15-779 Lecture 8: ML Compilers Part 1: Tile-based DSL

Zhihao Jia

Computer Science Department

Carnegie Mellon University

Triton

- “A lower-level PyTorch”
  - Tensors in GPU shared memory
  - Inputs can be PyTorch tensors or custom data-structures (e.g., tensors of pointers)

- Tensors can have pointer data-type:
  - All standard python control flow structure (for/if/while/return) are supported
  - Python code is lowered to Triton IR

* Slides from Philippe Tillet

Triton Example: Vector Addition (Z = X + Y)

- `triton.jit`: just-in-time compilation
- program_id is similar to blockIdx
- `tl.arange` returns all integer values within the interval
- `offsets` is a list of offsets
- **`mask`** is a list of bool to guard memory operations
- `tl.load` loads data from global to shared memory
- `tl.store` saves data back to global memory

```
N = 16 * 1024

@triton.jit
def add_kernel(x_ptr, y_ptr, z_ptr, BLOCK_SIZE):
    pid = tl.program_id(0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < N
    # Load x and y from global to shared memory
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    z = x + y
    # Write x + y back to global mem.
    tl.store(z_ptr + offsets, z, mask=mask)

X = torch.randn(N, device="cuda")
Y = torch.randn(N, device="cuda")
Z = torch.randn(N, device="cuda")

# Each threadblock process 1024 elements
grid = (16,)
add_kernel[grid](x, y, z, BLOCK_SIZE=1024)
```

Comparing CUDA and Triton

- CUDA Version

```
BLOCK = 512
def add(X, Y, Z, N):
    # In CUDA, each kernel
    # instance itself uses an SIMT execution
    # model, where instructions are executed in
    # parallel for different values of threadIdx
    tid = threadIdx.x
    bid = blockIdx.x
    # scalar index
    idx = bid * BLOCK + tid
    if idx < N:
        Z[idx] = X[idx] + Y[idx]


grid = (ceil_div(N, BLOCK),)
block = (BLOCK,)
add[grid, block](x, y, z, x.shape[0])
```

- Triton Version

```
BLOCK = 512
def add_kernel(x_ptr, y_ptr, z_ptr, BLOCK_SIZE):
    pid = tl.program_id(0)
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    mask = offsets < N
    # Load x and y from global to shared memory
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    z = x + y
    # Write x + y back to global mem.
    tl.store(z_ptr + offsets, z, mask=mask)

grid = (ceil_div(N, BLOCK),)
add_kernel[grid](x, y, z, BLOCK_SIZE=1024)
```

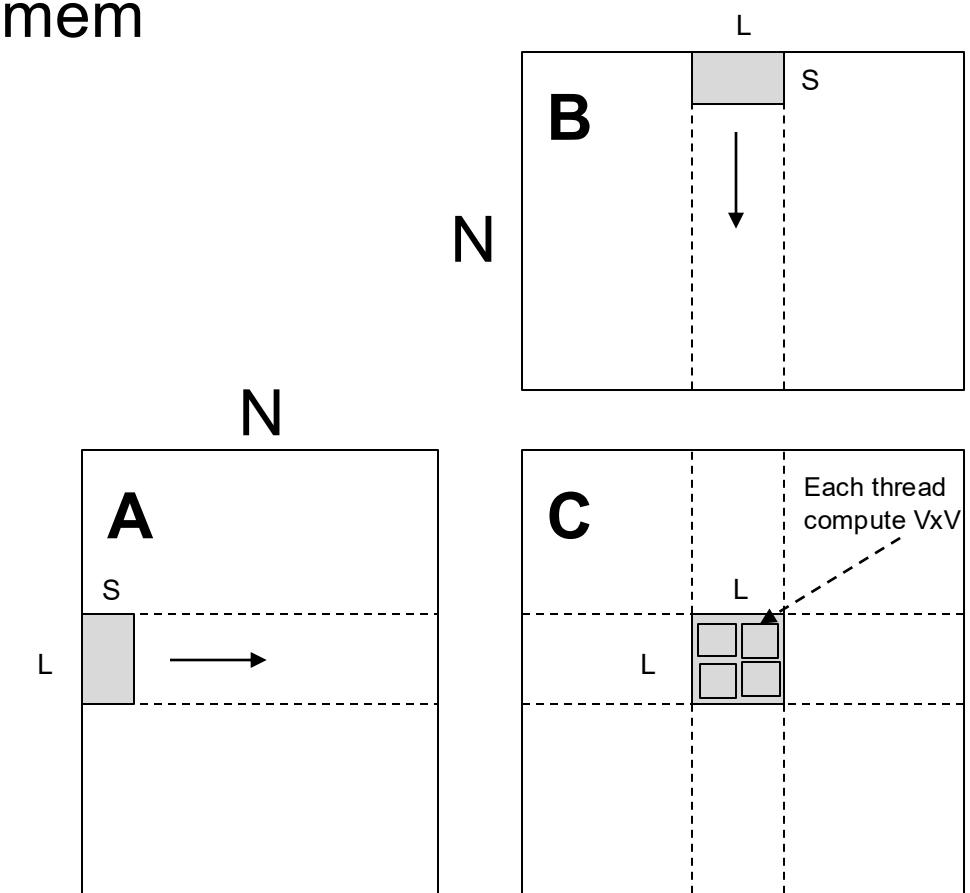
Example: Softmax

- $softmax(X_{ij}) = \frac{e^{X_{ij}}}{\sum_k e^{X_{ik}}} = \frac{e^{X_{ij}-m_i}}{\sum_k e^{X_{ik}-m_i}}$
- $m_i = max_k(X_{ik})$
- rows of softmax are independent, parallelize across those
- num_warps: how many warps to allocate per program
- num_warps=4 means each program has 32 * 4 = 128 threads

```
@triton.jit
def softmax_kernel(output_ptr, input_ptr,
                   input_row_stride, output_row_stride,
                   n_cols, BLOCK_SIZE):
    row_idx = tl.program_id(0)
    row_start_ptr = input_ptr + row_idx * input_row_stride
    col_offsets = tl.arange(0, BLOCK_SIZE)
    row = tl.load(input_ptrs, mask=col_offsets < n_cols, other=-inf)
    # subtract maximum
    row_minus_max = row - tl.max(row, axis=0)
    numerator = tl.exp(row_minus_max)
    denominator = tl.sum(numerator, axis=0)
    softmax_output = numerator / denominator
    output_row_start_ptr = output_ptr + row_idx * output_row_stride
    output_ptrs = output_row_start_ptr + col_offsets
    tl.store(output_ptrs, softmax_output, mask=col_offsets < n_cols)

def softmax(x):
    n_rows, n_cols = x.shape
    BLOCK_SIZE = triton.next_power_of_2(n_cols)
    num_warps = 4 if BLOCK_SIZE < 2048 else 8
    softmax_kernel[(n_rows,)](y, x,
                              x.stride(0), y.stride(0),
                              n_cols, BLOCK_SIZE=BLOCK_SIZE,
                              num_warps=num_warps)
```

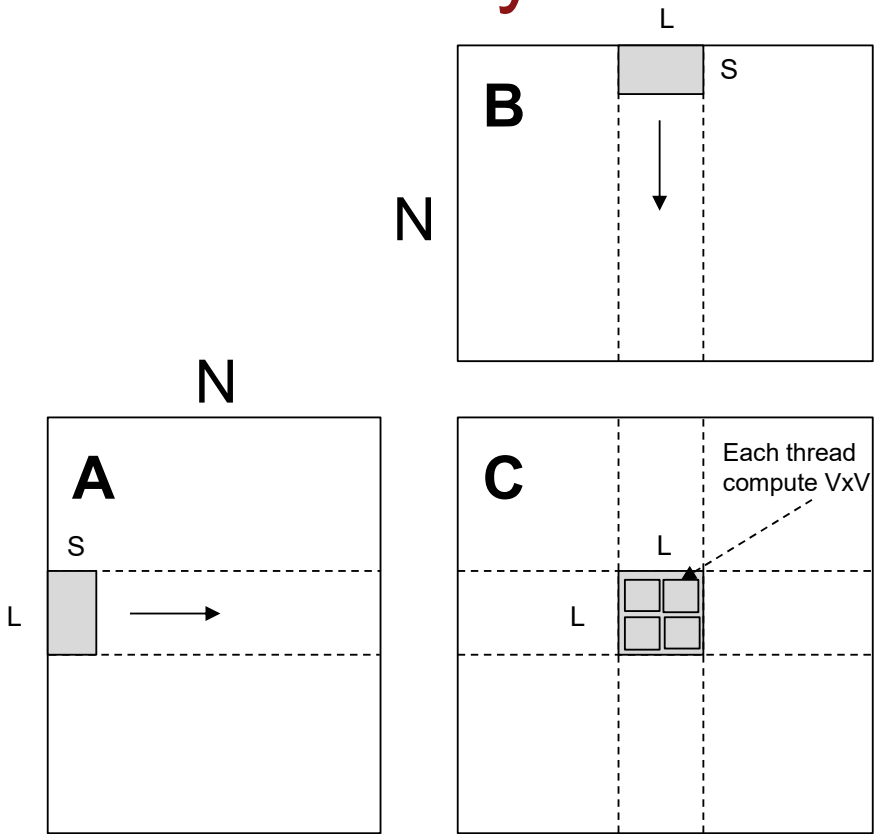
Recall: Block-Level Shared Memory Tiling for MatMul

- A block computes a L x L submatrix
- A thread computes a V x V submatrix and reuses the matrices in shared mem




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

Recall: Analysis of Memory Reuse



Global memory access per thread block: $2LN$
Number of thread blocks: 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[:];
}
```

Example: Matrix Multiplication in Triton

- Blocked algorithm to multiply matrices

```
# Do in parallel
for m in range(0, M, BLOCK_SIZE_M):
  # Do in parallel
  for n in range(0, N, BLOCK_SIZE_N):
    acc = zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=float16)
    for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
      acc += dot(a, b)
    C[m : m+BLOCK_SIZE_M, n : n+BLOCK_SIZE_N] = acc
```

Matrix Multiplication: Pointer Arithmetic in Triton

```
for k in range(0, K, BLOCK_SIZE_K):
      a = A[m : m+BLOCK_SIZE_M, k : k+BLOCK_SIZE_K]
      b = B[k : k+BLOCK_SIZE_K, n : n+BLOCK_SIZE_N]
```

- For a row-major 2D tensor **X**, the memory location of `X[i,j]` is given by `&X[i,j] = X + i * stride_x + j * stride_y`
- For a given program, how to compute all pointers for A and B?

```
# pid_m and pid_n are program ids along the m and n dimensions
offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
offs_k = tl.arange(0, BLOCK_SIZE_K)
a_ptrs = a_ptr + (offs_am[:, None]*stride_am + offs_k [None, :]*stride_ak)
b_ptrs = b_ptr + (offs_k [:, None]*stride_bk + offs_bn[None, :]*stride_bn)
```

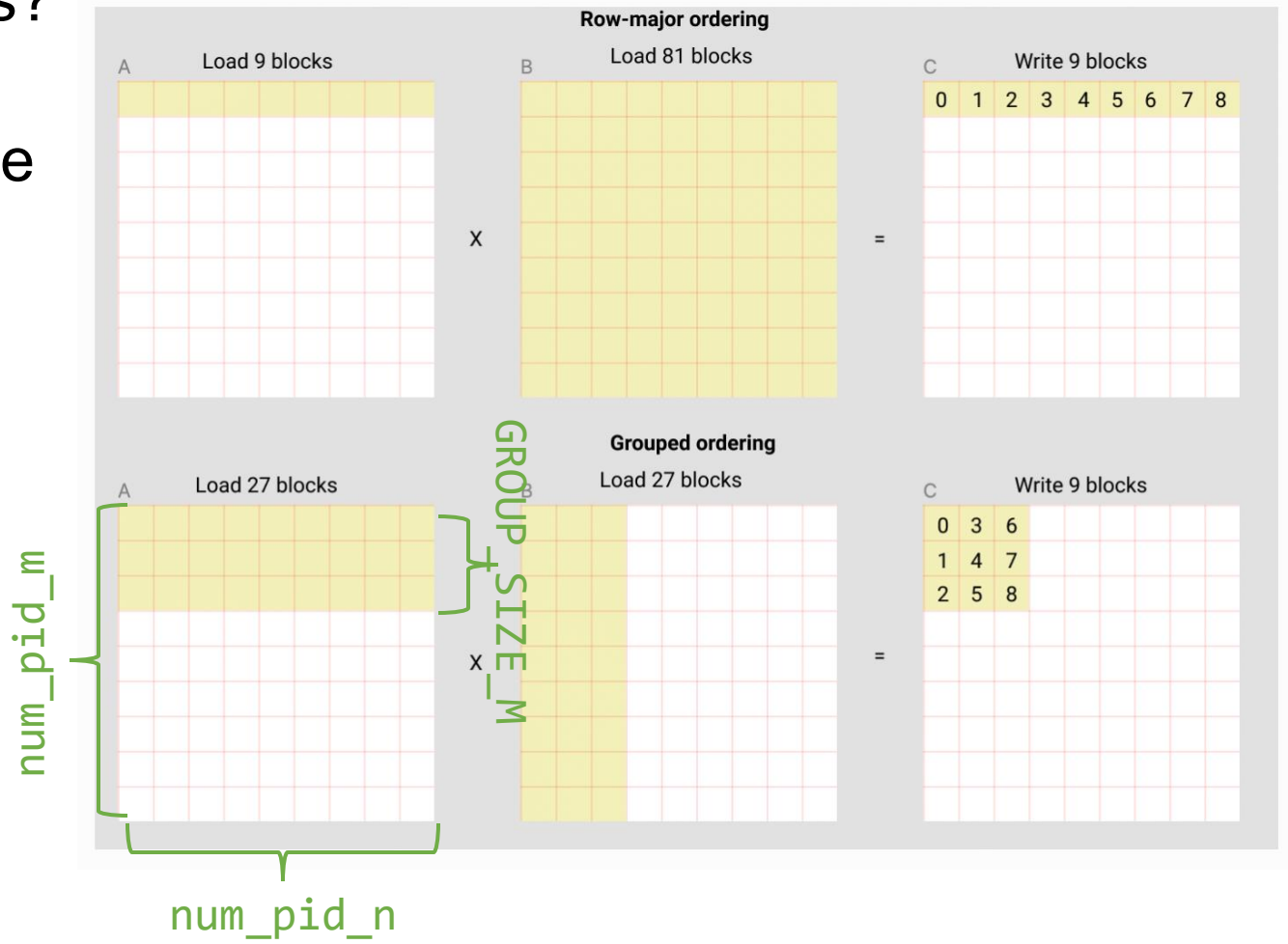
Matrix Multiplication: L2 Cache Reuse

Matrix Multiplication: L2 Cache Reuse

In which order to execute these tiles?

- Grouped ordering to reduce memory loads by reusing L2 cache

```
pid = tl.program_id(axis=0)
num_pid_m = tl.cdiv(M, BLOCK_SIZE_M) # = 9
num_pid_n = tl.cdiv(N, BLOCK_SIZE_N) # = 9
num_pid_in_group = GROUP_SIZE_M * num_pid_n # = 27
group_id = pid // num_pid_in_group
first_pid_m = group_id * GROUP_SIZE_M
# *Within a groups*,
# programs are ordered in a column-major order
# Row-id of the program in the *launch grid*
pid_m = first_pid_m + pid % GROUP_SIZE_M
# Col-id of the program in the *launch grid*
pid_n = (pid % num_pid_in_group) // GROUP_SIZE_M
```



MatMul in Triton

Compute which column/row the current program to process

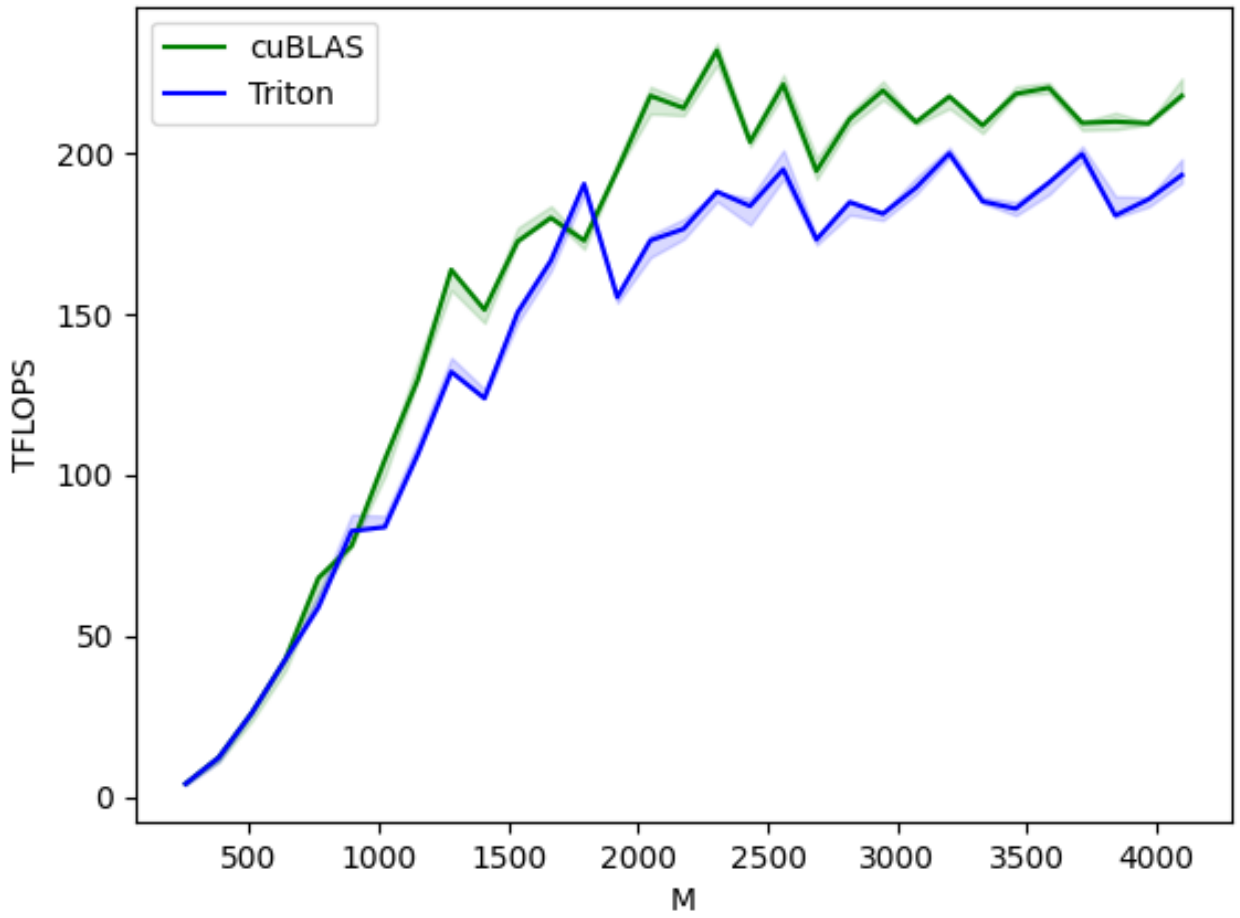
Compute pointers for loading A and B
- a_ptrs includes [M, K] pointers
- b_ptrs includes [K, N] pointers

Main loop over K dimension
- Load a, b into shared memory
- Accumulate results in shared memory

Write results back to global memory
- c_ptrs includes [M, N] pointers

```
@triton.jit
def matmul_kernel(…):
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    pid_m = first_pid_m + ((pid % num_pid_in_group) % GROUP_SIZE_M)
    pid_n = (pid % num_pid_in_group) // GROUP_SIZE_M
    offs_am = (pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M))
    offs_bn = (pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N))
    offs_k = tl.arange(0, BLOCK_SIZE_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_SIZE_K, other=0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_SIZE_K, other=0.0)
        accumulator = tl.dot(a, b, accumulator)
        a_ptrs += BLOCK_SIZE_K * stride_ak
        b_ptrs += BLOCK_SIZE_K * stride_bk
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
    offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

Matching 90% of cuBLAS Performance



```
a_desc = TensorDescriptor(a, a.shape, a.stride(), (BLOCK_SIZE_M, BLOCK_SIZE_K))
b_desc = TensorDescriptor(b, b.shape, b.stride(), (BLOCK_SIZE_K, BLOCK_SIZE_N))
```

Warp Specialization (Hopper GPUs)

Same as before:
Compute which column/row the current program to process

offs_am and offs_bn are scalars (instead of arrays of pointers)

Main loop over K dimension

- Load a, b into shared memory using TMA
- Accumulate results in shared memory

```
def matmul_kernel_tma(a_desc, b_desc, c_desc, M, N, K
                      BLOCK_SIZE_M: tl.constexpr,
                      BLOCK_SIZE_N: tl.constexpr,
                      BLOCK_SIZE_K: tl.constexpr,
                      GROUP_SIZE_M: tl.constexpr):
    pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_SIZE_M
    pid_m = first_pid_m + (pid % GROUP_SIZE_M)
    pid_n = (pid % num_pid_in_group) // GROUP_SIZE_M
    k_tiles = tl.cdiv(K, BLOCK_SIZE_K)
    offs_am = pid_m * BLOCK_SIZE_M
    offs_bn = pid_n * BLOCK_SIZE_N
    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
    for k in tl.range(k_tiles, warp_specialize=True):
        offs_k = k * BLOCK_SIZE_K
        a = a_desc.load([offs_am, offs_k])
        b = b_desc.load([offs_bn, offs_k])
        accumulator = tl.dot(a, b.T, accumulator)
    c = accumulator.to(tl.float16)
    offs_cm = pid_m * BLOCK_SIZE_M
    offs_cn = pid_n * BLOCK_SIZE_N
    c_desc.store([offs_cm, offs_cn], c)
```

Persistent Kernel in Triton

Launch NUM_OF_SMS programs;
Each program processes multiple tiles

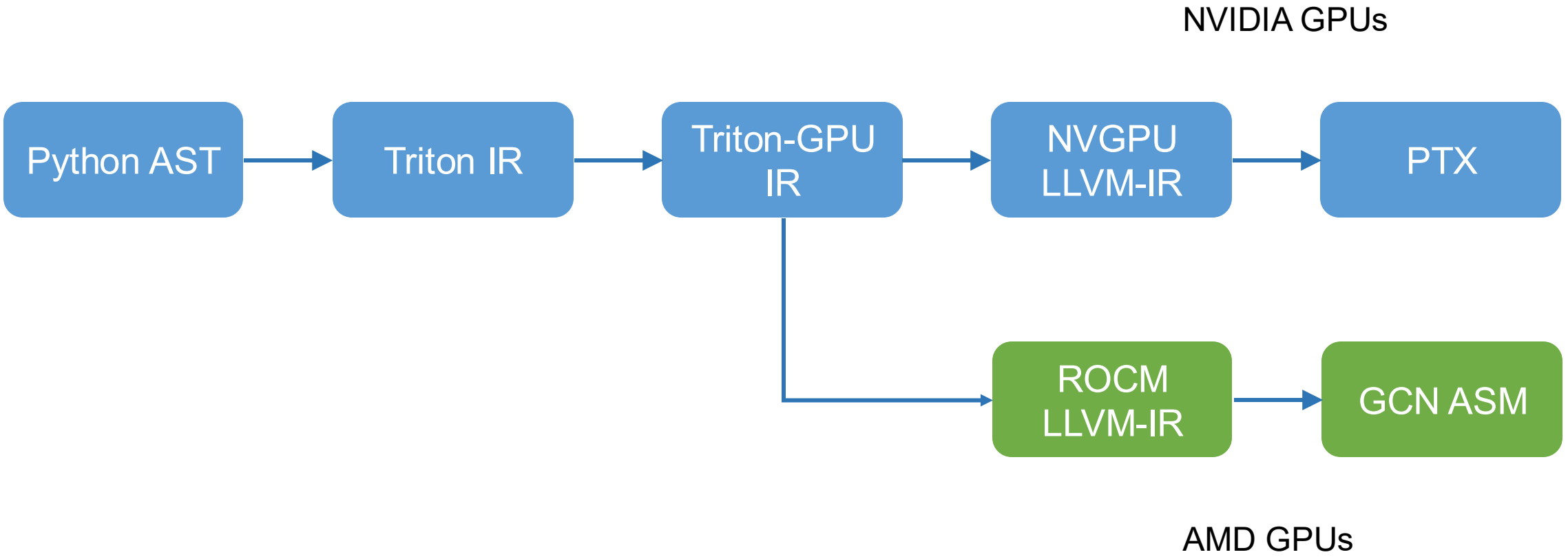
```
def matmul_kernel_persistent(…):
    start_pid = tl.program_id(axis=0)
    num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
    num_pid_n = tl.cdiv(N, BLOCK_SIZE_N)
    k_tiles = tl.cdiv(K, BLOCK_SIZE_K)
    num_tiles = num_pid_m * num_pid_n
    offs_k_for_mask = tl.arange(0, BLOCK_SIZE_K)
    num_pid_in_group = GROUP_SIZE_M * num_pid_n

    for tile_id in tl.range(start_pid, num_tiles, NUM_OF_SMS, flatten=True):
        pid_m, pid_n = compute_pid(tile_id, num_pid_in_group, num_pid_m, GROUP_SIZE_M, NUM_SMS)
        start_m = pid_m * BLOCK_SIZE_M
        start_n = pid_n * BLOCK_SIZE_N
        offs_am = start_m + tl.arange(0, BLOCK_SIZE_M)
        offs_bn = start_n + tl.arange(0, BLOCK_SIZE_N)

        accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
        for ki in range(k_tiles):
            offs_k = ki * BLOCK_SIZE_K + tl.arange(0, BLOCK_SIZE_K)
            a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] * stride_ak)
            b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] * stride_bn)
            a = tl.load(a_ptrs, mask=offs_k_for_mask[None, :] < K - ki * BLOCK_SIZE_K, other=0.0)
            b = tl.load(b_ptrs, mask=offs_k_for_mask[:, None] < K - ki * BLOCK_SIZE_K, other=0.0)
            accumulator = tl.dot(a, b, accumulator)

        offs_cm = pid_m * BLOCK_SIZE_M + tl.arange(0, BLOCK_SIZE_M)
        offs_cn = pid_n * BLOCK_SIZE_N + tl.arange(0, BLOCK_SIZE_N)
        c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None, :]
        c = accumulator.to(tl.float16)
        tl.store(c_ptrs, c, mask=(offs_cm[:, None] < M) & (offs_cn[None, :] < N))
```

Compute which tile to process

Compute pointers for loading A and B
- a_ptrs includes [M, K] pointers
- b_ptrs includes [K, N] pointers

Write results back to global memory
- c_ptrs includes [M, N] pointers

Triton Compilation Stack

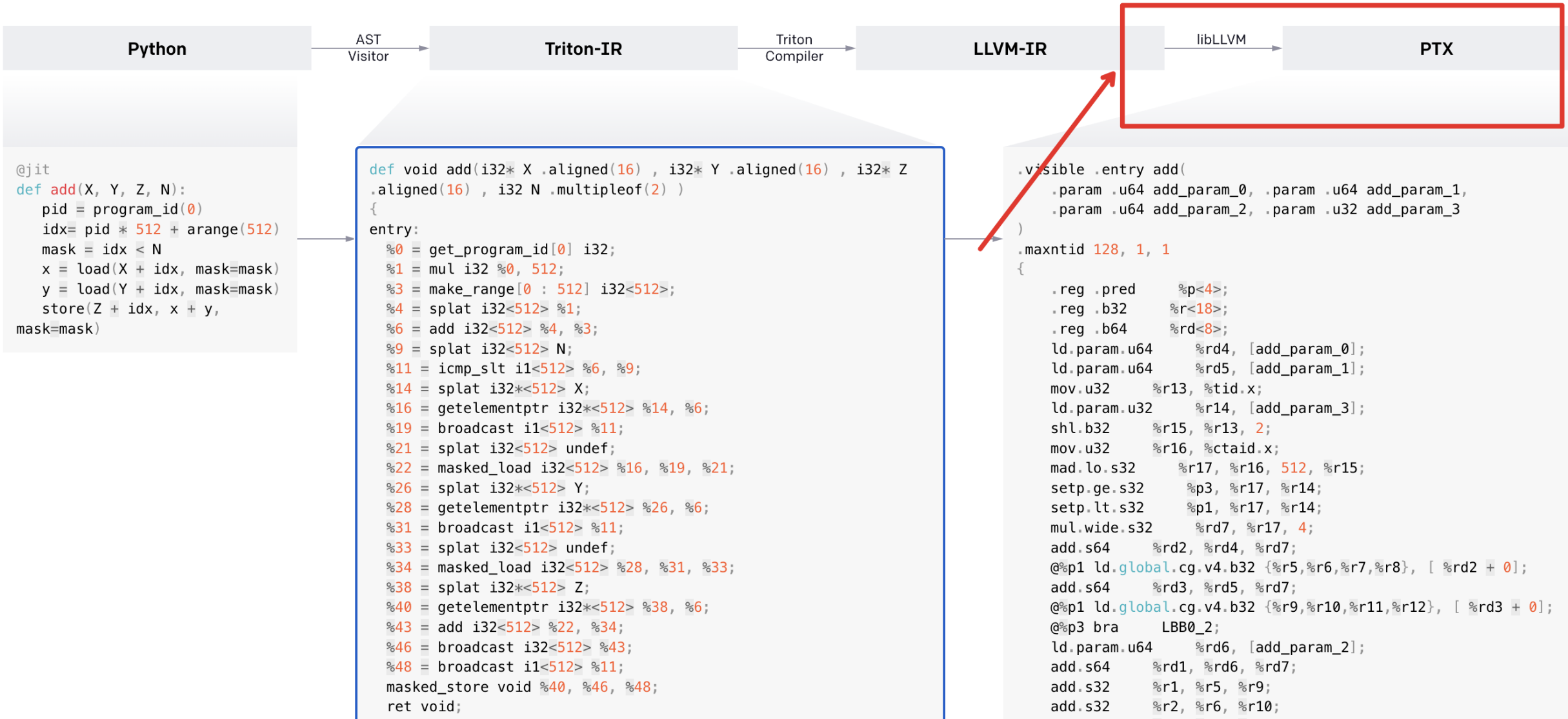


Step 1: Just-in-Time Symbol Resolution

- Type inference: implicit conversion from `troch.Tensor` to `ptr<dtype>`
- Kernel argument specialization
- Signatures are hashed: kernels that were compiled before are retrieved and loaded to GPU memory

```
@triton.jit
def kernel(X, stride_xm, stride_xn,
           Z, stride_zm, stride_zn,
           BLOCK_M, BLOCK_N):
    off_m = tl.arange(0, BLOCK_M)
    off_n = tl.arange(0, BLOCK_N)
    Xs = X+off_m[:,None]*stride_xm+off_n[None,:]*stride_xn
    Zs = Z+off_m[:,None]*stride_zm+off_n[None,:]*stride_zn
    tl.store(Zs, tl.load(Xs))

shape = [128, 128]
x = torch.randn(shape, device=`cuda`)
z = torch.randn(shape, device=`cuda`)
kernel[(1, 1)](x, x.stride(0), x.stride(1),
               z, z.stride(0), z.stride(1),
               BLOCK_M = shape[0],
               BLOCK_N = shape[1])
```

Step 2: Triton-IR Generation



Step 3: PTX CodeGen

Solve many optimization problems:

- Shared memory allocation
- Shared memory barrier placement
- Instruction selection

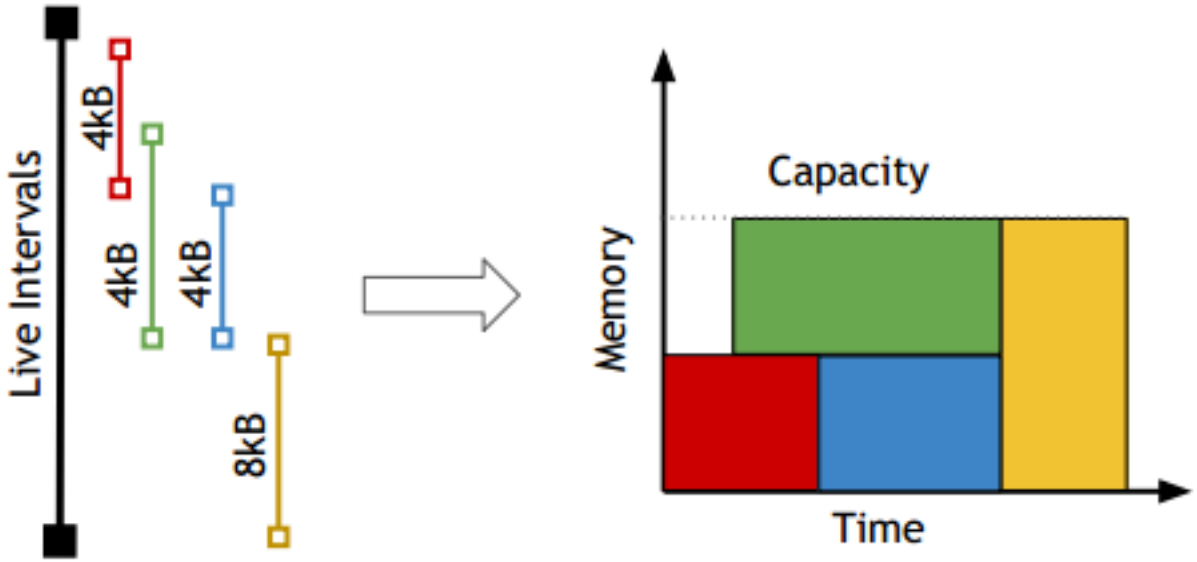


Figure 7. Shared Memory Allocation

https://ranko-mosic.medium.com/open-source-llm-software-stack-f052a402ff51

Step 4: Execution

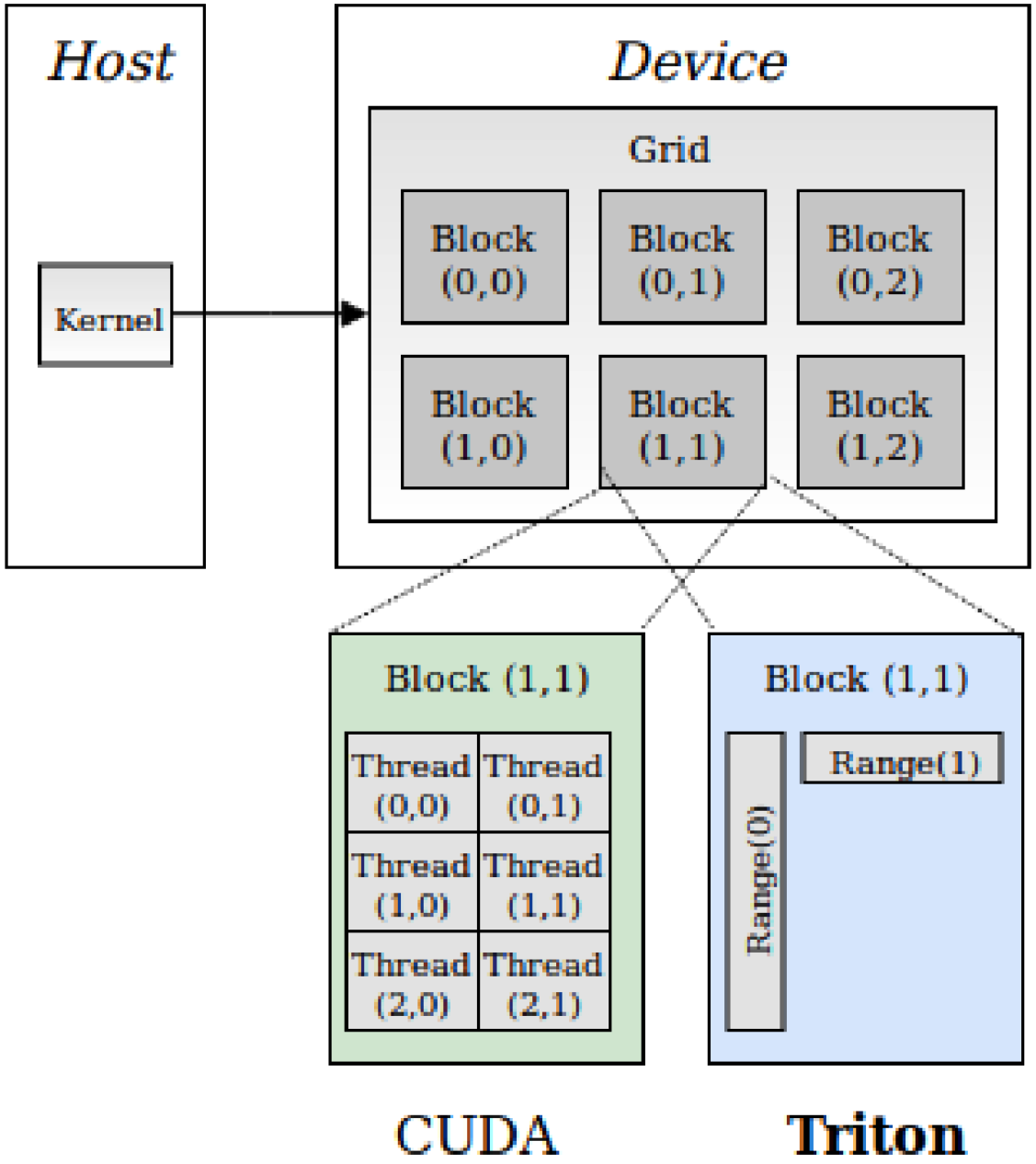
- Use PTX to generate binary from Triton-IR
- Triton kernels are compiled just-in-time and launched dynamically

```
@triton.jit
def kernel(X, stride_xm, stride_xn,
           Z, stride_zm, stride_zn,
           BLOCK_M, BLOCK_N):
    off_m = tl.arange(0, BLOCK_M)
    off_n = tl.arange(0, BLOCK_N)
    Xs = X+off_m[:,None]*stride_xm+off_n[None,:]*stride_xn
    Zs = Z+off_m[:,None]*stride_zm+off_n[None,:]*stride_zn
    tl.store(Zs, tl.load(Xs))

shape = [128, 128]
x = torch.randn(shape, device=`cuda`)
z = torch.randn(shape, device=`cuda`)
kernel[(1, 1)](x, x.stride(0), x.stride(1),
               z, z.stride(0), z.stride(1),
               BLOCK_M = shape[0],
               BLOCK_N = shape[1])
```

Discussion: CUDA and Triton

- What are the similarities?

- What are the key differences?



Comparing CUDA and Triton

	CUDA	Triton
Thread Block	Yes	Yes
Threads	Yes	No
Memory Coalescing	Manual	Automatic
Shared Memory Management	Manual	Automatic
Scheduling (Within SMs)	Manual	Automatic
Scheduling (Across SMs)	Manual	Manual