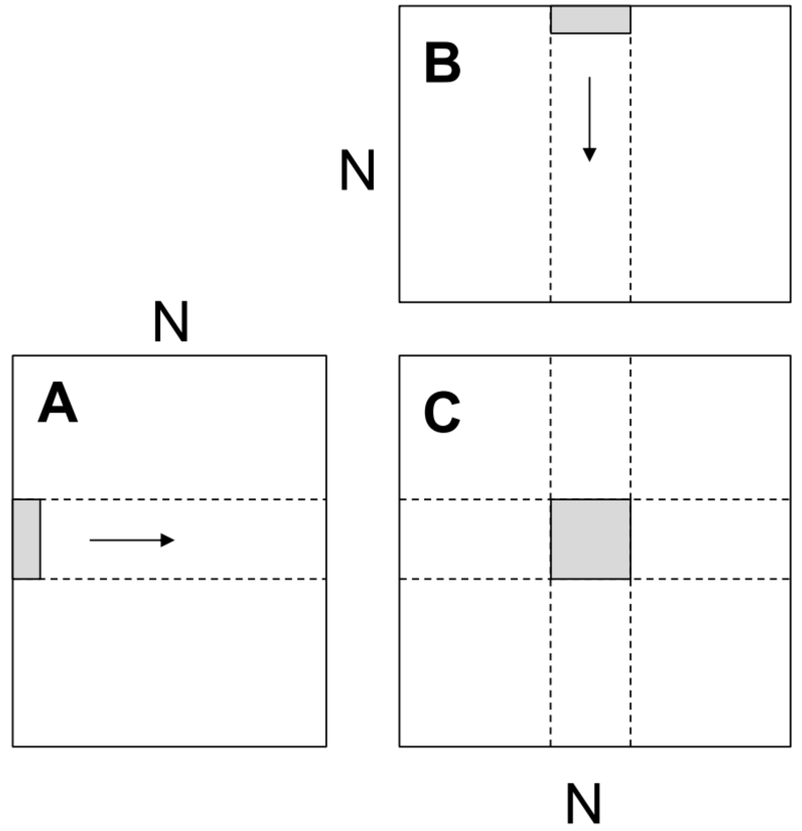
1. Identify work that can be performed in parallel
2. Partition work (and data associated with the work)
3. Manage data access, communication, and synchronization

And make sure

1. Oversubscription: create enough tasks to keep all execution units on a machine busy
2. Mitigate straggler: Balance workload (because GPU cores does not know control flow)
3. Minimize “communication”: reduce I/O across memory hierarchies

Case study: GPU Matmul v1

- C = A x B
- Q: what's the work that can be parallelizec
- 💡Each thread computes one element!

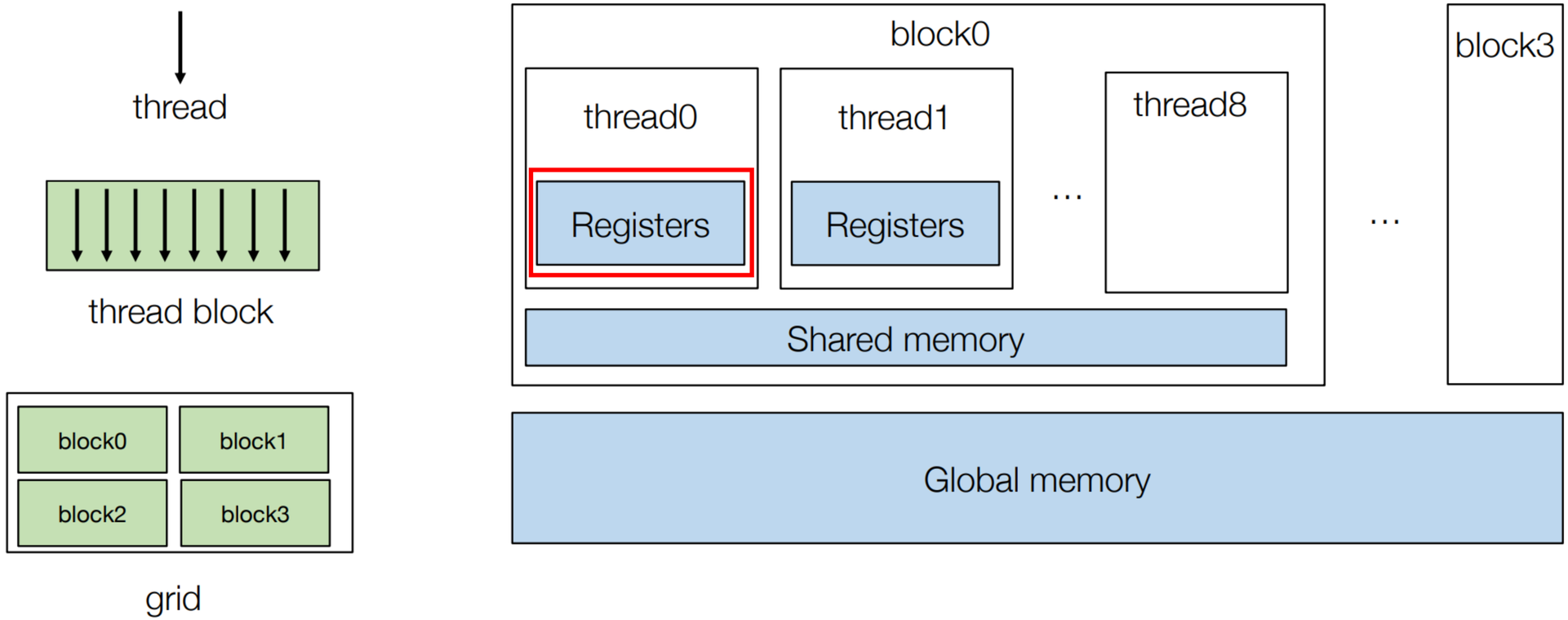



```
int N = 1024;
dim3 threadsPerBlock(32, 32, 1);
dim3 numBlocks(N/32, N/32, 1);

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;
}
```

- Global memory read per thread?
  - N + N = 2N
- # threads?
  - N^2
- Total global memory access?
  - N^2 * 2N = 2N^3
- Memory?
  - 1 float per thread

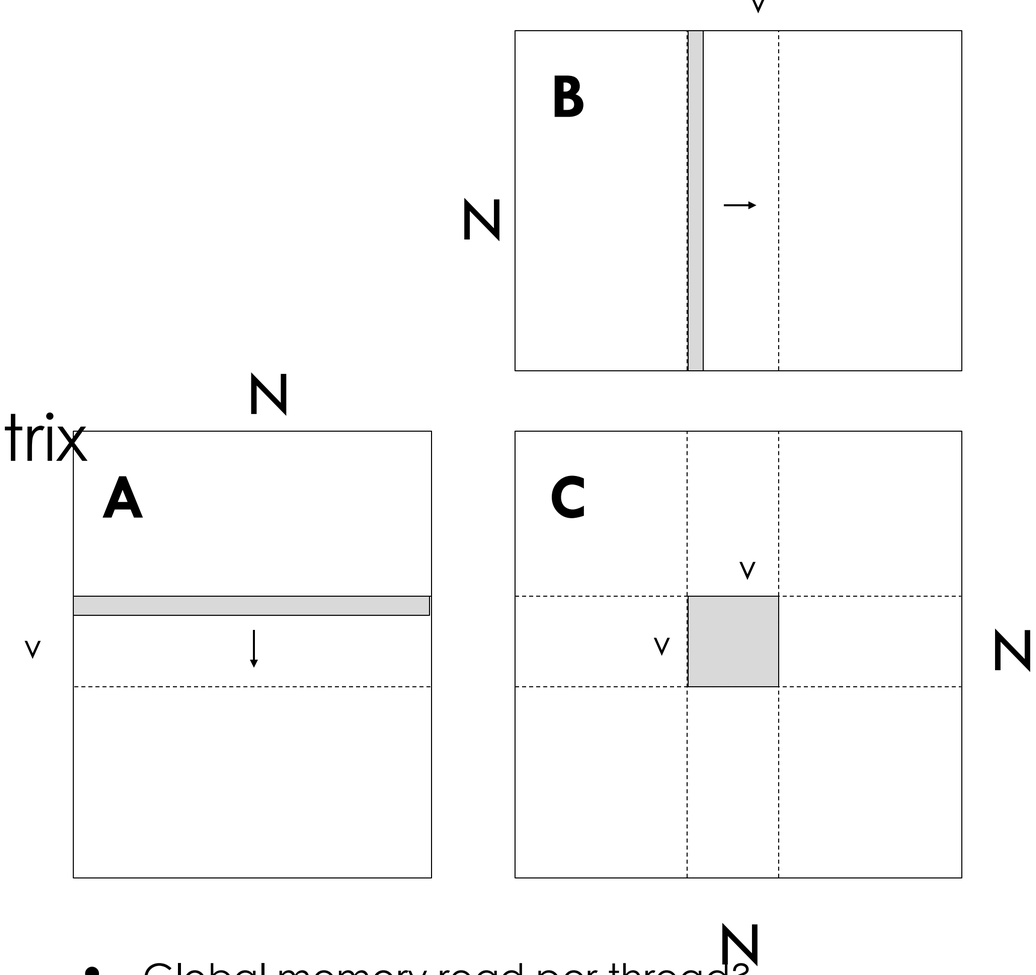
Recall Memory Hierarchy and Register tiling



💡Each thread uses more thread-level registers to compute outputs to save I/o

GPU Matmul v1.5: Thread Tiling

- Each thread computes a VxV submatrix

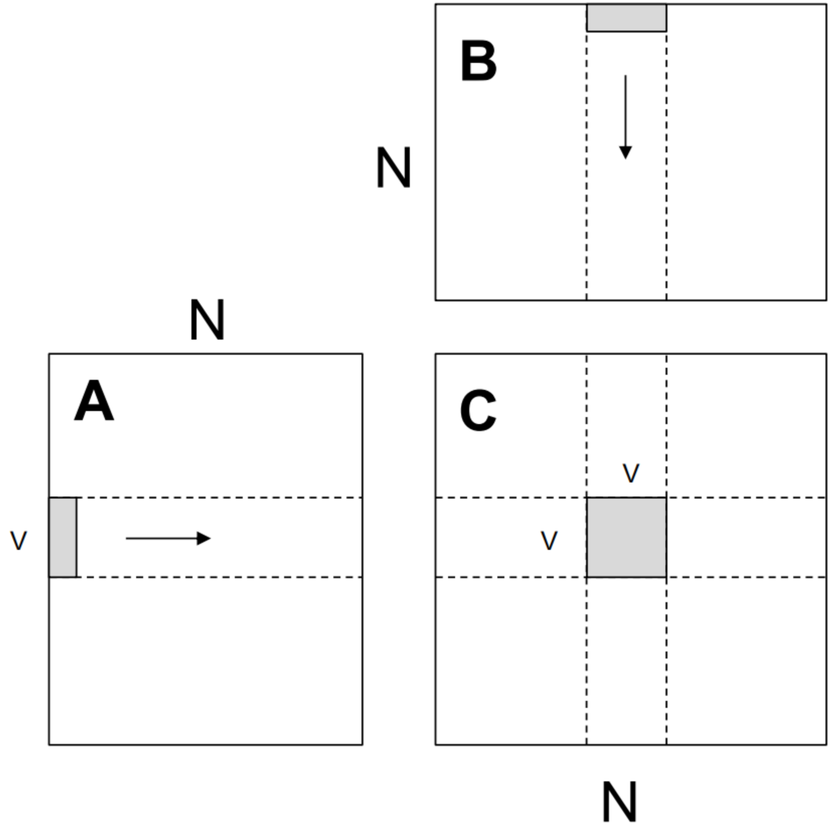



```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

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

- Global memory read per thread?
  - NV + NV^2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * (NV + NV^2) = N^3/V + N^3
- Memory?
  - V^2 + 2N float per thread

GPU Matmul v2: Can we do better?

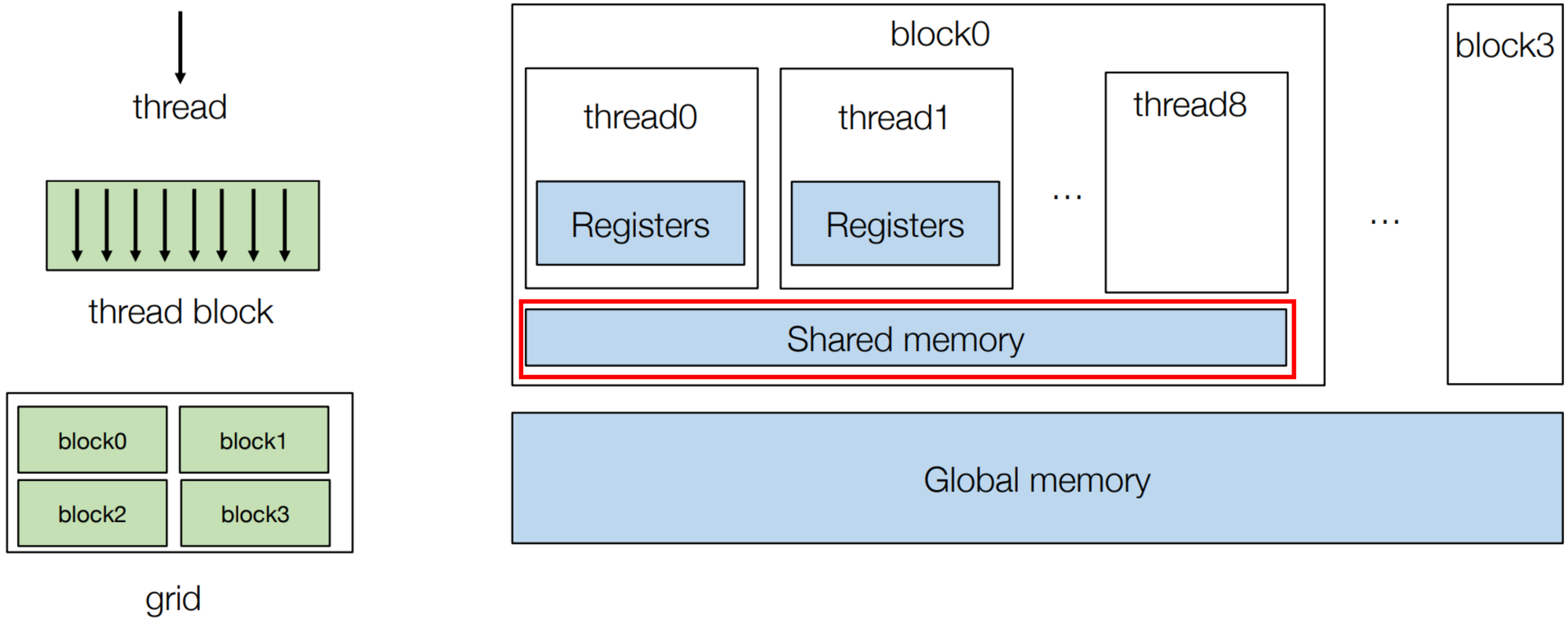
- Each thread computes a VxV submatrix
- 💡compute partial sum: $[X_1, X_2]\begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = X_1Y_1 + X_2Y_2$




```
__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[:];
}
```

- Global memory read per thread?
  - NV * 2
- # threads?
  - N/V * N/V = N^2/V^2
- Total global memory access?
  - N^2 / V^2 * 2NV = 2N^3/V
- Memory?
  - V^2 + 2V float per thread

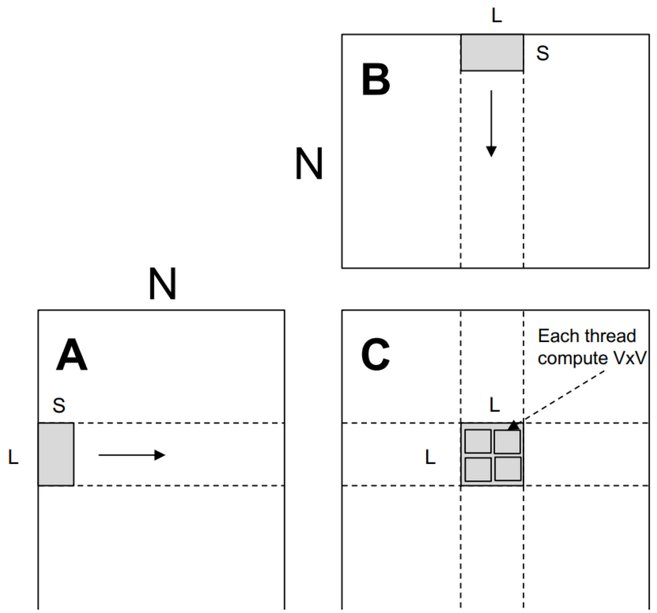
Recall Memory Hierarchy and Cache tiling



💡Try to utilize block-level shared memory (SRAM)

GPU Matmul v3: SRAM Tiling (GPU)

- 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[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + 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[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + 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[:];
}
```

Cooperative Fetching

```
sA[:, :] = A[k : k + S, yblock * L : yblock * L + L];
```

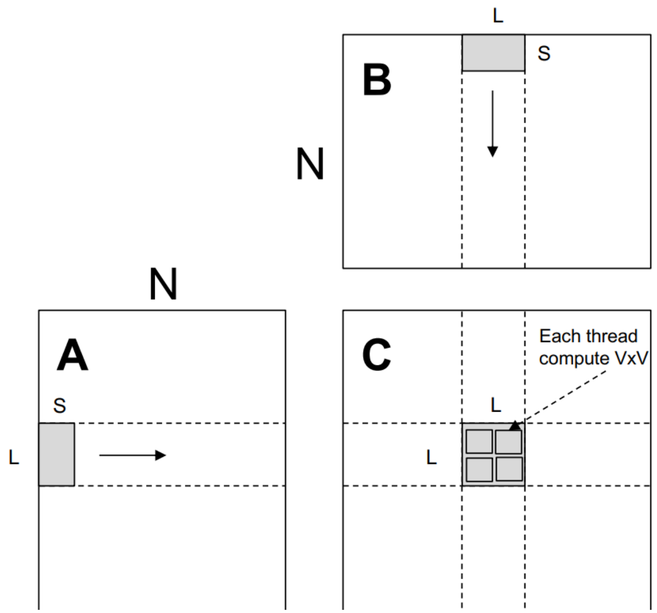
```
int nthreads = blockDim.y * blockDim.x;
int tid = threadIdx.y * blockDim.x + threadIdx.x;

for(int j = 0; j < L * S / nthreads; ++j) {
  int y = (j * nthreads + tid) / L;
  int x = (j * nthreads + tid) % L;
  s[y, x] = A[k + y, yblock * L + x];
}
```

Many More GPU Optimizations

- Global memory continuous read
- Shared memory bank conflict
- Pipelining
- Tensor core
- Lower precision

Core Problems Here

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




```
__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[ko : ko + S, yblock * L : yblock * L + L];
    sB[:, :] = B[ko : ko + 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[:];
}
```

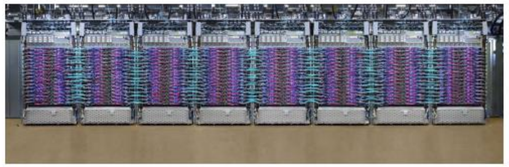
In Reality

MKL-DNN

cuDNN

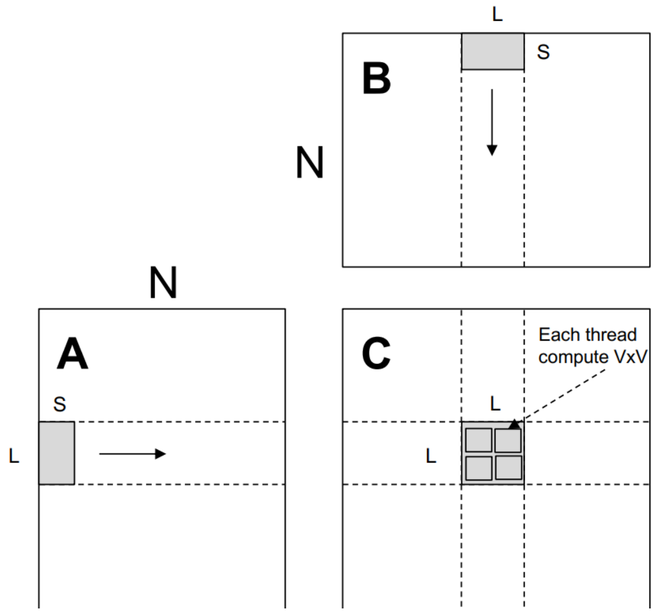
ARM-Compute

TPU Backends



Back to Today's Problem

- How to implement an highly efficient kernel
- How to choose configs.
  - #threads
  - #registers
  - Amount of SRAM



- Solution 1:
  - expert-craft -> Enumerate configs -> profile
- Solution 2: Operator compilation

Introduce ML Compilation: Big Picture

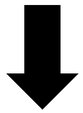
ML compilation's Promise:

Automatically generate *optimal* configurations and code

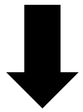
given users code and target hardware

Traditional vs. ML Compiler

Human
Code (e.g., cpp)



Compiler



Machine code

ML
Models

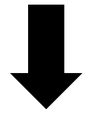
Transformer,
ResNet, LSTM ...

Dataflow Graph

Transformed Dataflow Graph

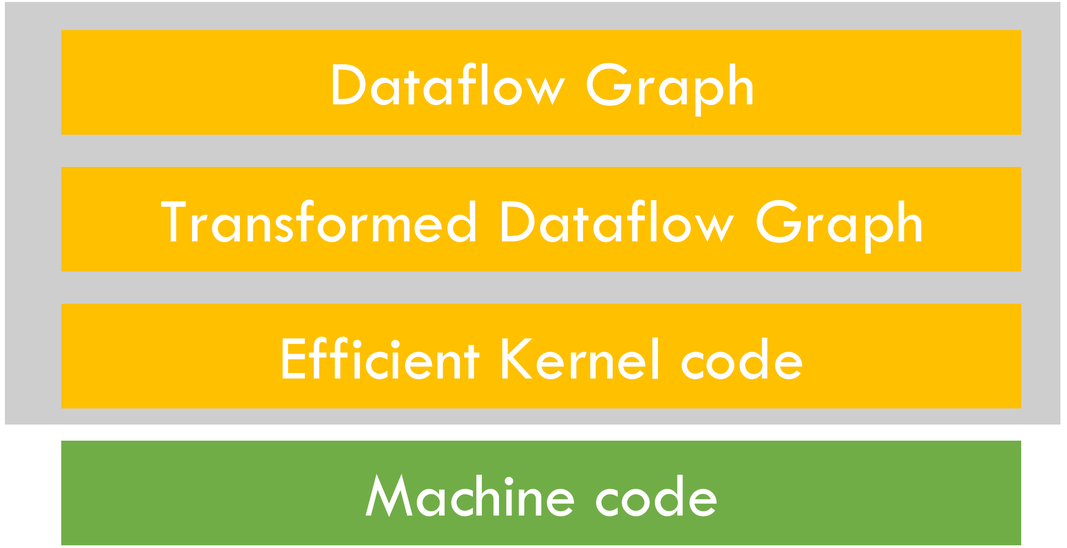
Efficient Kernel code

Machine code

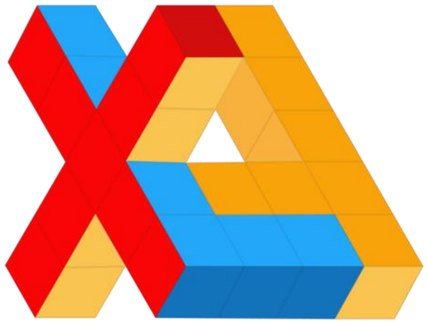


Grand Problems:

- Programming-level:
  - Automatically transform an arbitrary (usually imperative) code (by developers) into a compile-able code (e.g., static dataflow graph)?
- Graph-level:
  - Automatic graph transformations to make it faster
- Op-level:
  - How to make operator fast on different hardware?



Notable Compilers

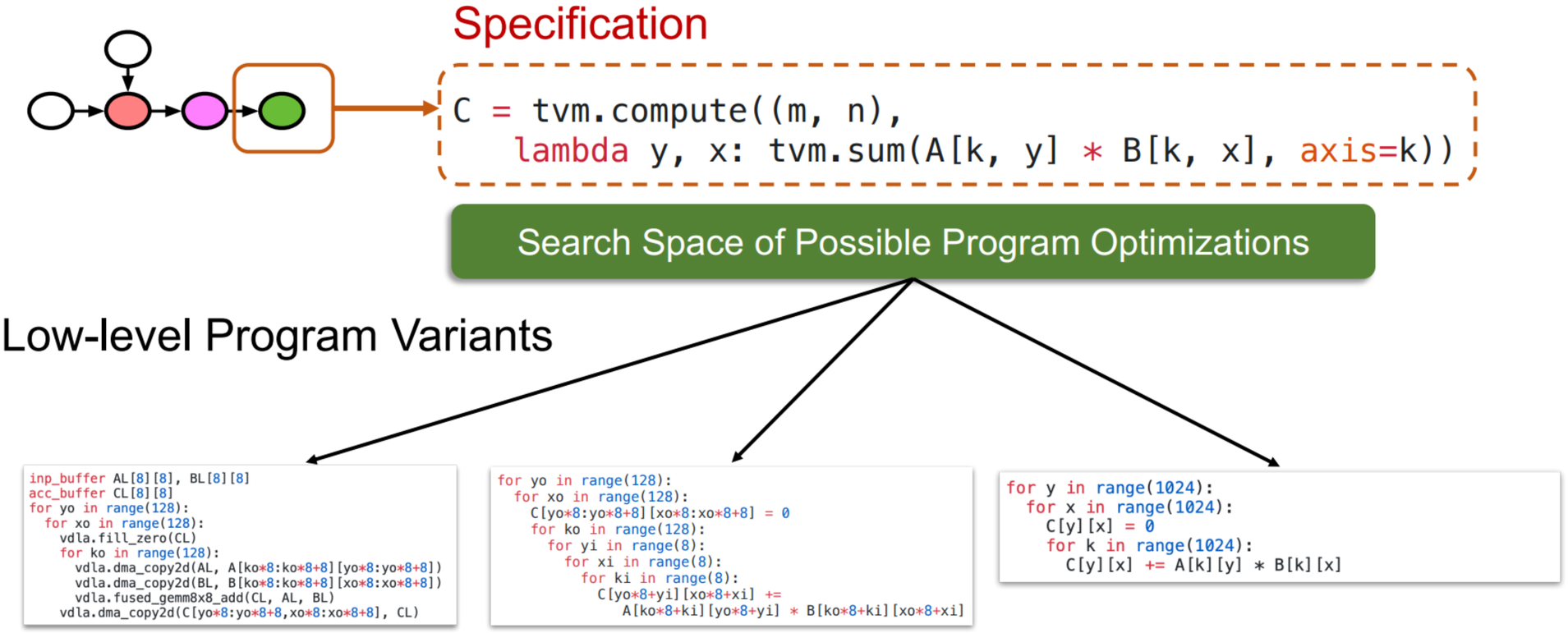


tvm

2.0

Modular

Operator Compilation

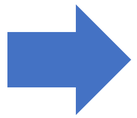


Transforming Loops: Loop Splitting

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]
```



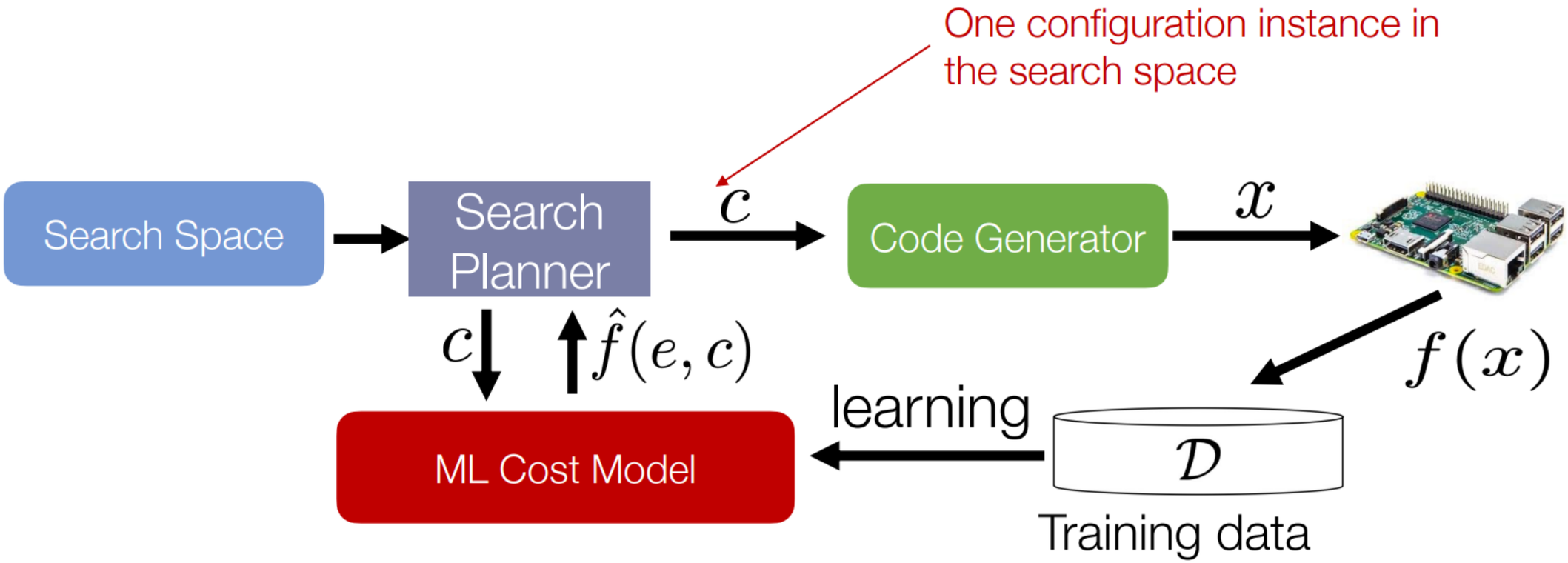
```
for xi in range(4):
  for xo in range(32):
    C[xo * 4 + xi]
      = A[xo * 4 + xi] + B[xo * 4 + xi]
```

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

Problems

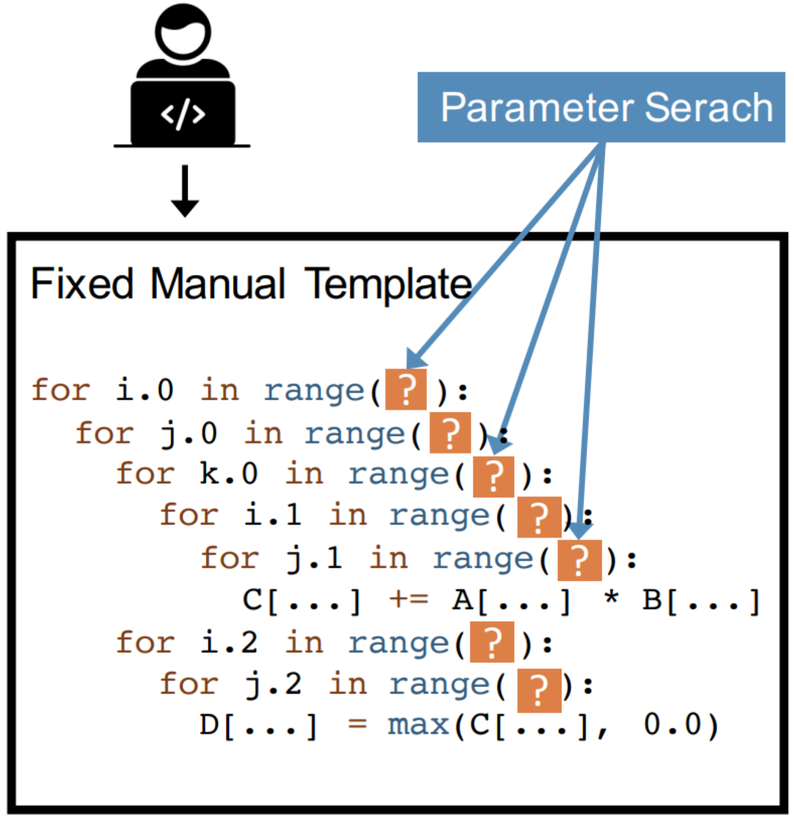
- We need to enumerate many possibilities
  - How to represent all “possibilities”
- We need to find the (close-to-)optimal values (register/cache sizes)
  - How to search?
- We need to apply this to so many operators and devices
  - How to reduce search space
  - How to generalize?

Search via Learned Cost Model



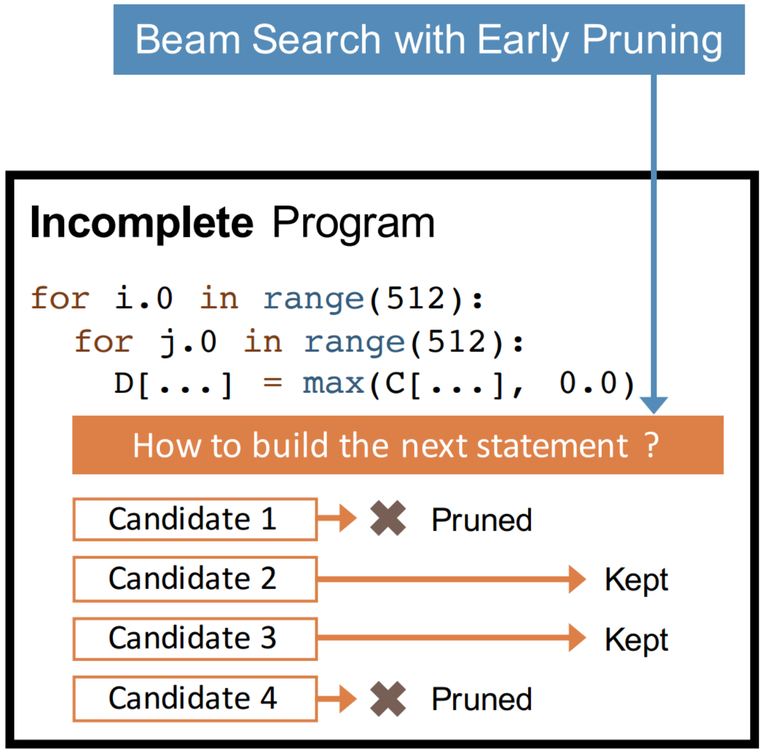
Search Space Definition e.g. Template based

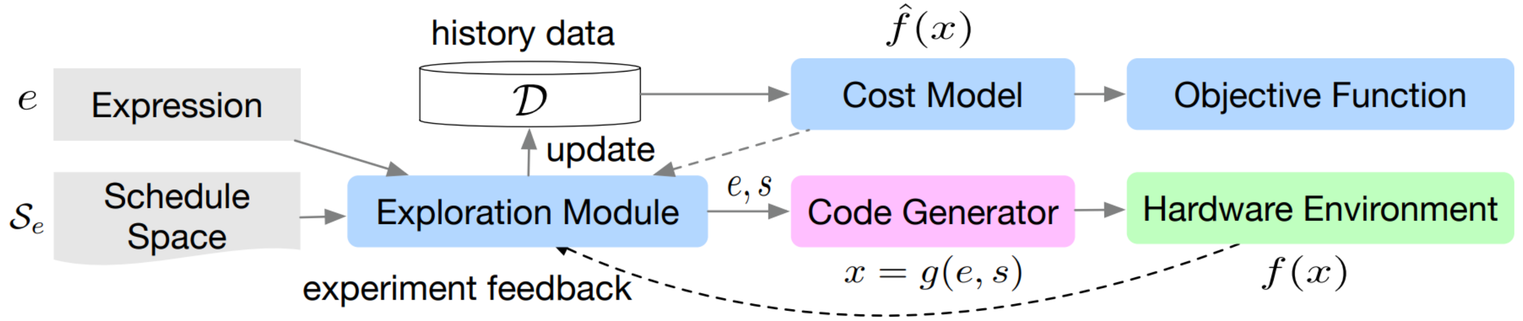
- Issue: still need experts to write templates



How to Search

- Sequential Construction using Early pruning
- 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

Discussion

ML compilation's Promise:

Automatically generate *optimal* configurations and code

given users ML code on target hardware

Q: How well are ML compilers delivering their promises?

Today's Learning Goal

- Case study: Matmul on GPU
- Operator Compilation
- **High-level DSL for CUDA: Triton**
- Graph Optimization Starter

Device-specific DSL (e.g., CUDA) vs. Compiler

+ developers can do whatever the heck they want:

- squeeze the last bits of performance
- use whatever data-structure you want

-- developers can do whatever the heck they want:

- Require deep expertise; performance optimization is very time-consuming
- Codebases are complex and hard to maintain

+ Very fast iteration speed for developers

- Can prototype ideas quickly and give it to compiler

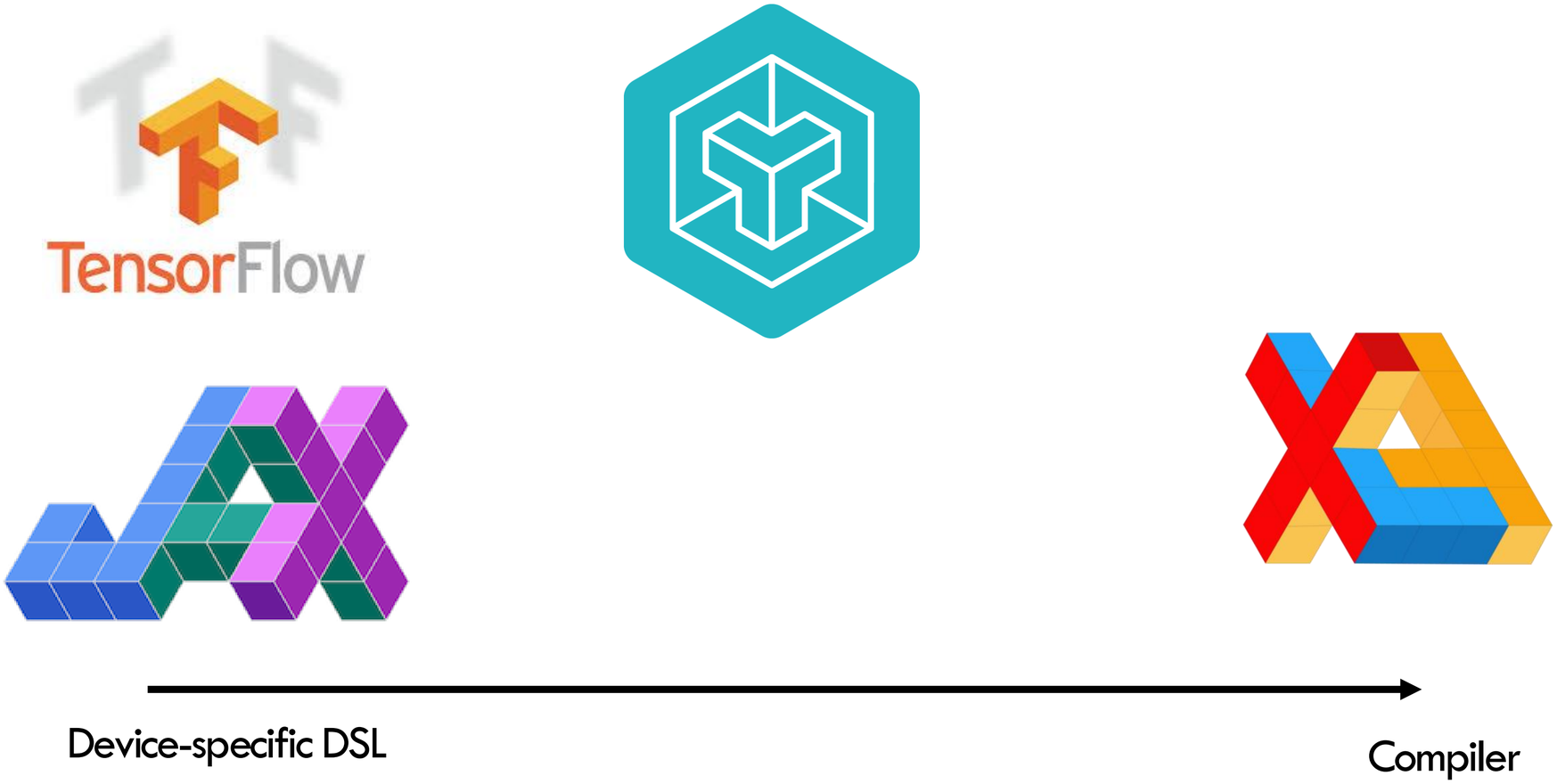
-- Cannot represent certain types of ideas

- In-operator control flow
- Custom data structure

-- Code generation is a old difficult problem

- heavy use of templates and pattern-matching
- lots of performance cliffs

High Level Idea of Triton



High Level Idea of Triton

+ simpler than CUDA; more expressive than graph compilers:

-- less expressive than CUDA; more complicated than graph compilers;

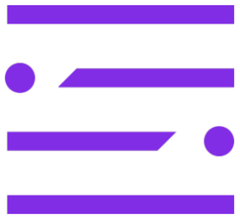
Triton Programming Model

- Users define tensors in SARM, and modify them using torch-like primitives

Embedded in Python

Kernels are defined in Python using triton.jit

Pointer arithmetics



Users construct tensors of pointers and (de)reference them elementwise

Shape Constraints

Must have power-of-two number of elements along each dimension

Example: elementwise add v1 (z = x + y)

- Triton kernel will be mapped to a single block (SM) of threads
- Users will be responsible for mapping to multiple blocks

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arrange
  offsets = tl.arange(0, 1024)
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs)
  y = tl.load(y_ptrs)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 1024
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = (1, )
_add[grid](z, x, y, N)
```

Example: elementwise add v2 (z = x + y)

- Use multiple blocks
  - Index the block and apply offs
- Adds bound check

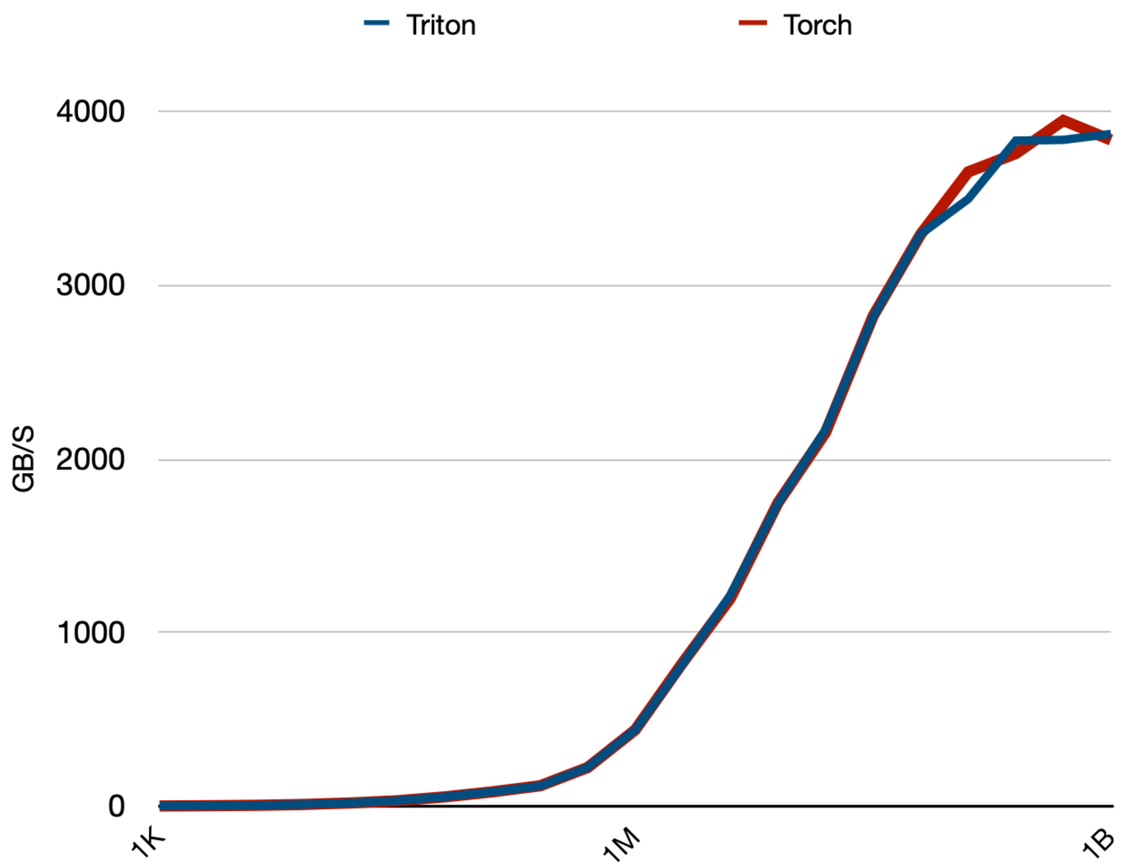
```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
  # same as torch.arrange
  offsets = tl.arange(0, 1024)
  offsets += tl.program_id(0)*1024
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 192311
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

Example: elementwise add v2 (z = x + y)

- Parametrize block size
- Why we do this?
  - Triton will do tiling for users
  - Avoid manipulating loops

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, BLOCK: tl.constexpr):
  # same as torch.arrange
  offsets = tl.arange(0, BLOCK)
  offsets += tl.program_id(0)*BLOCK
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 192311
x = torch.randn(N, device='cuda’)
y = torch.randn(N, device='cuda’)
z = torch.randn(N, device='cuda’)
grid = lambda args: (triton.cdiv(N, args[‘BLOCK’]), )
_add[grid](z, x, y, N)
```

Elementwise Add Performance

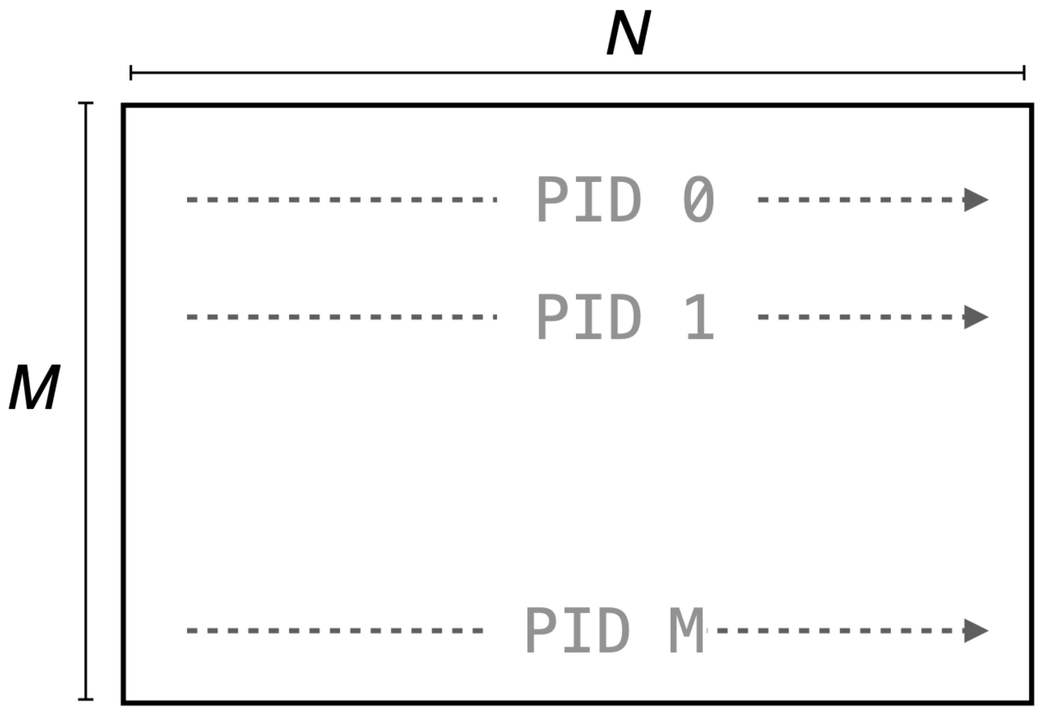


Another Example: Softmax

$$y_i = softmax(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum e^{x_d}}$$

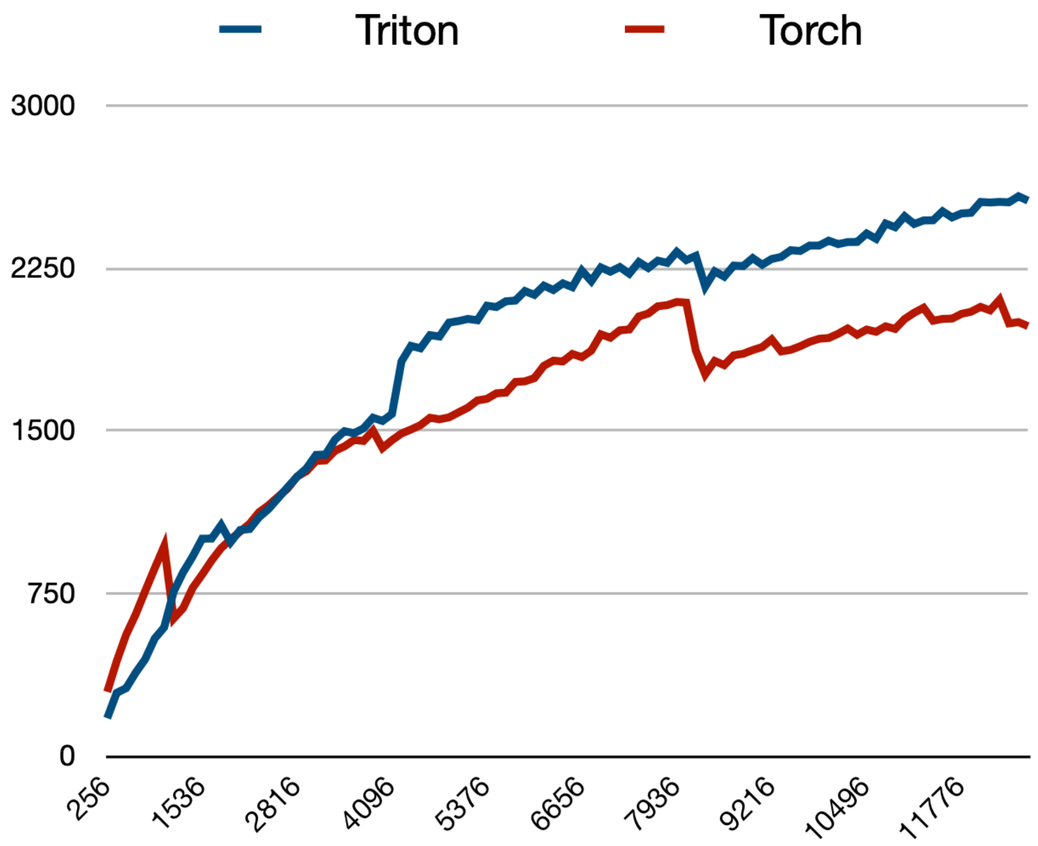
- How did you implement this in PA1?
  - Think about the potential overhead when compose softmax from primitives
- What if implementing an end-to-end softmax kernel
  - Think about the complexity of implementing in CUDA

Triton Example: softmax

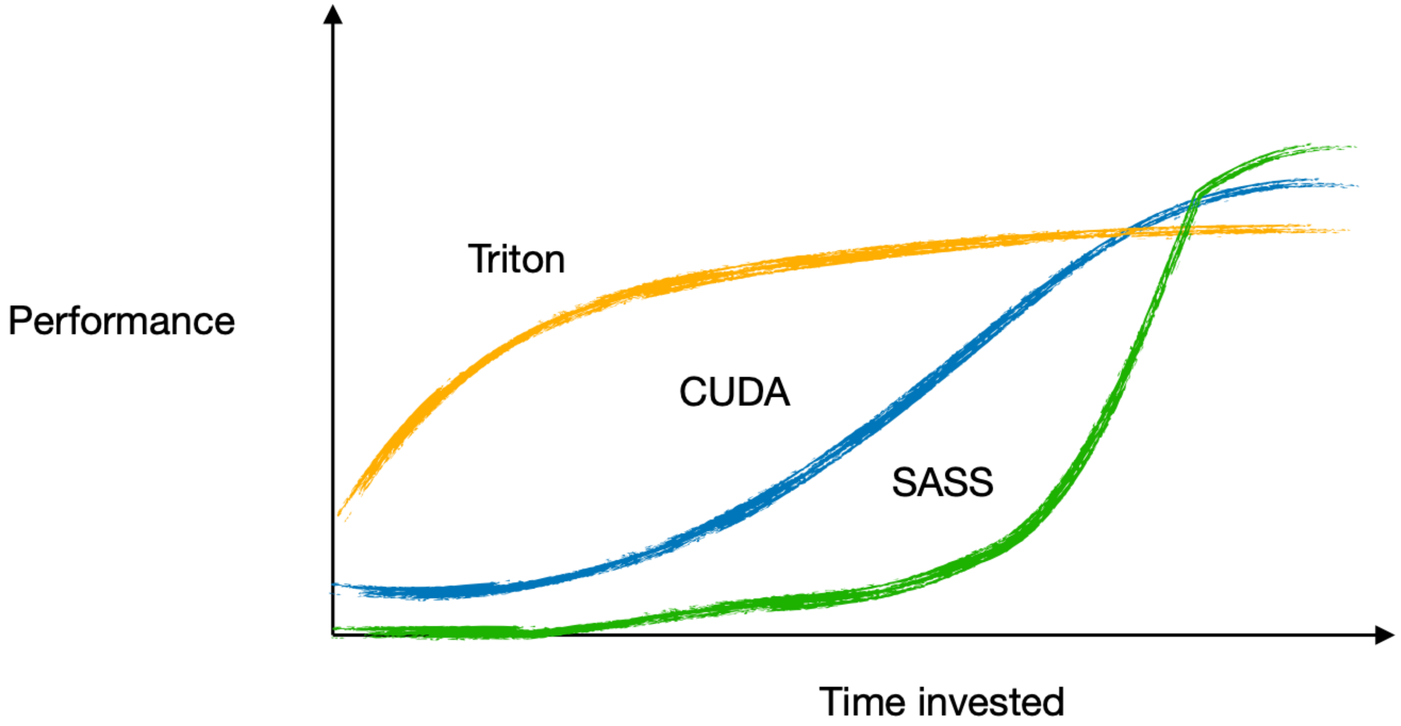



```
import triton.language as tl
Import triton

@triton.jit
def _softmax(z_ptr, x_ptr, stride, N, BLOCK: tl.constexpr):
  # Each program instance normalizes a row
  row = tl.program_id(0)
  cols = tl.arange(0, BLOCK)

  # Load a row of row-major X to SRAM
  x_ptrs = x_ptr + row*stride + cols
  x = tl.load(x_ptrs, mask = cols < N, other = float('-inf'))

  # Normalization in SRAM, in FP32
  x = x.to(tl.float32)
  x = x - tl.max(x, axis=0)
  num = tl.exp(x)
  den = tl.sum(num, axis=0)
  z = num / den;
  # Write-back to HBM
  tl.store(z_ptr + row*stride + cols, z, mask = cols < N)
```

Performance



Revisit Main Arguments of Triton

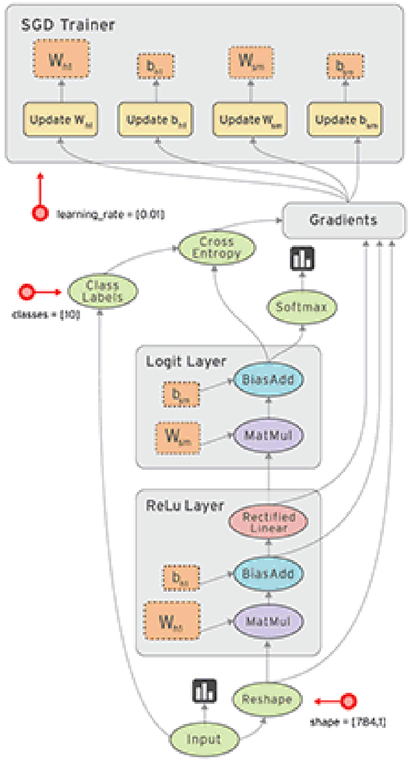


Wrapping Up Operator Optimization

- Goal: to make individual operator run fast on diverse devices

1. General ways: vectorization, data layout, etc.
2. Matmul-specific: tiling to use fast memory
3. Parallelization SIMD using accelerators
4. Handcrafted operator kernels vs. automatically compile code

Wrapping Up Operator Optimization



Dataflow Graph

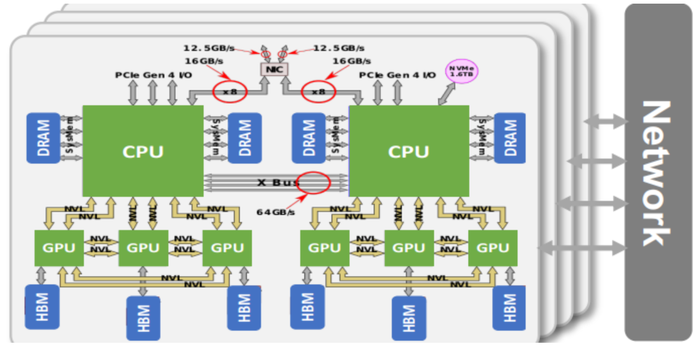
Autodiff

Graph Optimization

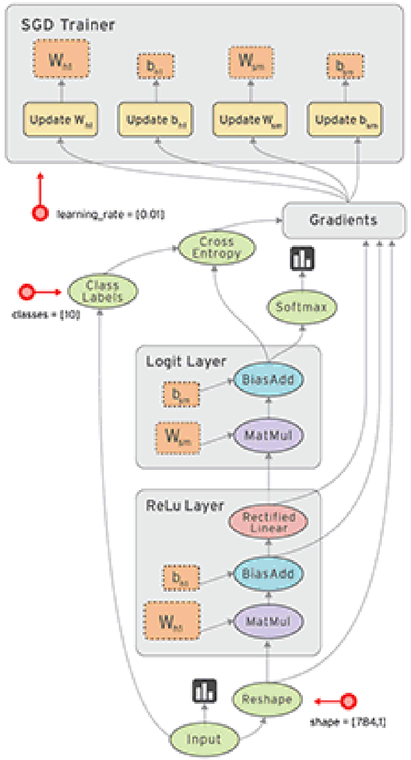
Parallelization

Runtime: schedule / memory

Operator optimization/compilation



Next: Graph Optimization



Dataflow Graph

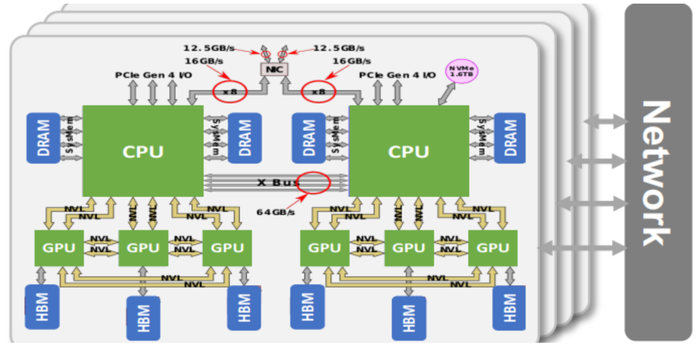
Autodiff

Graph Optimization

Parallelization

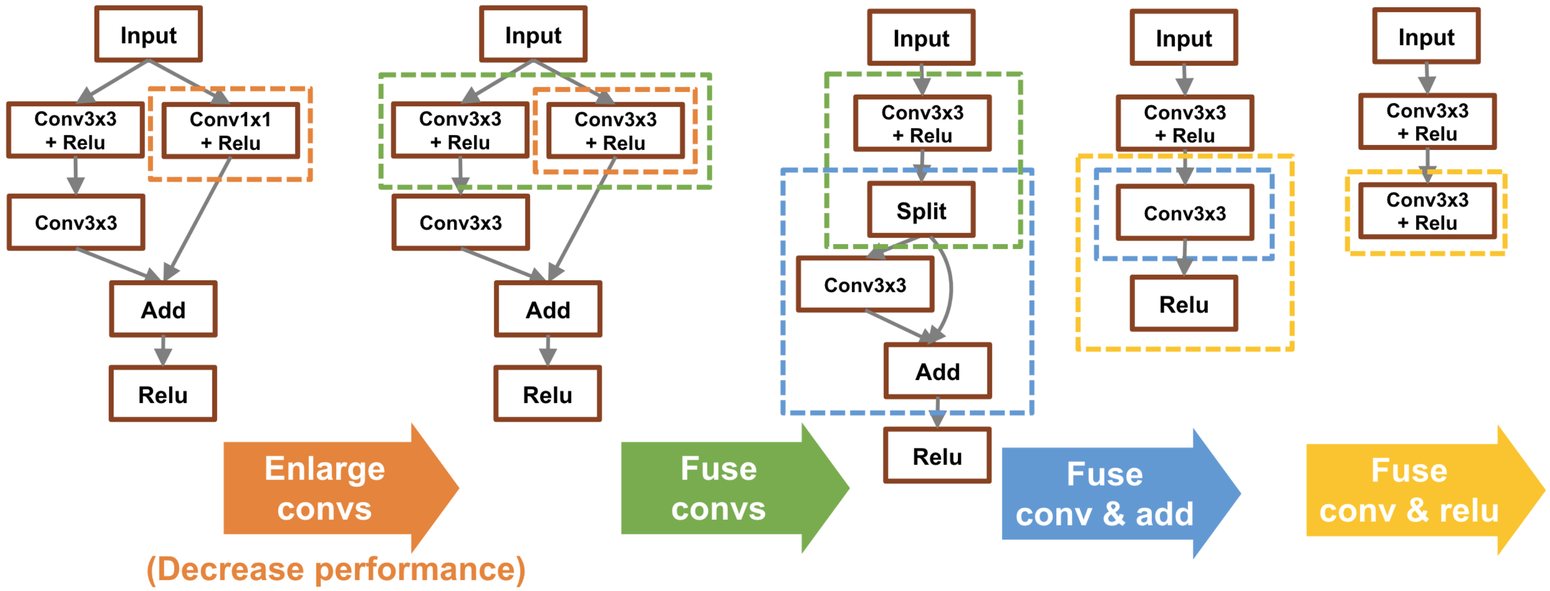
Runtime: schedule / memory

Operator optimization/compilation



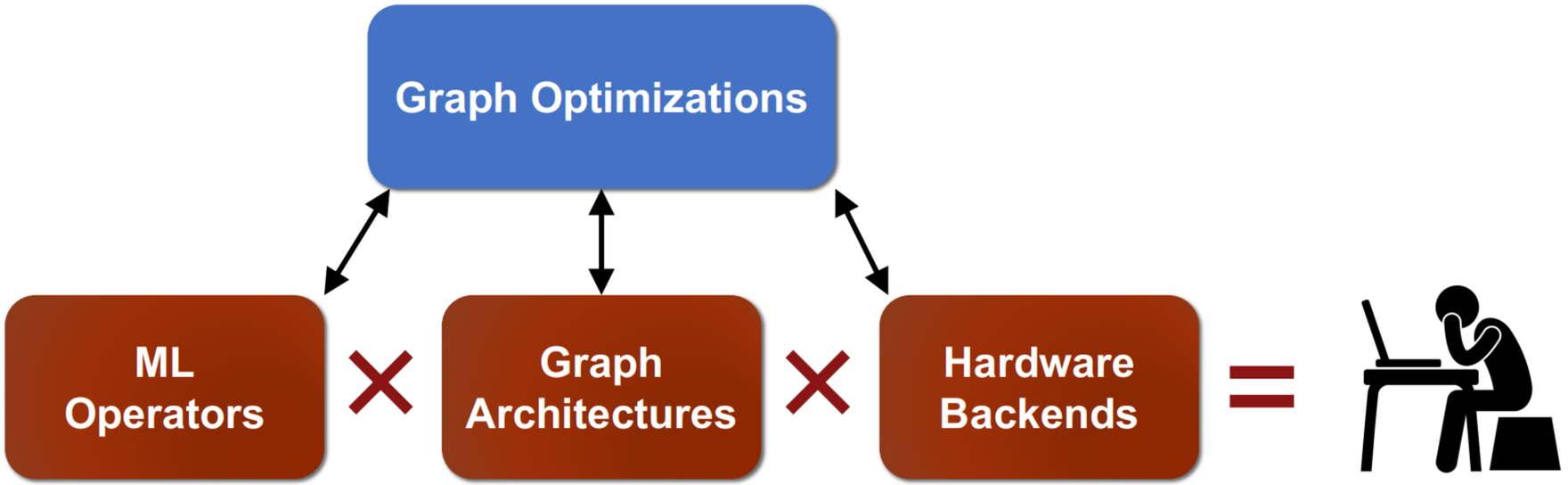
Recall Our Goal

- Goal:
  - Rewrite the original Graph G to G'; G' runs faster than G
  - Straightforward solution: expert templates



- The final graph is 30% faster on V100 but 10% slower on K80.

Problems of High-level Graph Optimizations



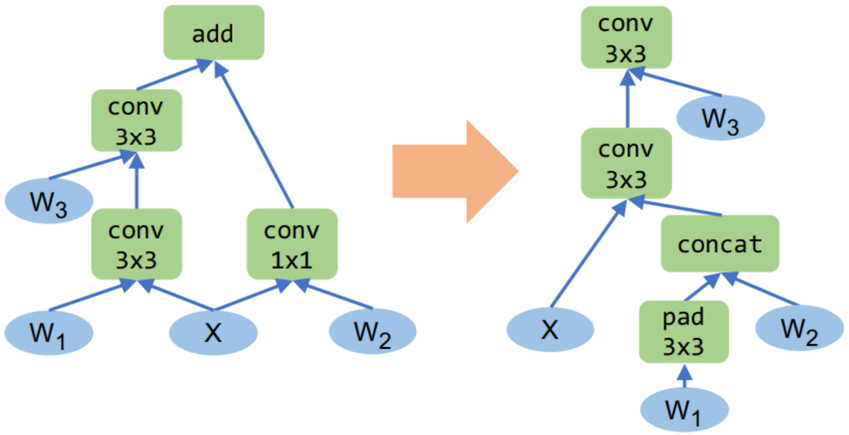
Problem: Infeasible to manually design graph optimizations for all cases

Summary of Limitations

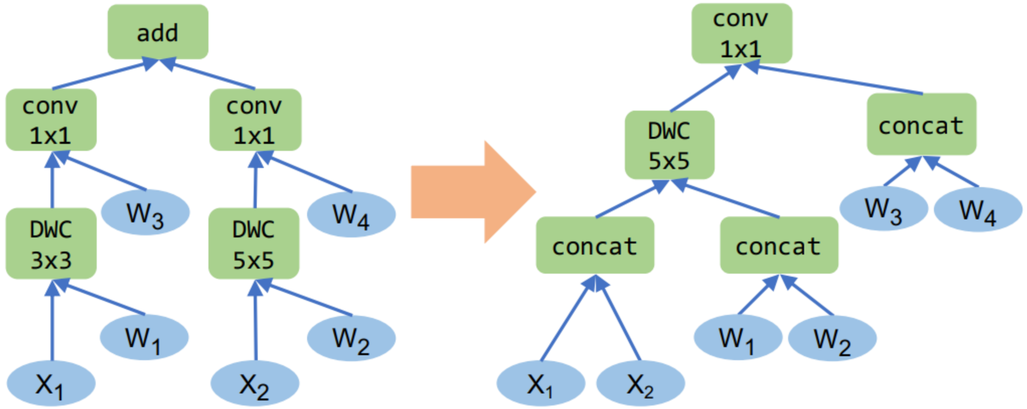
Robustness
Experts' heuristics do not apply to all DNNs/hardware

Scalability
New operators and graph structures require more rules

Performance
Miss subtle optimizations for specific DNNs/hardware



Only apply to **specific hardware**



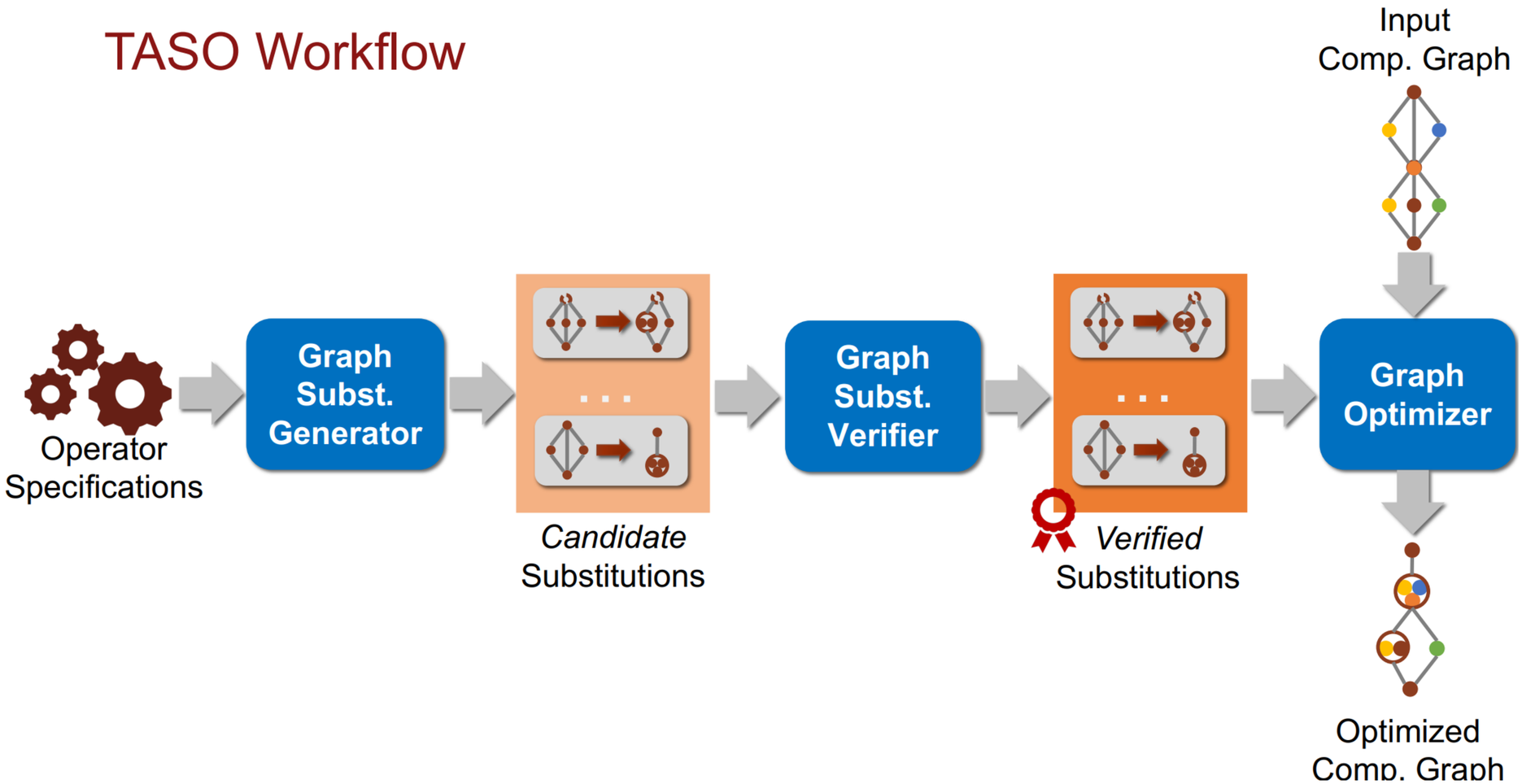
Only apply to **specialized graph structures**

Automate Graph Transformation: Big Picture

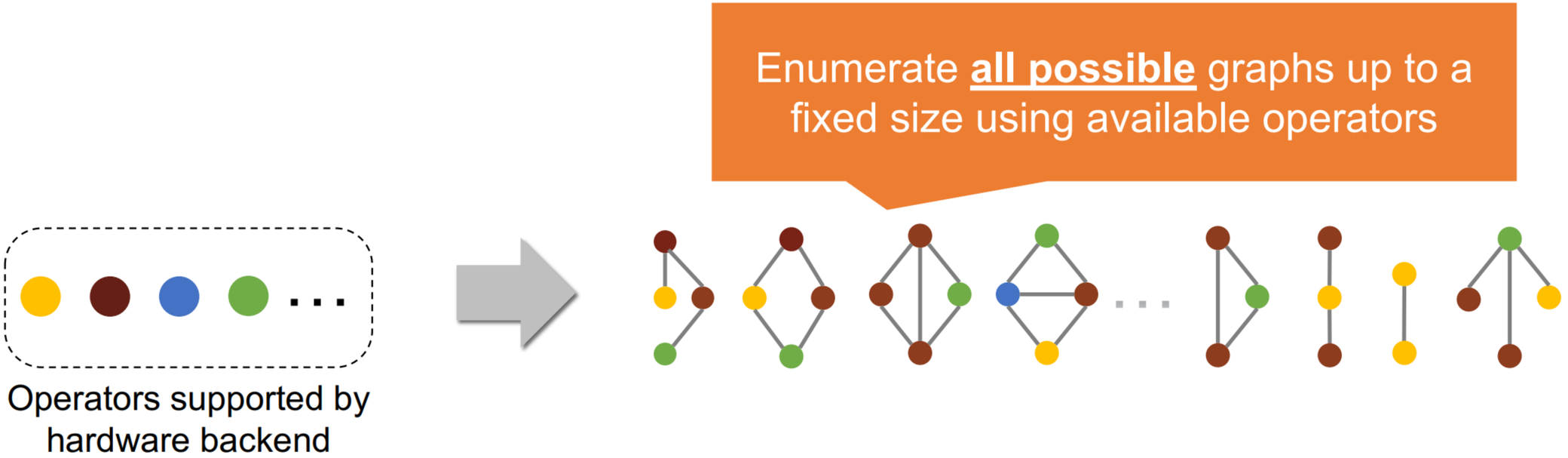
Key idea: replace manually-designed graph optimizations with automated generation and verification of graph substitutions for tensor algebra

- Less engineering effort: 53,000 LOC for manual graph optimizations in TensorFlow → 1,400 LOC
- Better performance: outperform existing optimizers by up to 3x
- Correctness: formally verified

Enumerate and Verify ALL possible graph

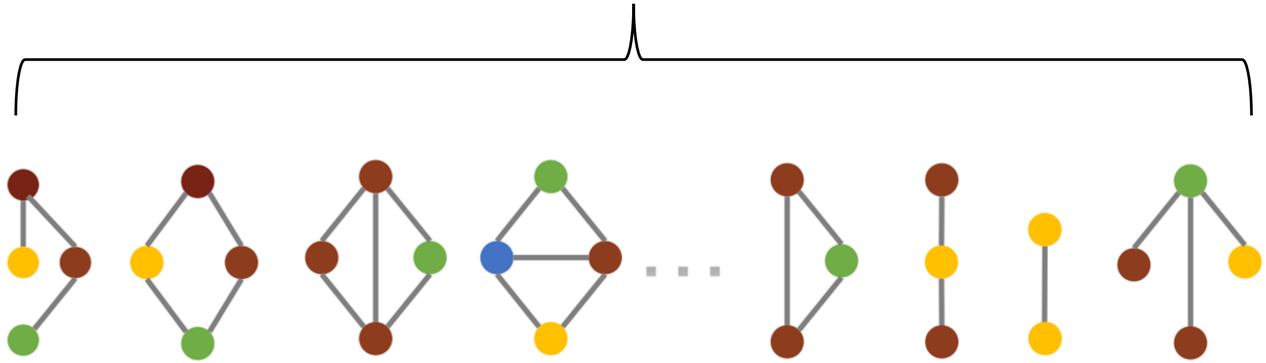


Graph Substitution Generator



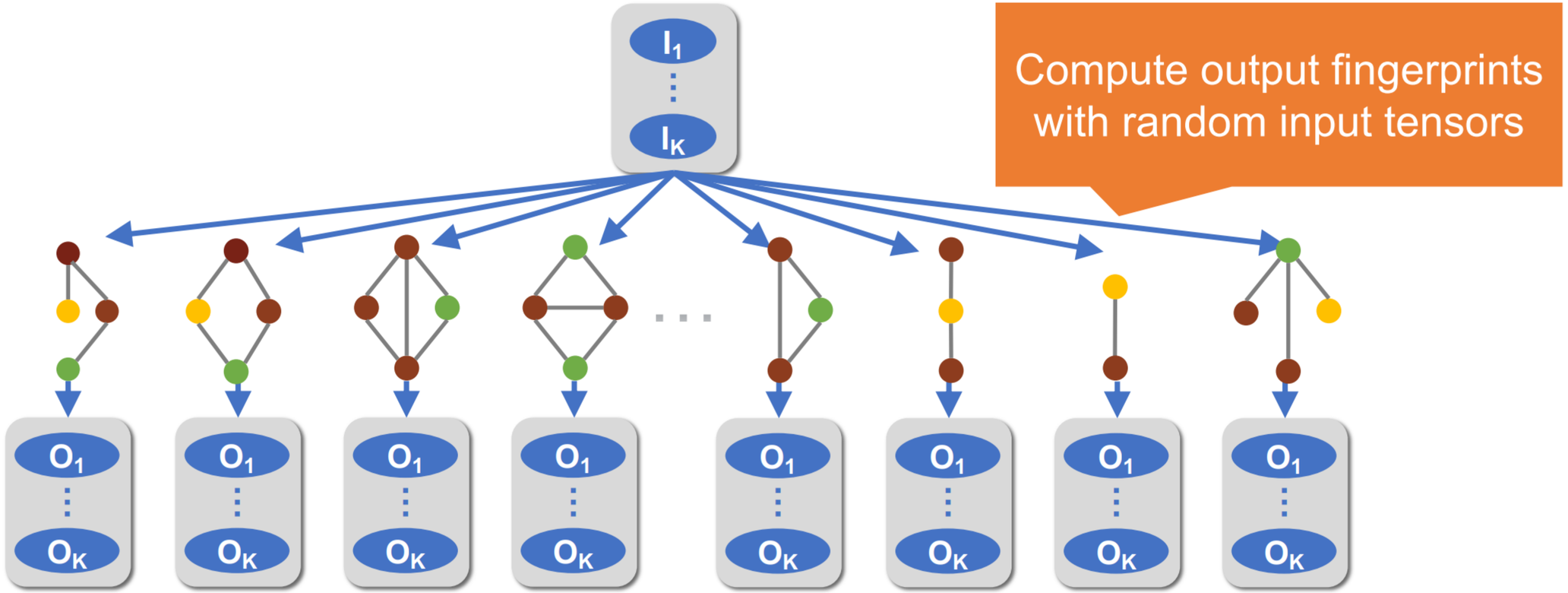
There are many subgraphs even only given 4 Ops

66M graphs with up to 4 operators



A substitution = a pair of equivalent graphs

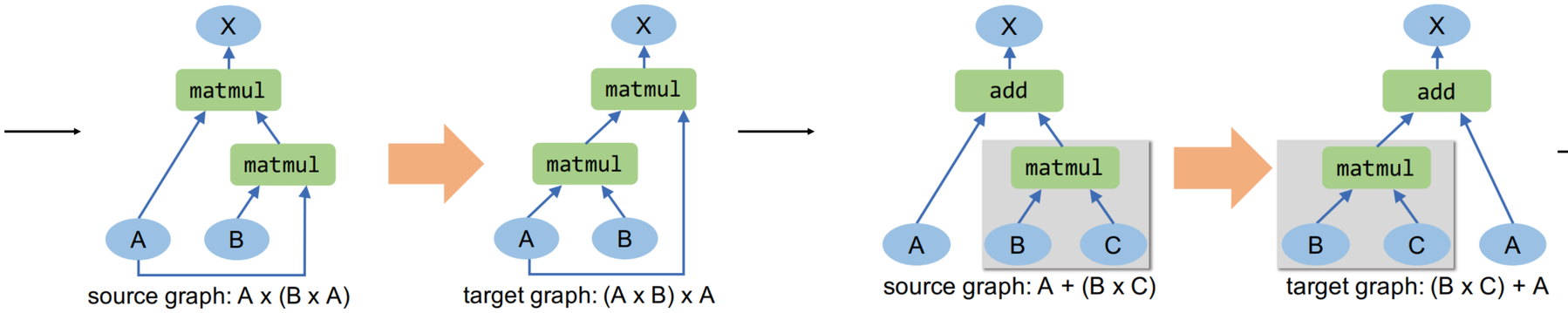
Graph Substitution Generator



We can generate 28744 substitutions by enumerating graphs with up to 4 ops

Pruning repeated graphs

28744 substitutions



734 substitutio

Variable renaming

Common subgraph

Can we trust graph substitutions?

- We have f(a) = g(b), f(b) = g(b)
  - But can we say: f(x) = g(x) for ∀x
- We need to verify formally.