CS6216 Advanced Topics in Machine Learning (Systems)

Hardware Acceleration

Yao LU

4 Sep 2024

National University of Singapore
School of Computing

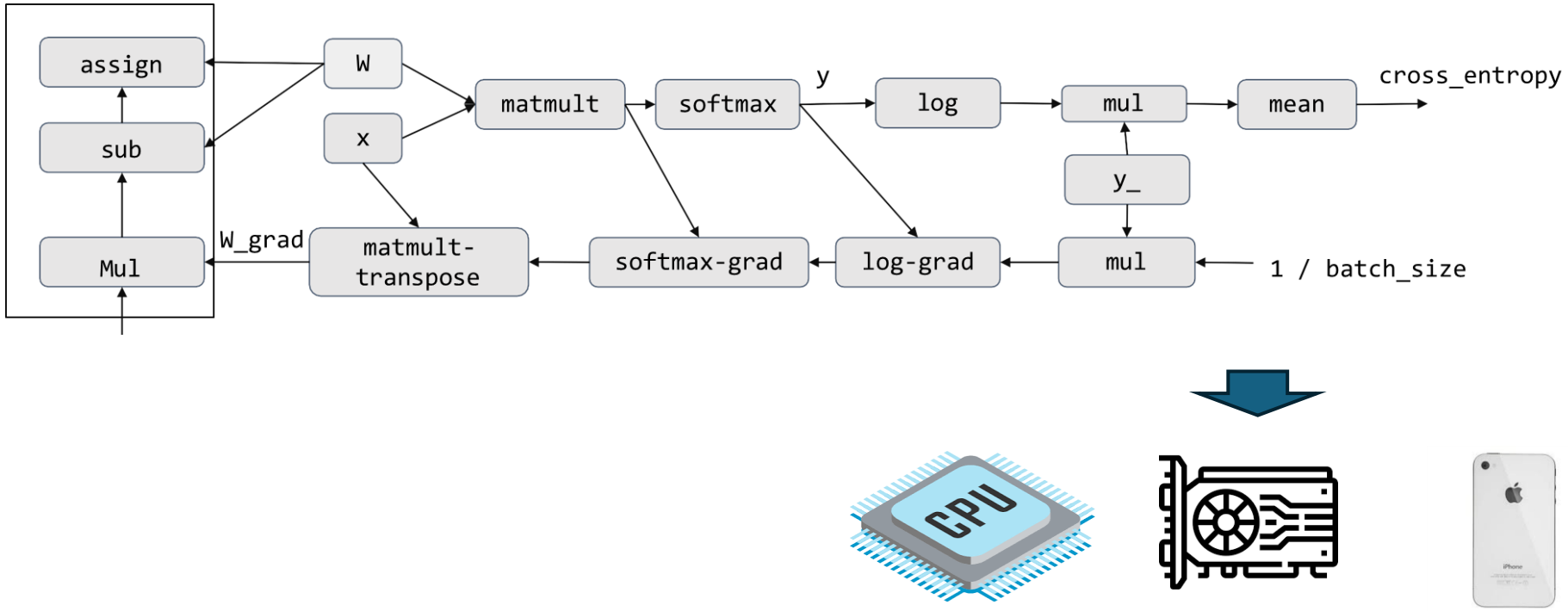
Logistics

- Homework 2 will be released
  - Topic: back propagation and AutoDiff
- Project proposal
  - Indicate your group members
  - 1-page writeup to describe your project and expected outcome
- Both due Sep 21
  - Decide HW vs. project

Outline

- General acceleration techniques
  - Case study: matrix multiplication on CPUs
- Overview of CUDA programming & Nvidia GPU architecture
  - Case study: 1D convolution on CUDA
  - Case study: matrix multiplication on CUDA

Recap: mapping compute graph to actual runtime



What can you do to make your program run faster on GPU / x86 CPU / any backend ?

Vectorization

Adding two arrays of length 256 bits
(float = 4B/32 bits)

```
void vec_add(void* A, void* B, void* C) {
      for (int i = 0; i < 64; ++i) {
            float4 a = load_float4(A + i*4);
            float4 b = load_float4(B + i*4);
            float4 c = add_float4(a, b);
            store_float4(C + i* 4, c);
       }
}
```

Additional requirements: memory (A, B, C) needs to be aligned to 128 bits

Parallelization

```
void vec_add(void* A, void* B, void* C) {
      #pragma omp parallel for
      for (int i = 0; i < 64; ++i) {
            float4 a = load_float4(A + i*4);
            float4 b = load_float4(B + i*4);
            float4 c = add_float4(a, b);
            store_float4(C * 4, c);
       }
}
```

Executes the computation on multiple threads

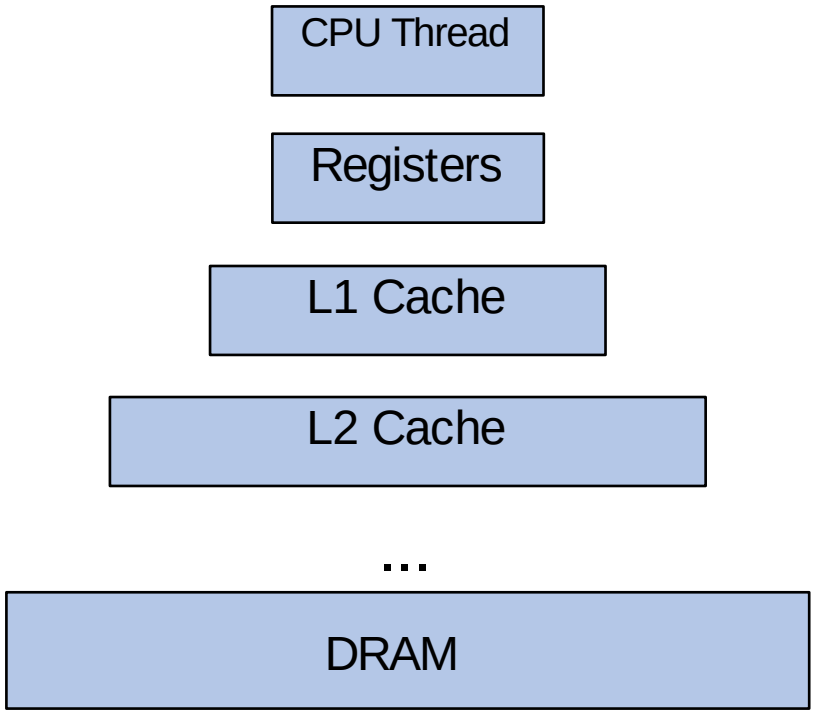
Vanilla matrix multiplication

Compute C = dot(A, B.T)

```
float A[n][n], B[n][n], C[n][n];

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[j][k];
    }
  }
}
```

O(n^3)

Memory hierarchy on modern CPUs



Latency

0.5 ns

7ns 14x L1 cache

200ns 20x L2 cache, 200x L1 cache

Architecture aware analysis

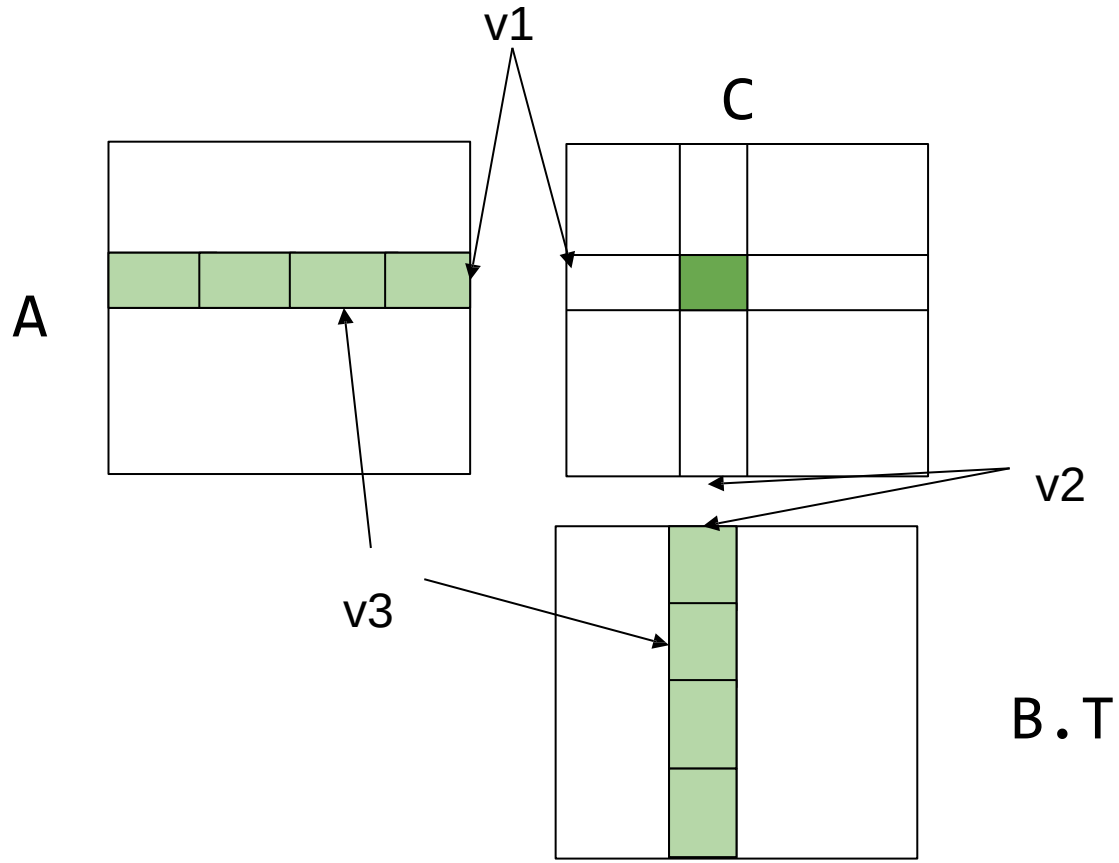
```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      register float c = 0;
      for (int k = 0; k < n; ++k) {
        register float a = A[i][k];
        register float b = B[j][k];
        c += a * b;
      }
      C[i][j] = c;
   }
}
```

```
A's dram->register time cost:  n^3
B's dram->register time cost:  n^3
```

Load cost: 2 * dram_speed * n^3

Register tiled matrix multiplication

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
   for (int j = 0; j < n/v2; ++j) {
      register float c[v1][v2] = 0;
      for (int k = 0; k < n/v3; ++k) {
        register float a[v1][v3] = A[i][k];
        register float b[v2][v3] = B[j][k];
        c += dot(a, b.T);
      }
      C[i][j] = c;
   }
}
```



Register tiled matrix multiplication

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
   for (int j = 0; j < n/v2; ++j) {
      register float c[v1][v2] = 0;
      for (int k = 0; k < n/v3; ++k) {
        register float a[v1][v3] = A[i][k];
        register float b[v2][v3] = B[j][k];
        c += dot(a, b.T);
      }
      C[i][j] = c;
   }
}
```

A's dram->register time cost: n^3/v2

B's dram->register time cost: n^3/v1

a get reused v2 times

b get reused v1 times

load cost: dram_speed * (n^3/v2 + n^3/v1)

Common reuse patterns

```
float A[n][n];
float B[n][n];
float C[n][n];

C[i][j] = sum(A[i][k] * B[j][k], axis=k)
```

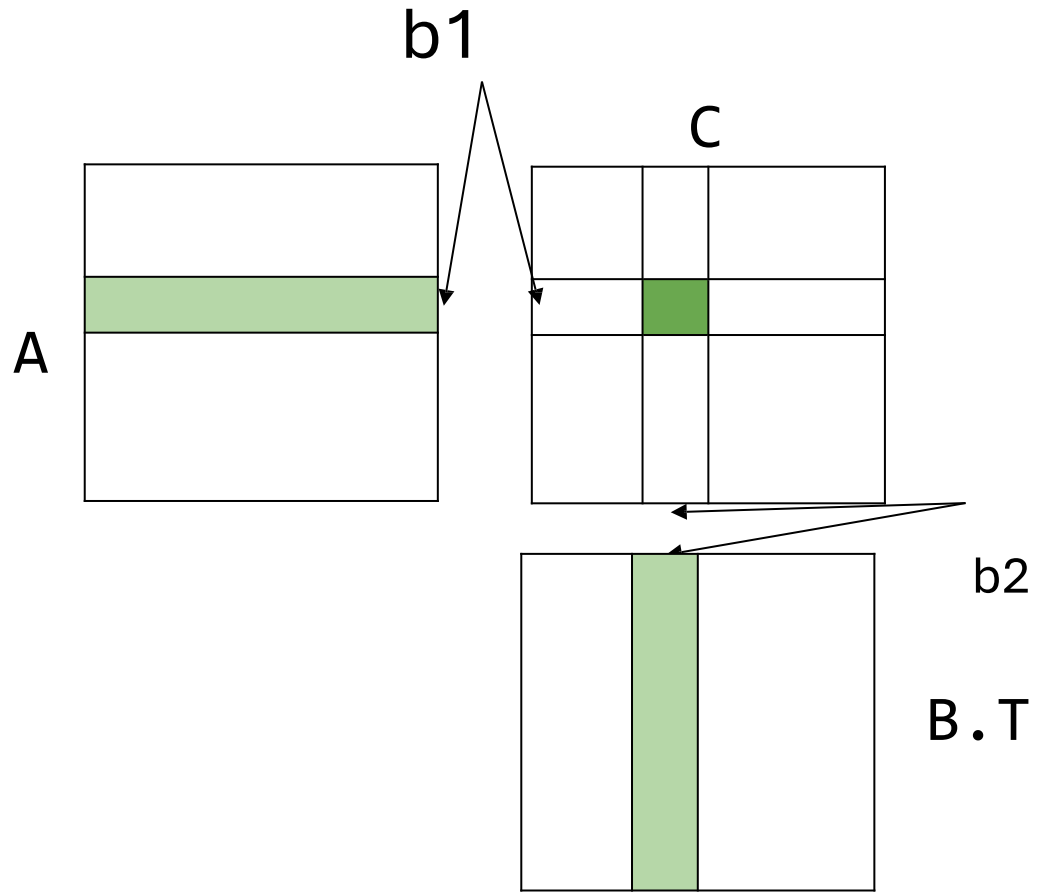
Access of A is independent of j,
tile the j dimension by v enables reuse of A for v times.

Possible reuse pattern in (2D) convolution

```
float Input[n][ci][h][w];
float Weight[co][ci][K][K];
float Output[n][co][h][w];

Output[b][co][y][x] =
    sum(Input[b][k][y+ry][x+rx] *
        Weight[co][k][ry][rx], axis=[k, ry, rx])
```

Cache line aware tiling

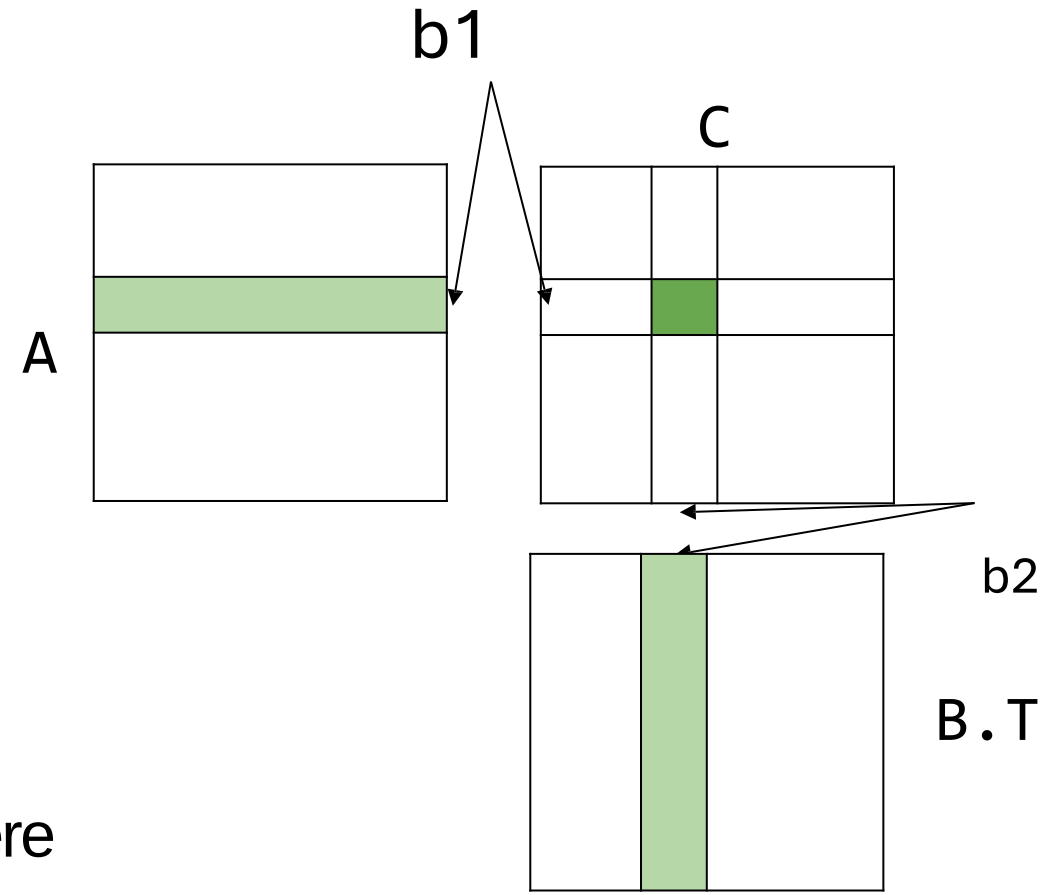
```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```



Cache line aware tiling

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

Sub-procedure, can apply register tiling here



Cache line aware tiling

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

A's dram->l1 time cost: n^2

B's dram->l1 time cost: n^3 / b1

Constraints:

- b1 * n + b2 * n < l1 cache size
- To still apply register blocking on dot
  - b1 % v1 == 0
  - b2 % v2 == 0

Putting together everything

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1/v1][n][v1] = A[i];
   for (int j = 0; j < n/b2; ++j) {
     l1cache b[b2/v2][n][v2] = B[j];
     for (int x = 0; x < b1/v1; ++x)
       for (int y = 0; y < b2/v2; ++y) {
         register float c[v1][v2] = 0;
         for (int k = 0; k < n; ++k) {
            register float ar[v1] = a[x][k][:];
            register float br[v2] = b[y][k][:];
                    C += dot(ar, br.T)
         }
       }
   }
}
```

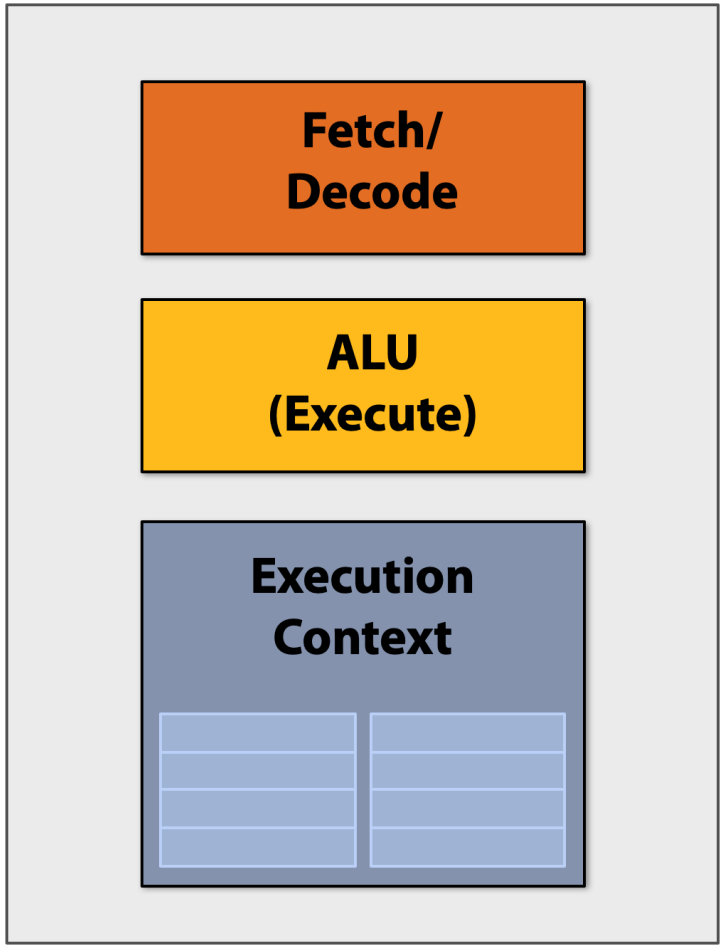
load cost:

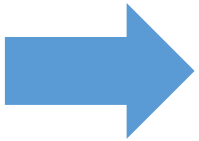
```
l1_speed * (n^3/v2 + n^3/v1)
+ dram_speed * (n^2 + n^3/b1)
```

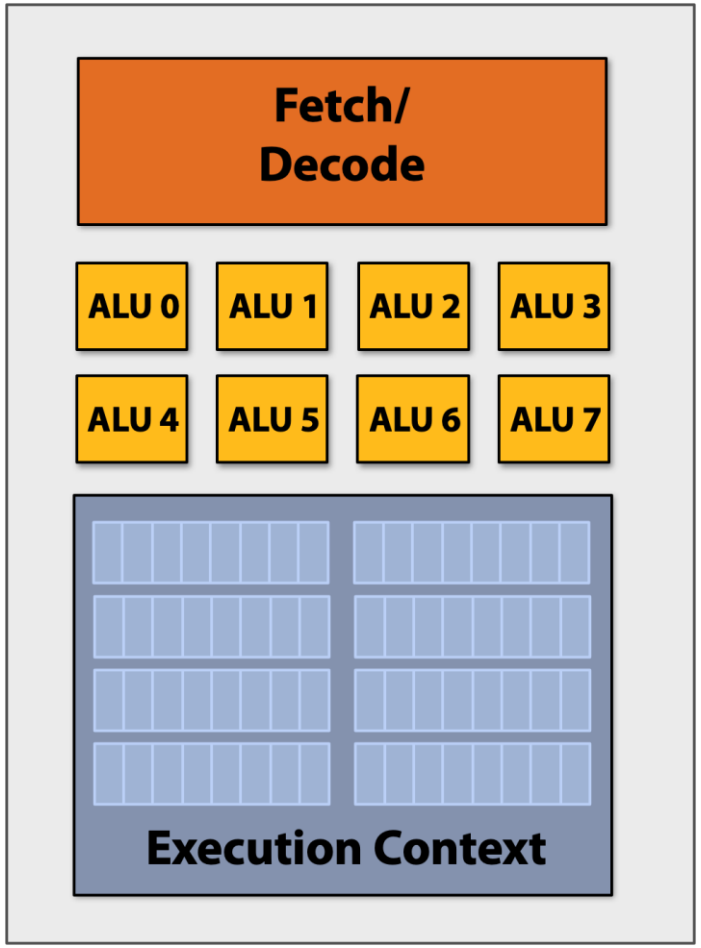
Outline

- General acceleration techniques
  - Case study: matrix multiplication on CPUs
- Overview of CUDA programming & Nvidia GPU architecture
  - Case study: 1D convolution on CUDA
  - Case study: matrix multiplication on CUDA

Single instruction, multiple data (SIMD)







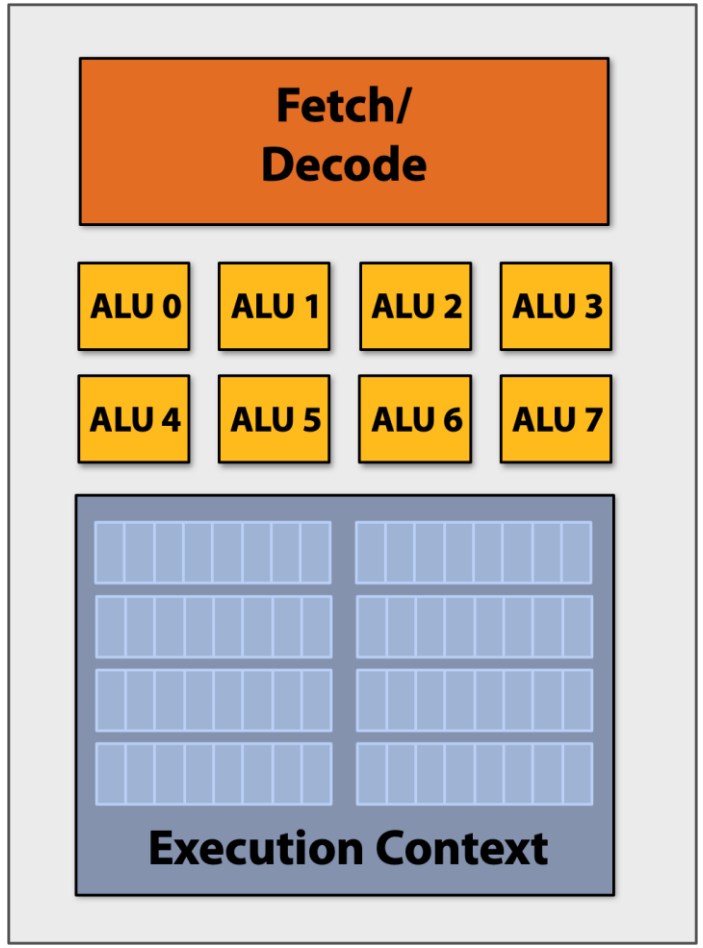
Conventional single instruction, single data processor

Modern single instruction, multiple data processor

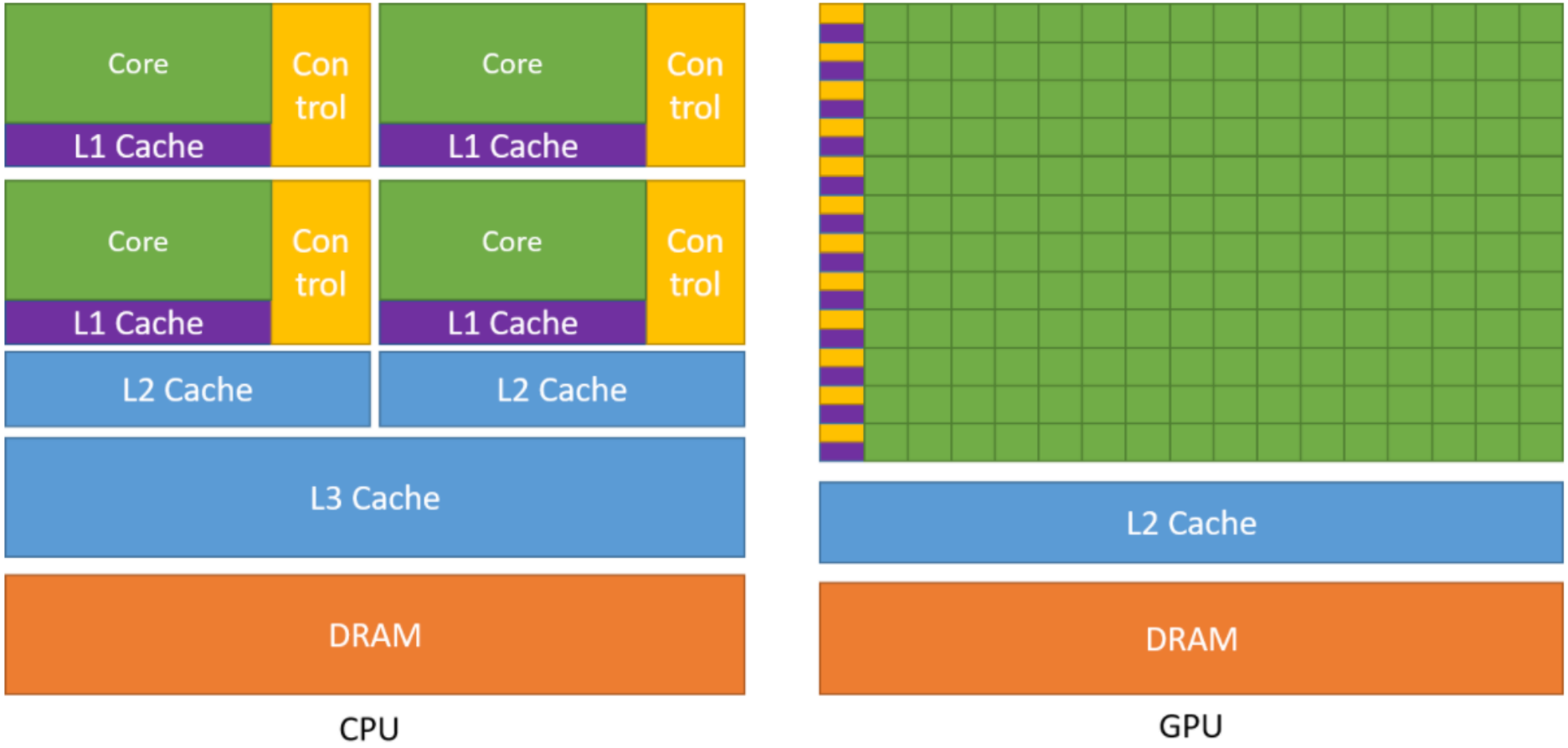
Single instruction, multiple data (SIMD)

- Same instruction broadcast and executed in parallel on all ALUs

- Add ALUs to increase compute capability



Massive parallel computing units

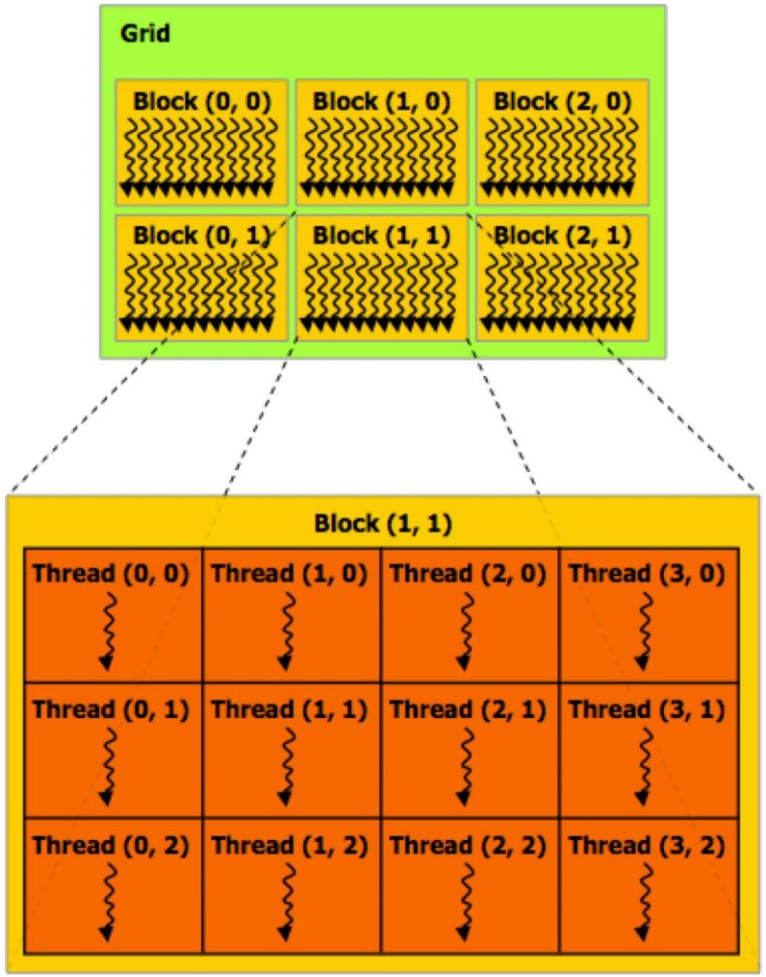


CUDA programming language

- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming on GPUs
- CUDA's abstractions closely match the capabilities/performance characteristics of modern GPUs
- Design goal: maintain low abstraction distance

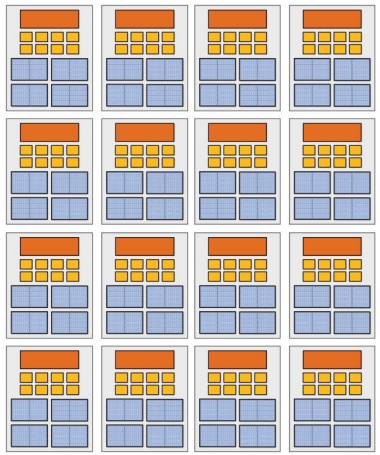
CUDA programs consist of a hierarchy of threads

- Thread IDs are up to 3-dimensional (2D example below)

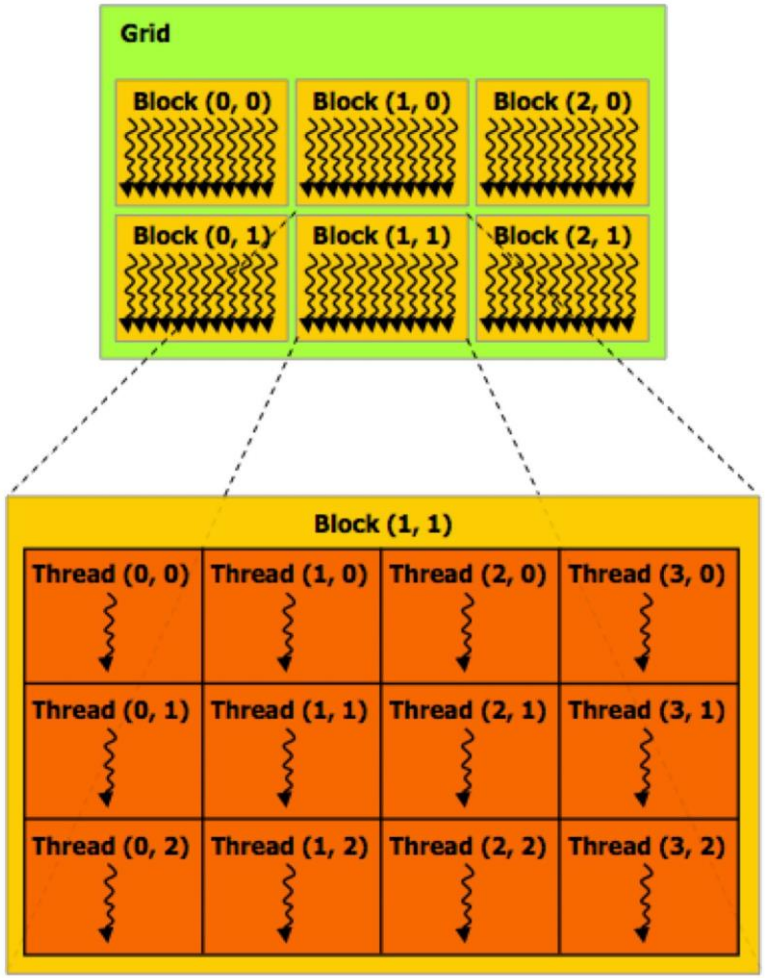


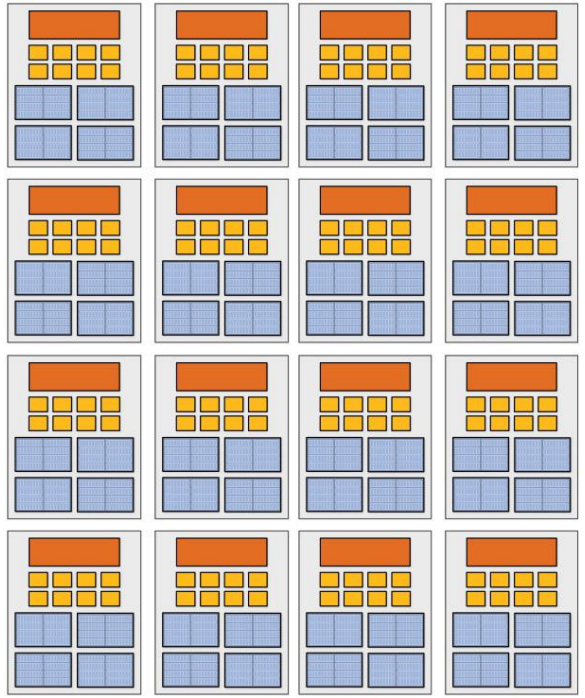

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

Run on CPU



CUDA blocks map to GPU cores



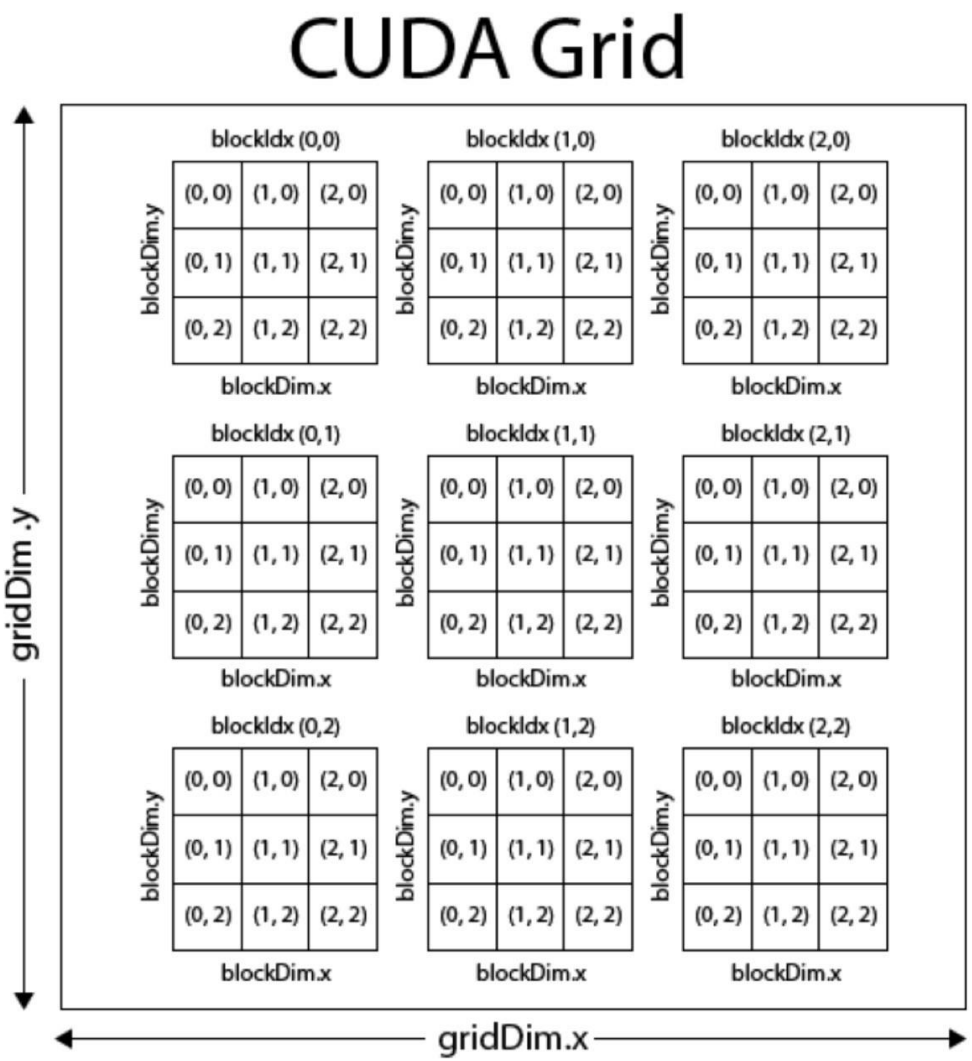


GPU

Grid, block, and thread

- gridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of a block
- threadIdx: The thread index within a block

We always have:
gridIdx = (1, 1, 1)
threadDim = (1, 1, 1)

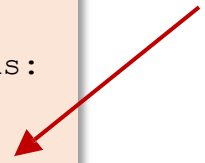


Basic CUDA syntax

Serial execution: running as part of normal C/C++ application on CPU

Host

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



Bulk launch of many CUDA threads
"launch a grid of CUDA thread blocks"
Call returns when all threads have terminated

Device

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + B[j][i];
}
```

__global__ denotes a CUDA kernel function runs on GPU

Each thread computes its overall grid thread id from its position in its block (threadIdx) and its block's position in the grid (blockIdx)

CUDA kernel: executed in parallel on multiple CUDA cores

Clear separation of host and device code

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72
// 6 blocks of 12 threads threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

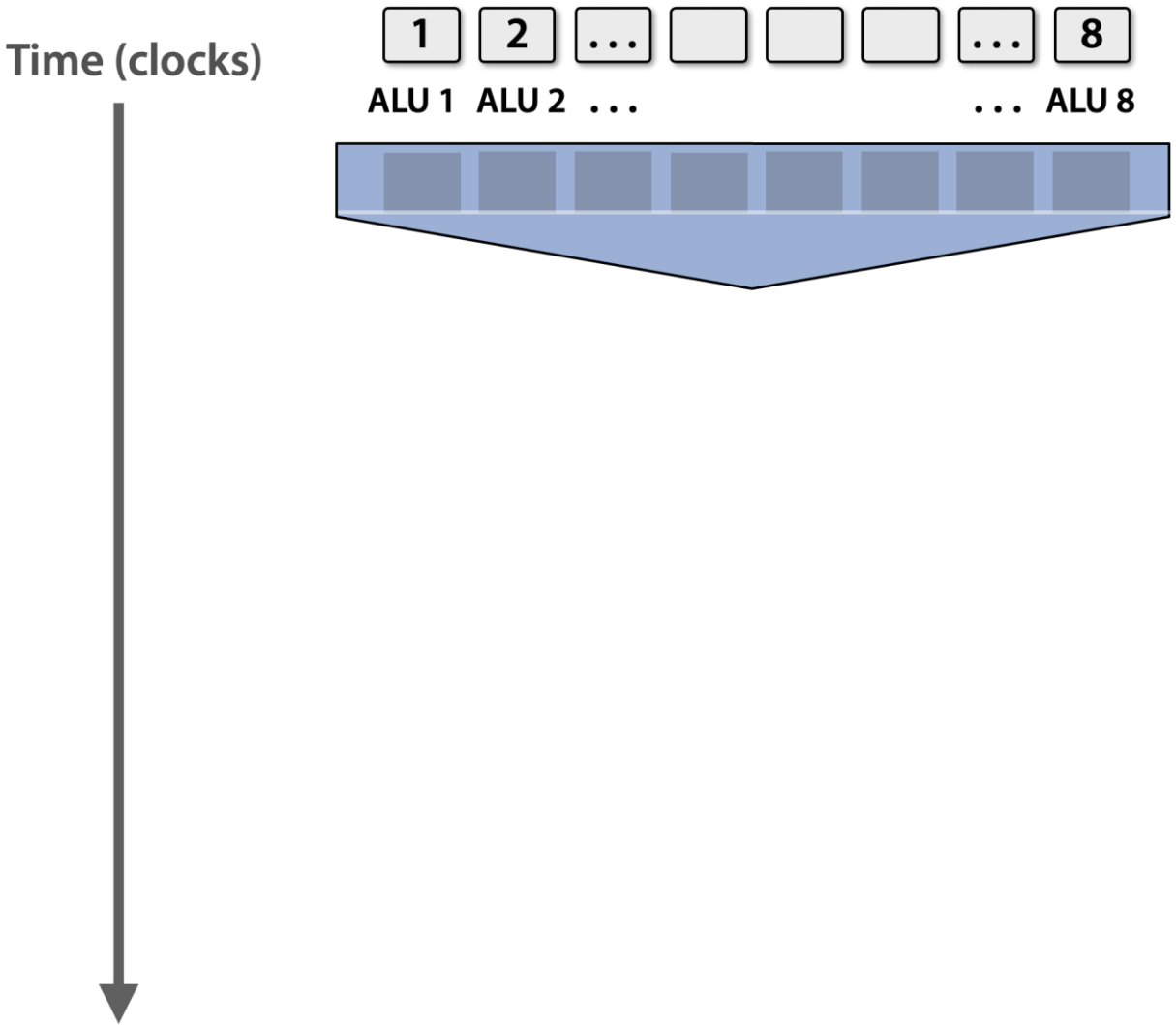
Separation of execution into host and device code is performed statically by the programmer

"Host" code : serial execution on CPU

```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

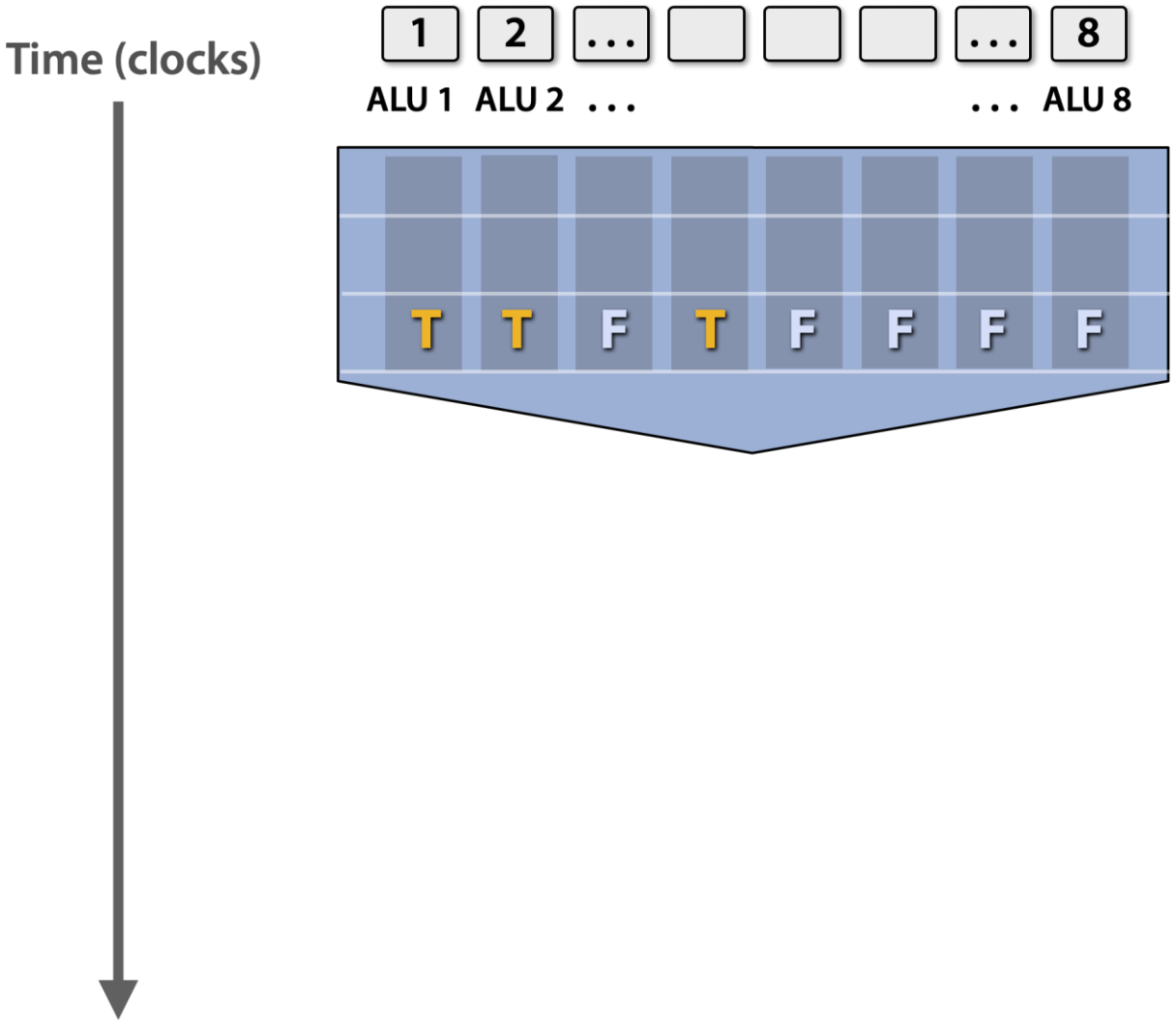
"Device" code (SIMD execution on GPU)

Conditional execution



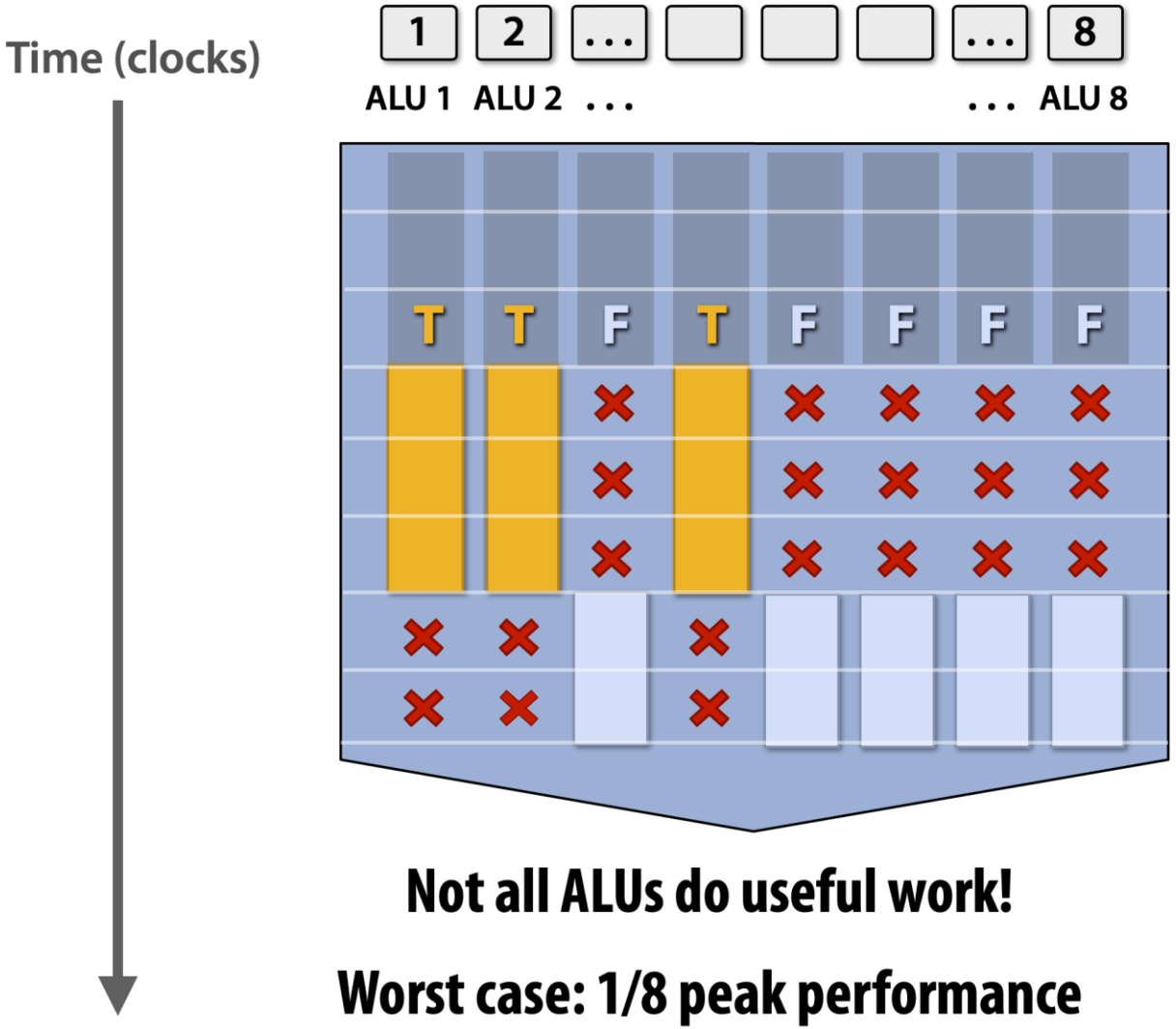

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x +
   threadIdx.x; float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Conditional execution



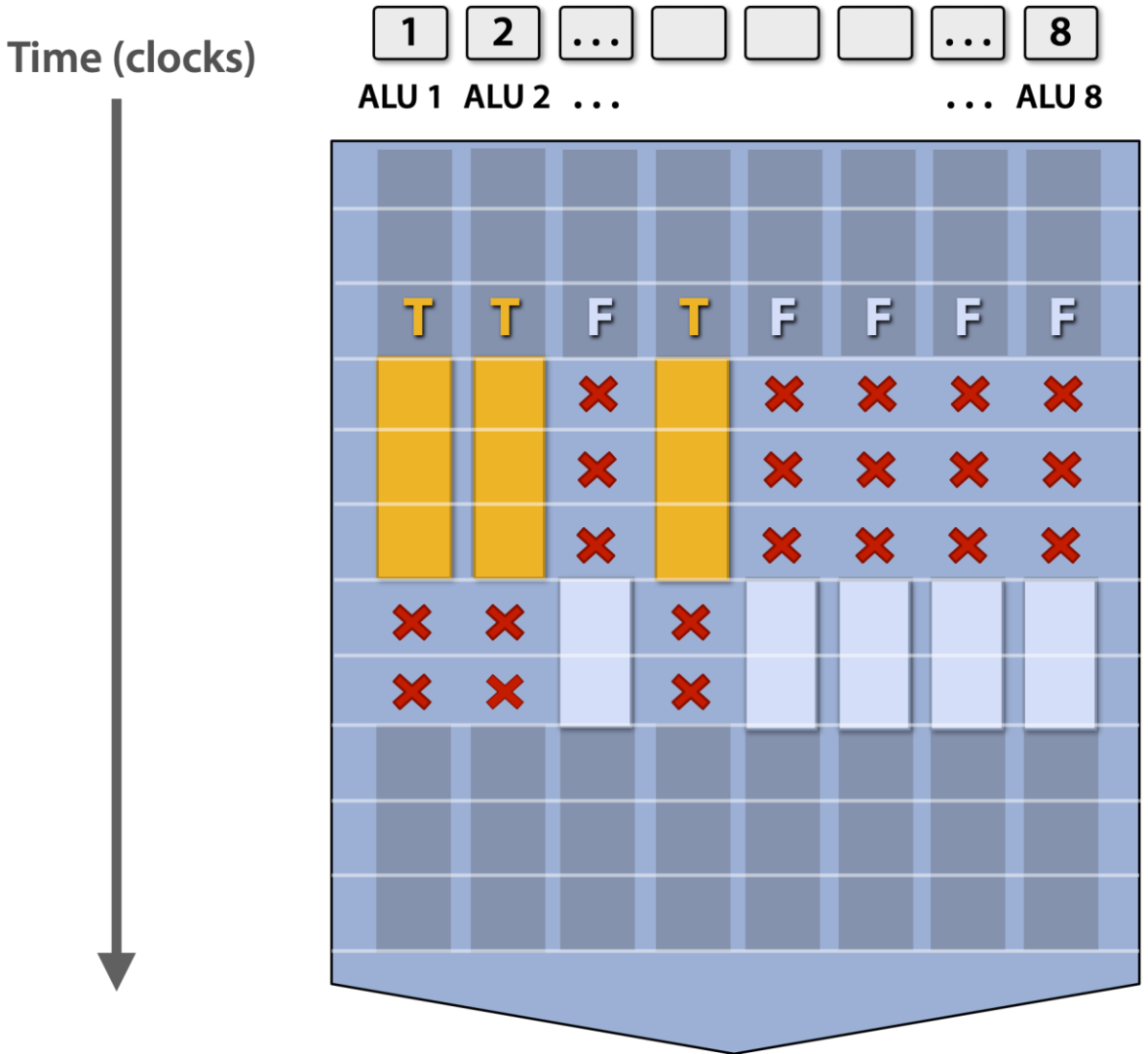

```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x +
   threadIdx.x; float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```

Mask (discard) output of ALU




```
// kernel definition
__global__ void f(float A[N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float x = A[i];
    if (x > 0) {
        x = 2.0f * x;
    } else {
        x = exp(x, 5.0f);
    }
    A[i] = x;
}
```

After branch: continue at full performance




```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;

   } else {
      x = exp(x, 5.0f);

   }
   A[i] = x;
}
```

cudaMemcpy: move data between host and device

Host (serial execution on CPU)

Host memory address space

CUDA Device (SIMD execution on GPU)

Device memory address space

```
float* A = new float[N];

// populate host address space pointer A
for (int i=0 i<N; i++)
   A[i] = (float)i;

int bytes = sizeof(float) * N
float* deviceA;                  // allocate buffer in
cudaMalloc(&deviceA, bytes);     // device address
                                 space
// populate deviceA
cudaMemcpy(deviceA, A, bytes, cudaMemcpyHostToDevice);

// note: deviceA[i] is an invalid operation here (cannot
// manipulate contents of deviceA directly from host.
// Only from device code.)
```

Distinct host and device address spaces

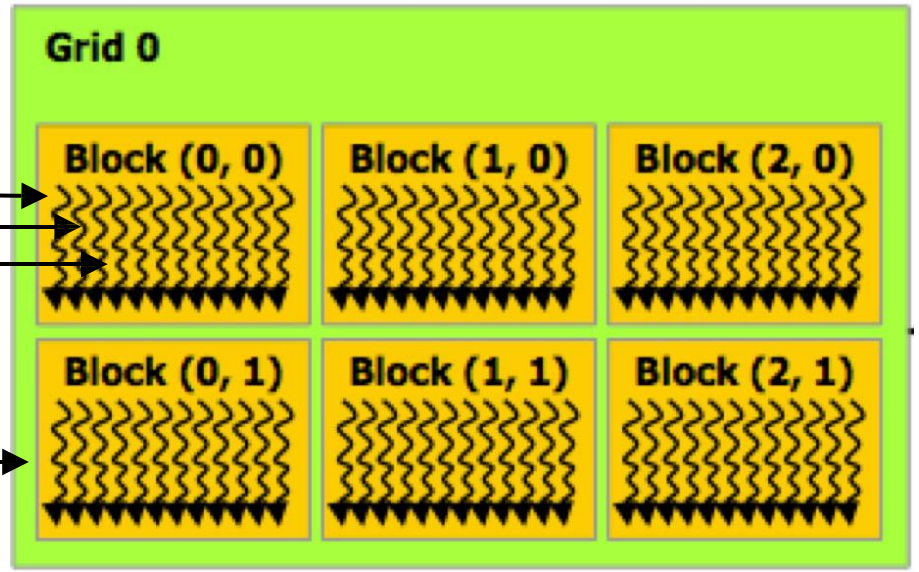
- Cannot access host memory from device
- Cannot access device memory from host

CUDA device memory model

- Three distinct types of memory available to kernels

Per-thread private memory (readable/writable by thread)

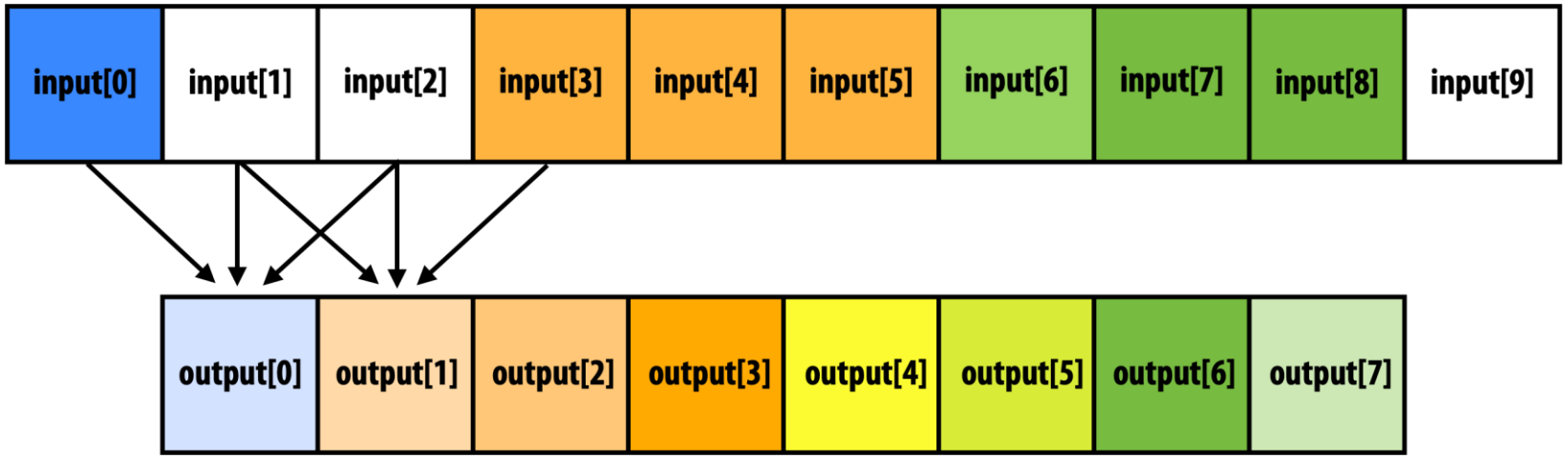
Per-block shared memory (readable/writable by all threads in a block)



Device global memory (readable/writable by all threads)

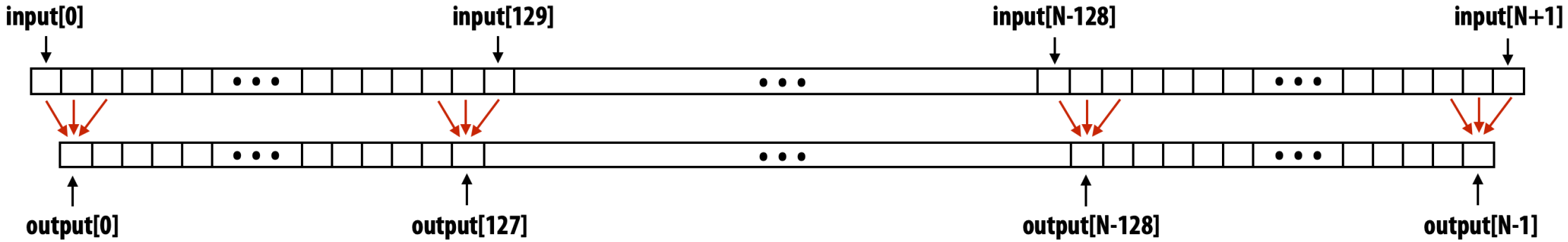
Why shared memory?
Enable cooperation across threads in a block

CUDA programming example: 1D convolution



output[i] = (input[i] + input[i+1] + input[i+2]) / 3.f;

1D convolution (v1)




```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += input[index + i];

   output[index] = result / 3.f;
}
```

each thread computes result for one element

1D convolution (reused shared memory)

```
int N = 1024 * 1024
cudaMalloc(&devInput, sizeof(float) * (N+2) );  // allocate array in device memory
cudaMalloc(&devOutput, sizeof(float) * N);      // allocate array in device memory

// property initialize contents of devInput here ...

convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

```
#define THREADS_PER_BLK 128

__global__ void convolve(int N, float* input, float* output) {

   int index = blockIdx.x * blockDim.x + threadIdx.x;  // thread local variable

   __shared__ float support[THREADS_PER_BLK+2];        // per-block allocation
   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK + threadIdx.x] = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result / 3.f;
}
```

All threads cooperatively load block's support region from global into shared memory (total of 130 loads instead of 3 * 128 loads)

barrier (all threads in block)

each thread computes result for one element

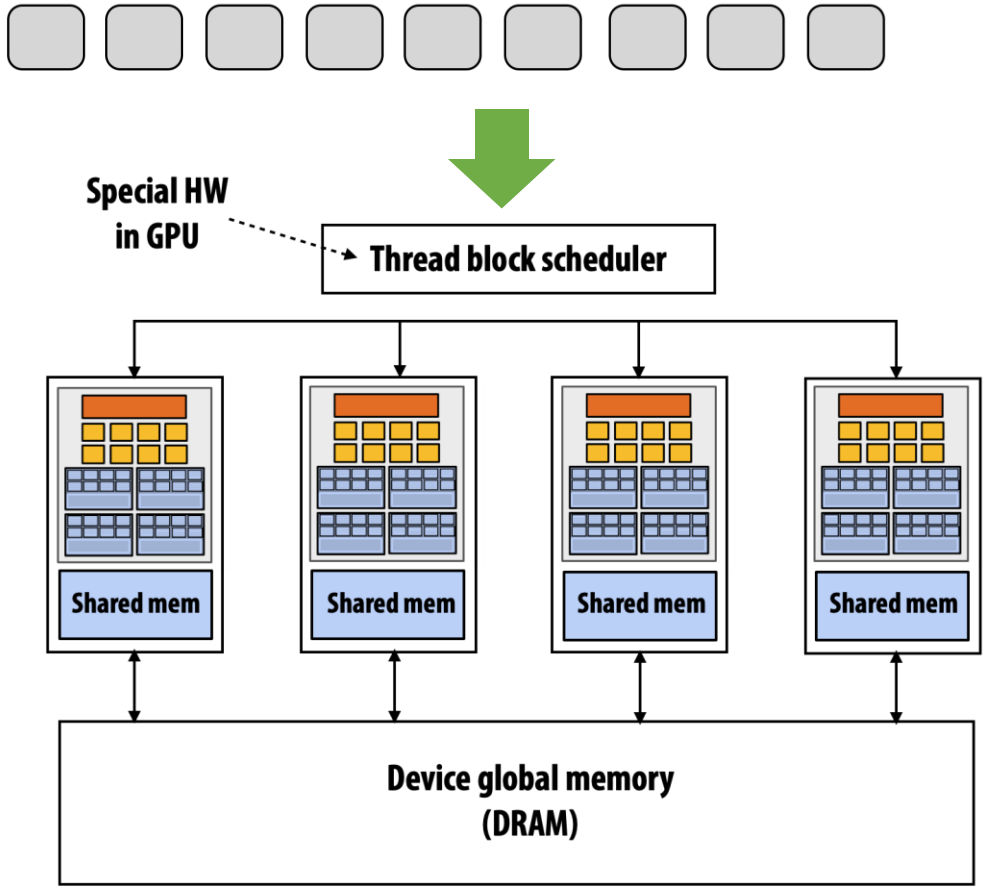
CUDA thread block scheduling

- **Major CUDA assumption**: threadblocks can be executed in any order (no dependencies between threadblocks)

- GPU maps threadblocks to cores using a dynamic scheduling policy that respects resource requirements

Grid of 8K convolve thread blocks (specified by kernel launch)
Block requirements:

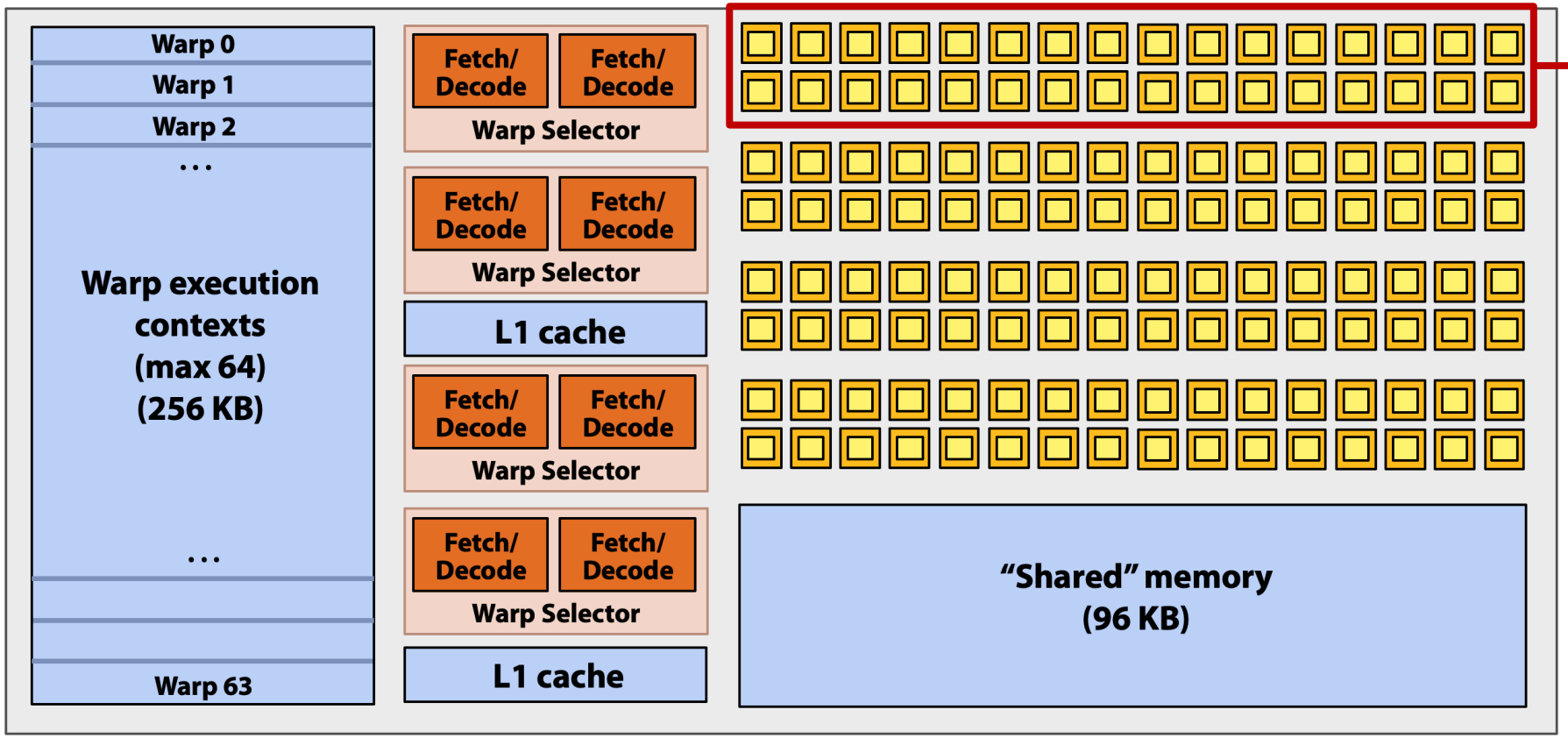
- 128 threads
- 520 bytes of shared memory
- 1024 bytes of local memory



A GPU core: streaming multiprocessor (SMM)

SMM resource limits:

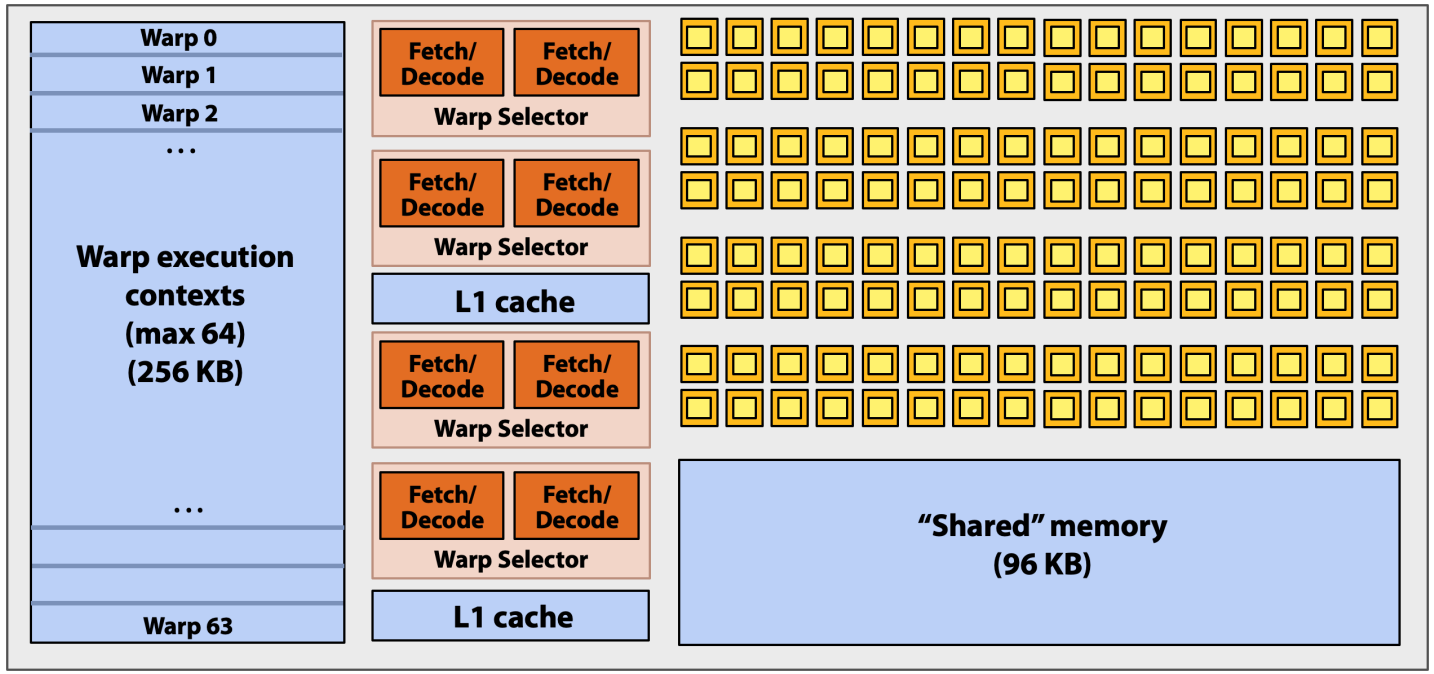
- Max warp execution contexts: 64 (up to 64 * 32 = 2K total CUDA threads)
- 96 KB of shared memory



SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

Running a thread block on an SMM

- Warp: A group of 32 CUDA threads shared an instruction stream.
  - A convolve thread block is executed by 4 warps (4 warps x 32 threads / warp = 128 threads)
- SMM operation each clock:
  - Select up to four runnable warps from 64 resident on an SMM (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)

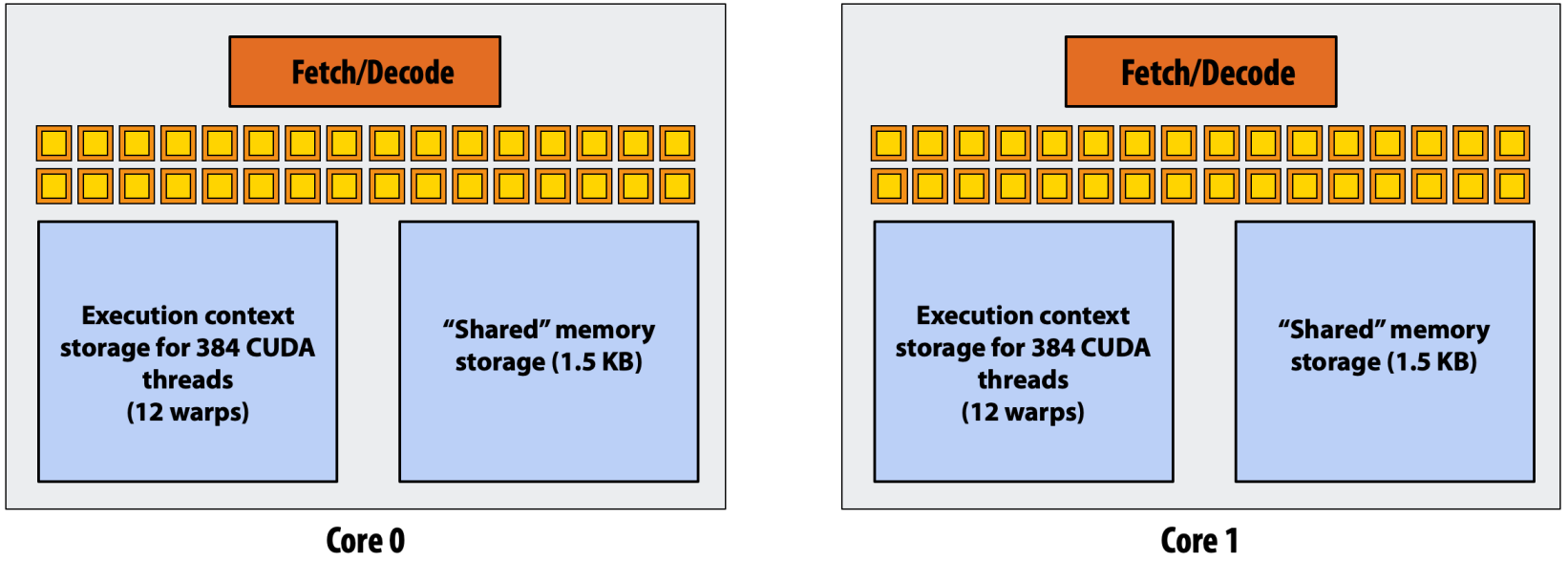


Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 520 bytes of shared memory
- Assume the host side launches 1000 thread blocks

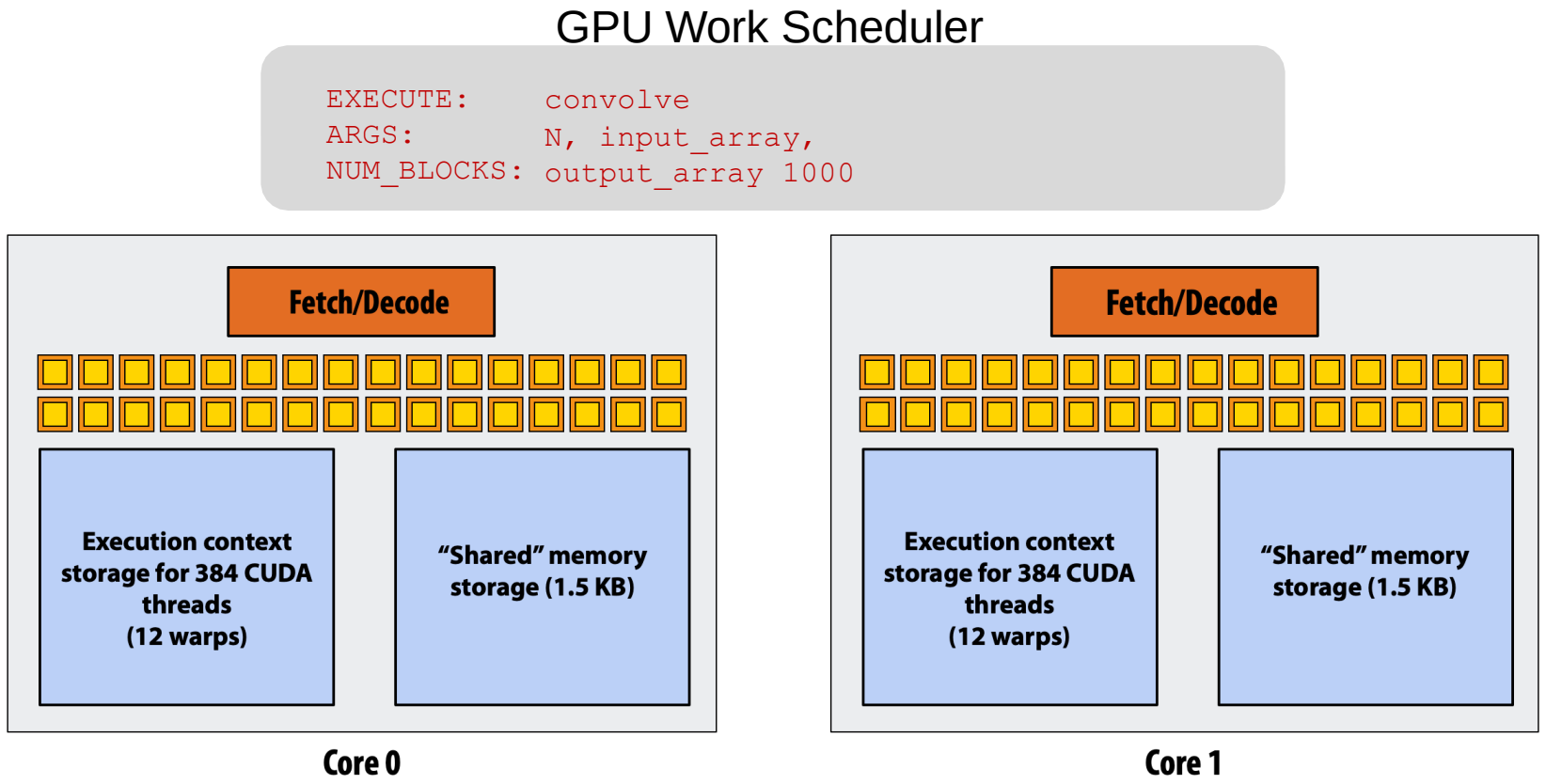
```
#define THREADS_PER_BLK 128
convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, input_array, output_array);
```

- Run the program on a two-SMM GPU



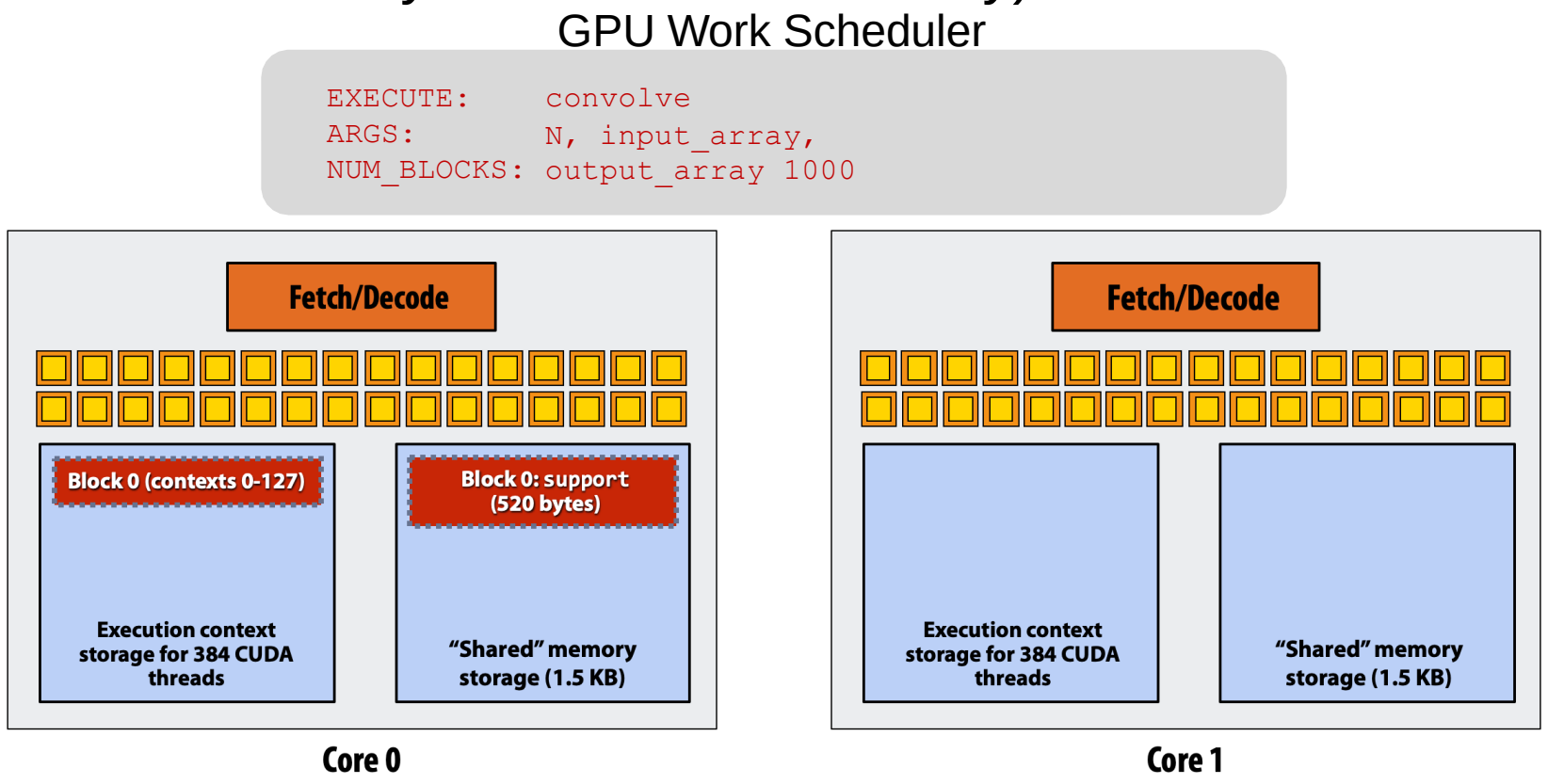
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 1: host sends CUDA kernel to device



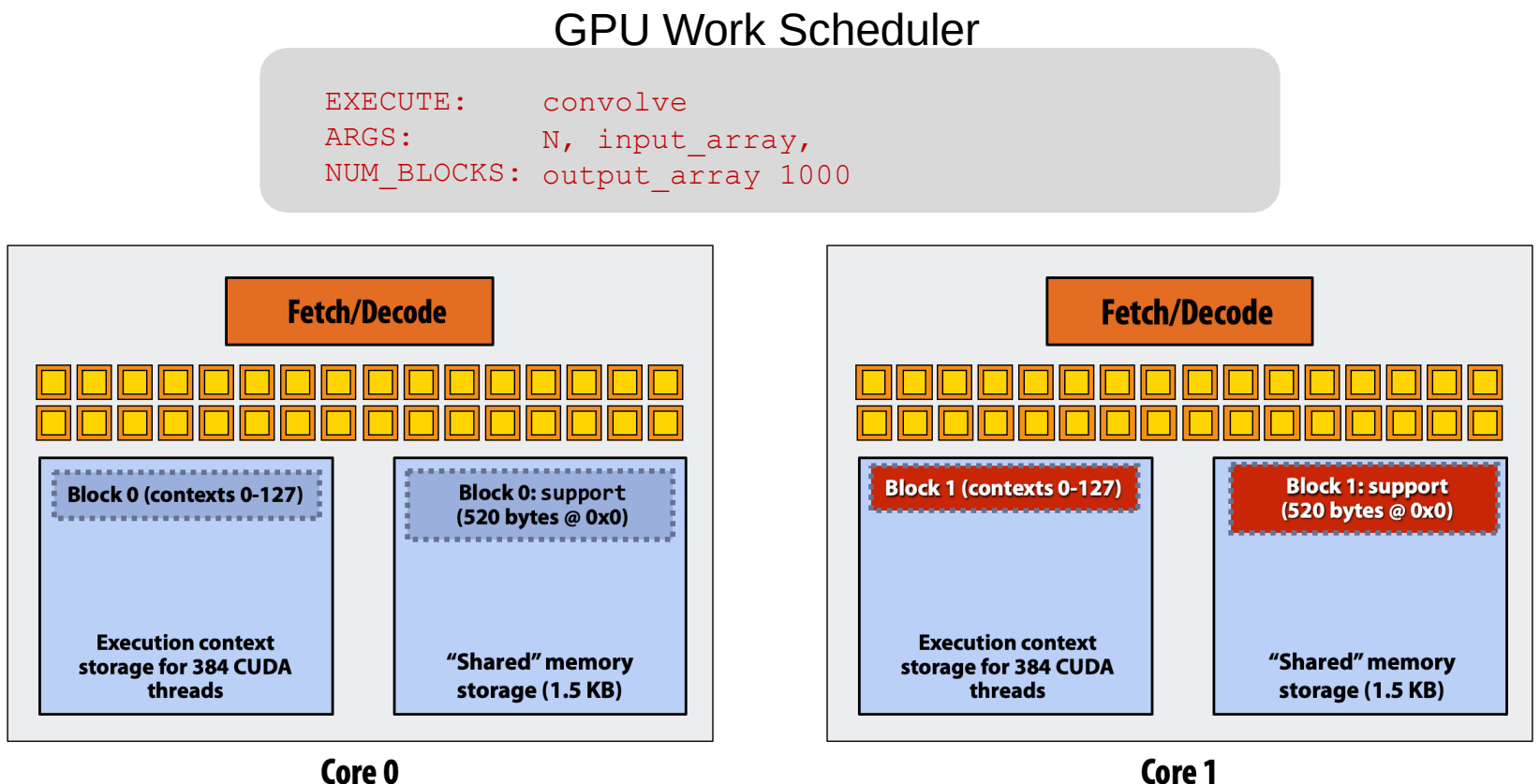
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 2: scheduler maps block 0 to core 0 (reserves execution contexts for 128 threads and 520 bytes of shared memory)



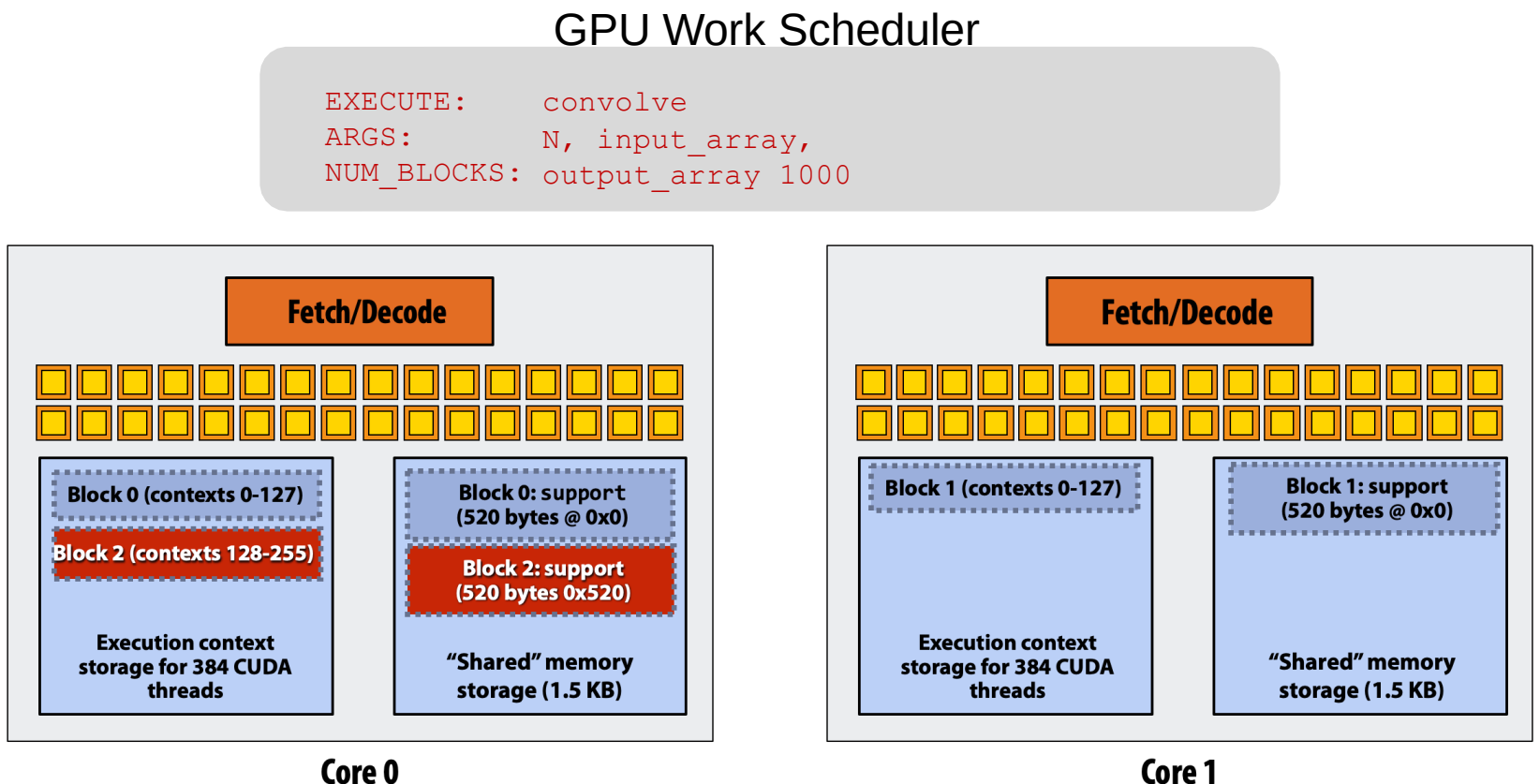
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 3: scheduler continues to map blocks to available execution contexts



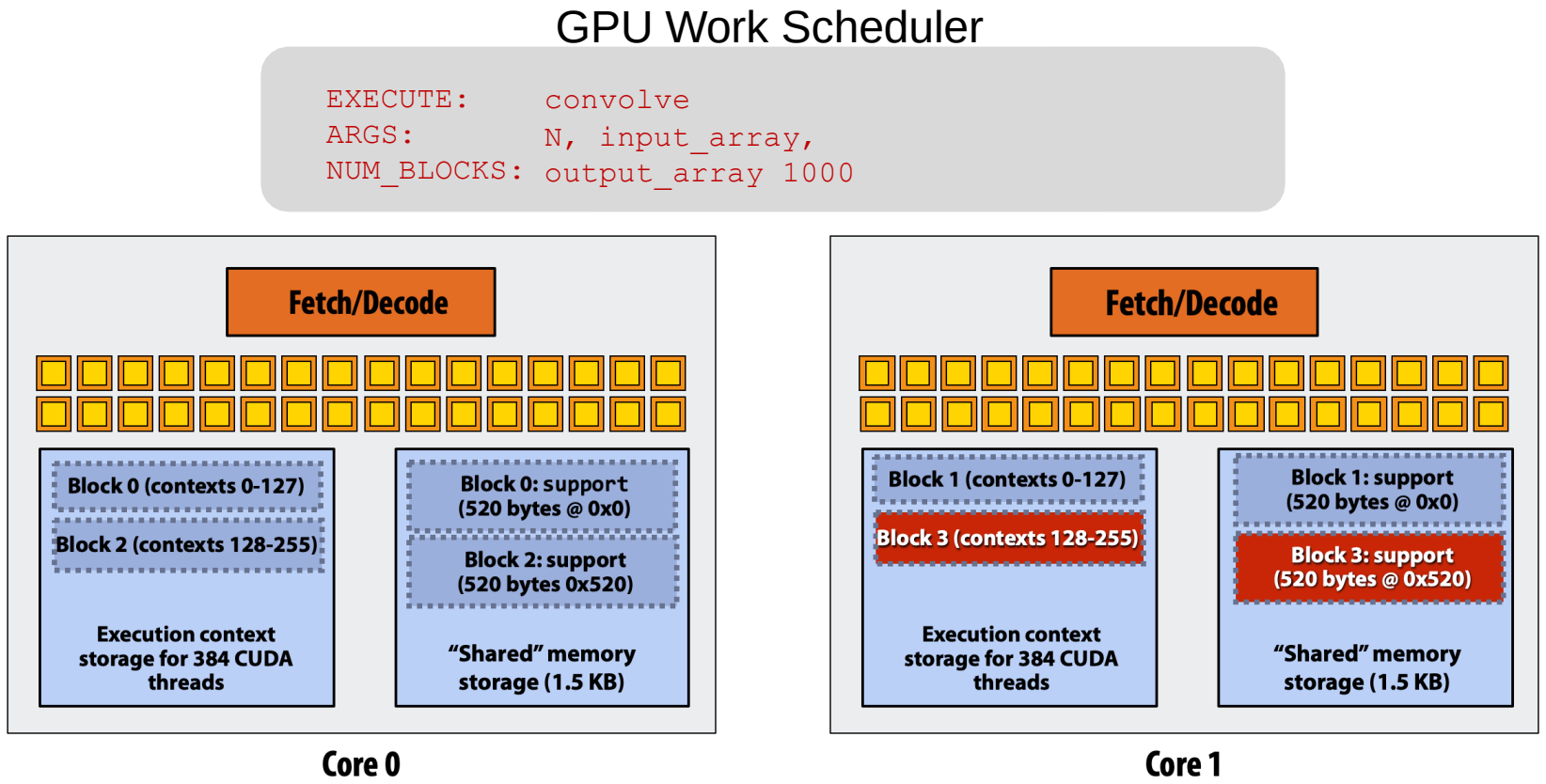
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 3: scheduler continues to map blocks to available execution contexts



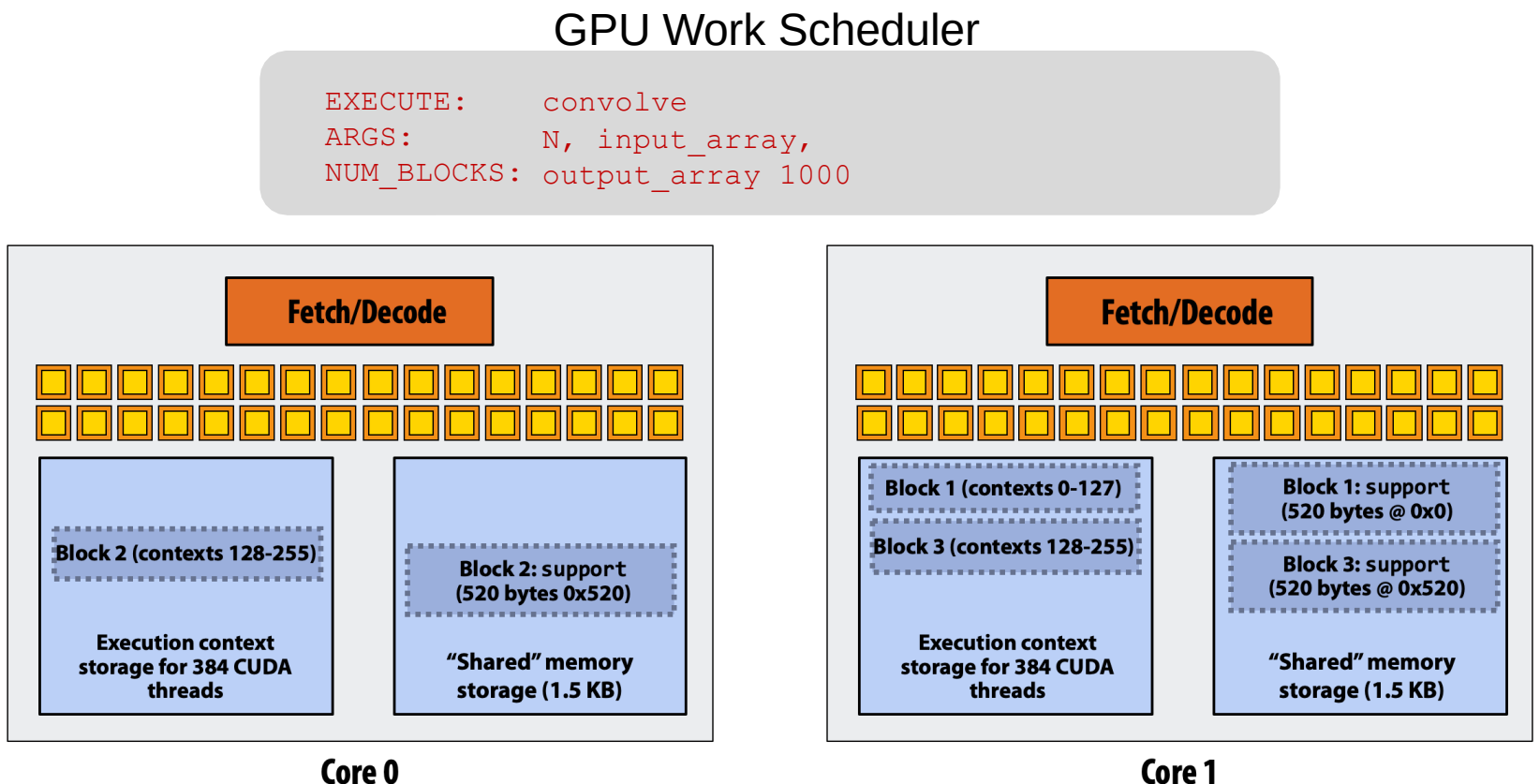
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- (third block won't fit due to insufficient shared storage 3 x 520B > 1.5KB)



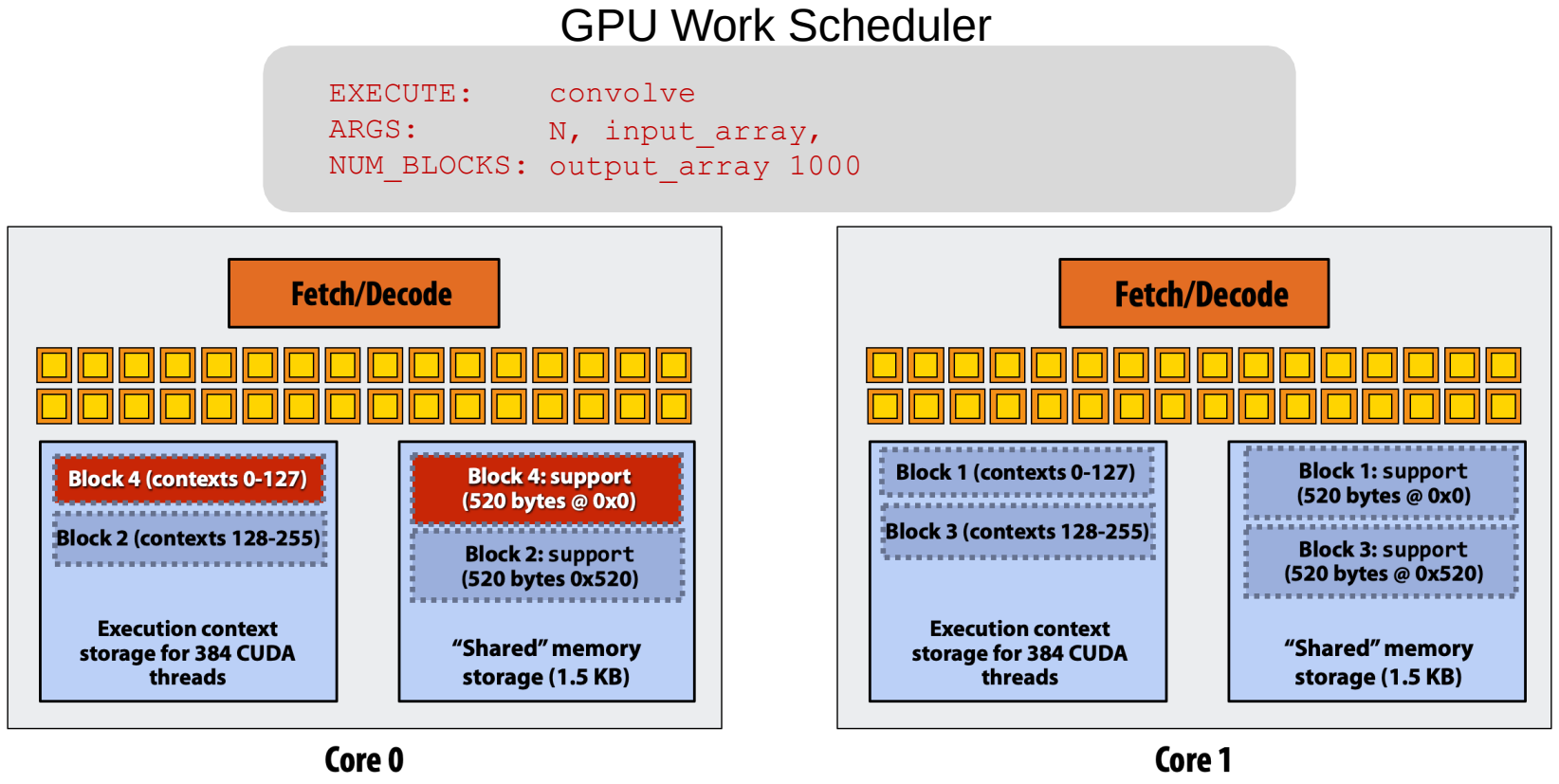
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 4: thread block 0 completes on core 0



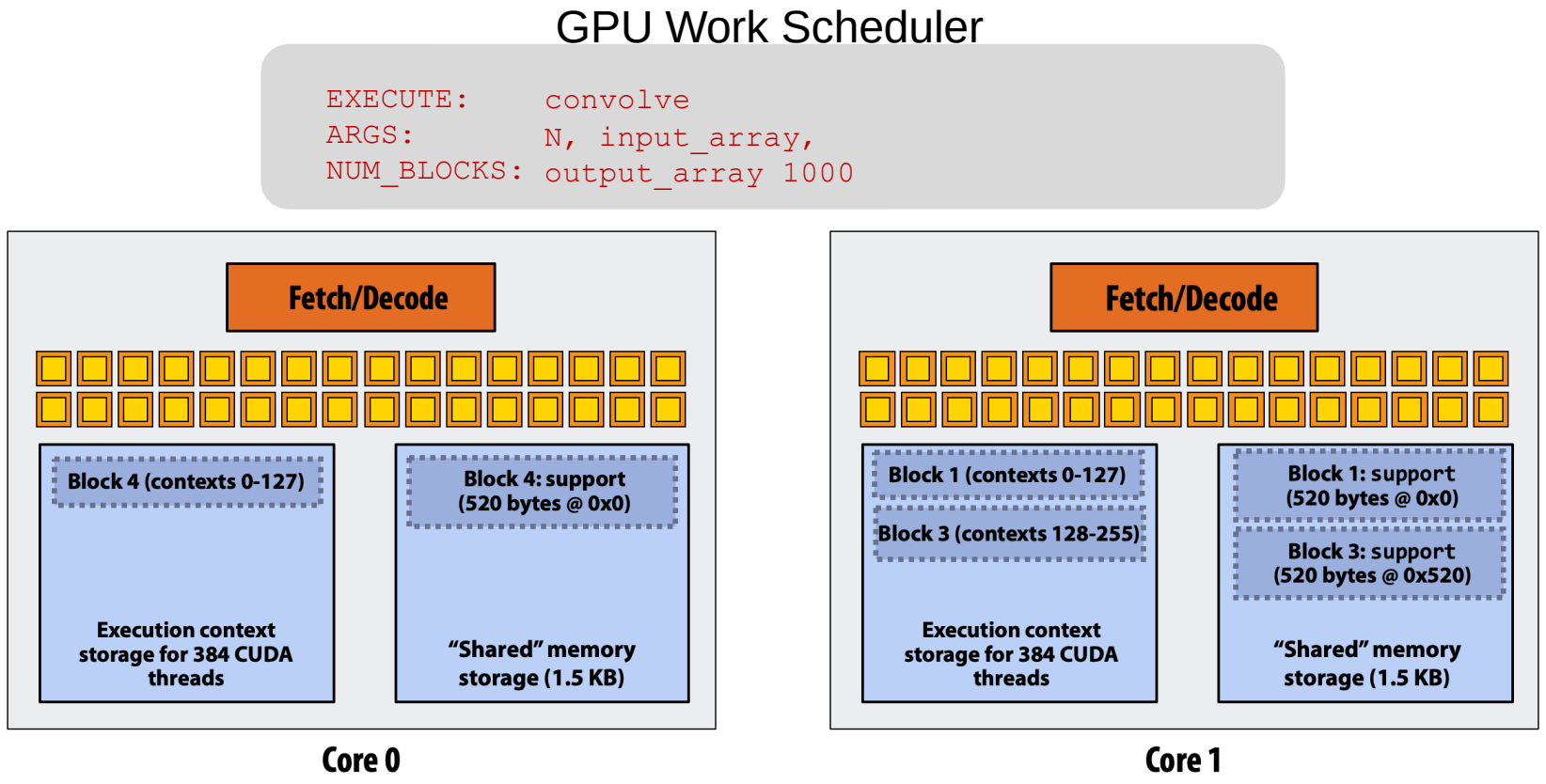
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 5: thread block 4 is scheduled on core 0 (mapped to execution contexts 0-127)



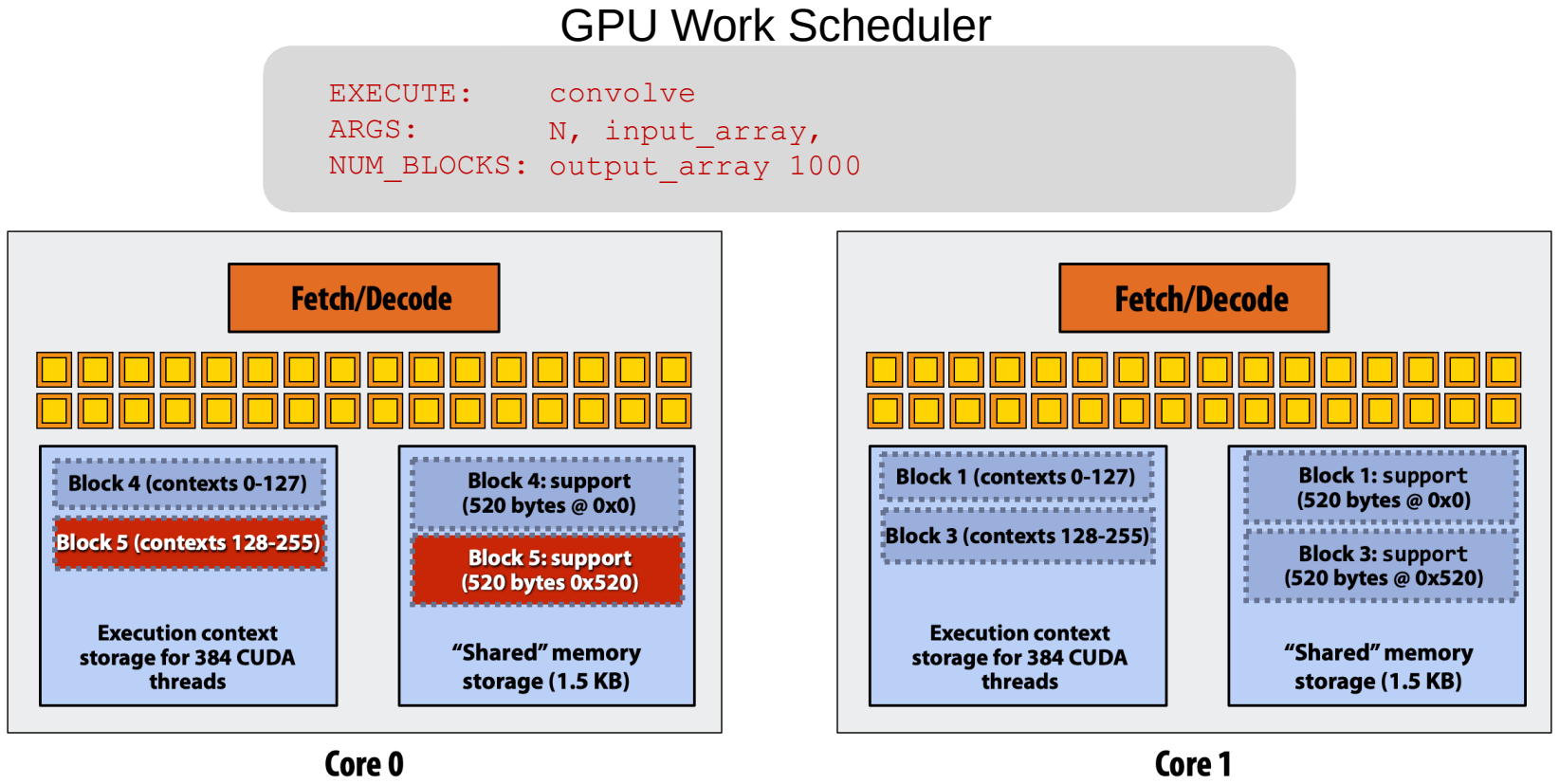
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 6: thread block 2 completes on core 0



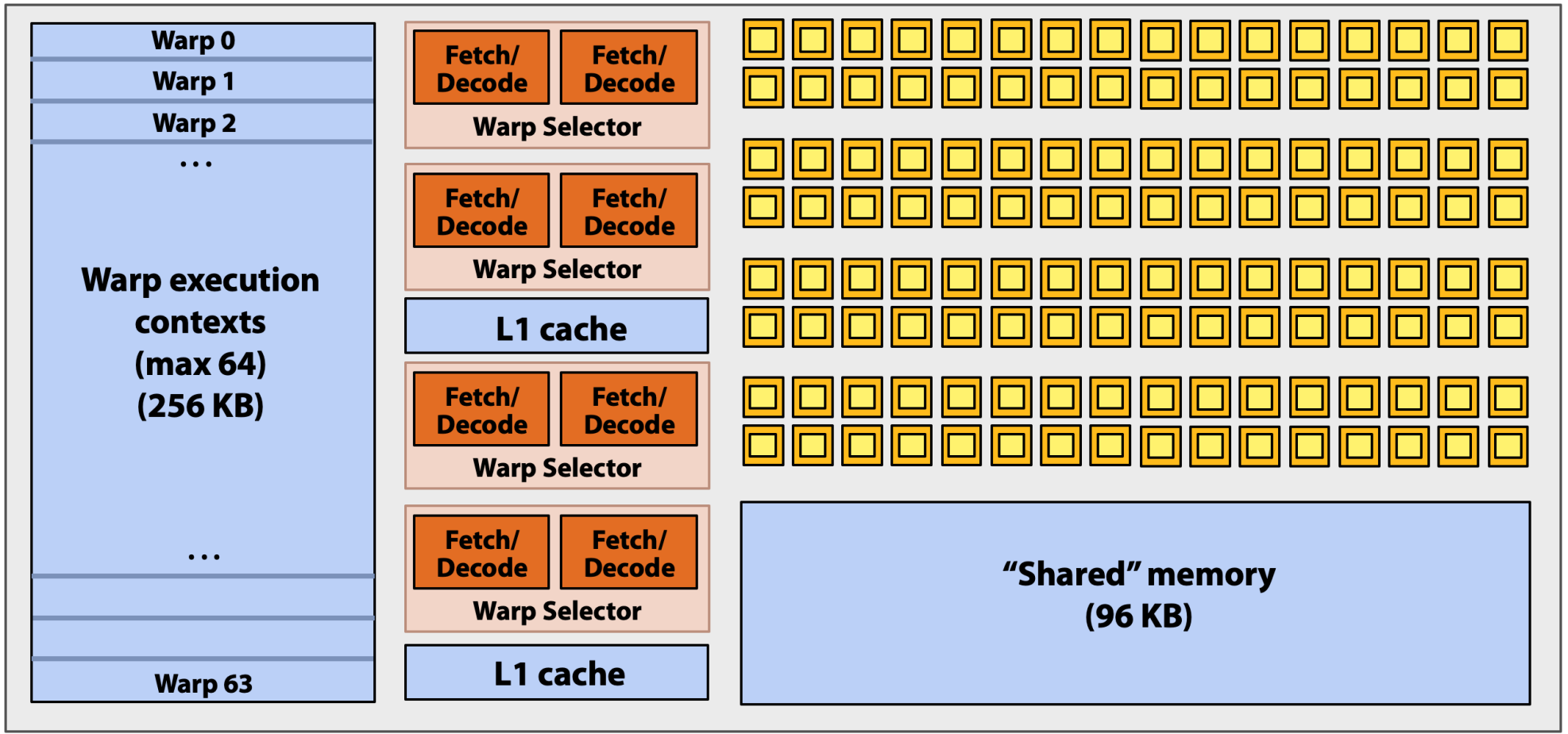
Running a CUDA kernel

- Convolve kernel's requirement:
  - Each thread block execute 128 CUDA threads
  - Each thread block allocate 130 * 4 = 512 bytes of shared memory
- Step 7: thread block 5 is scheduled on core 0 (mapped to execution contexts 128-255)

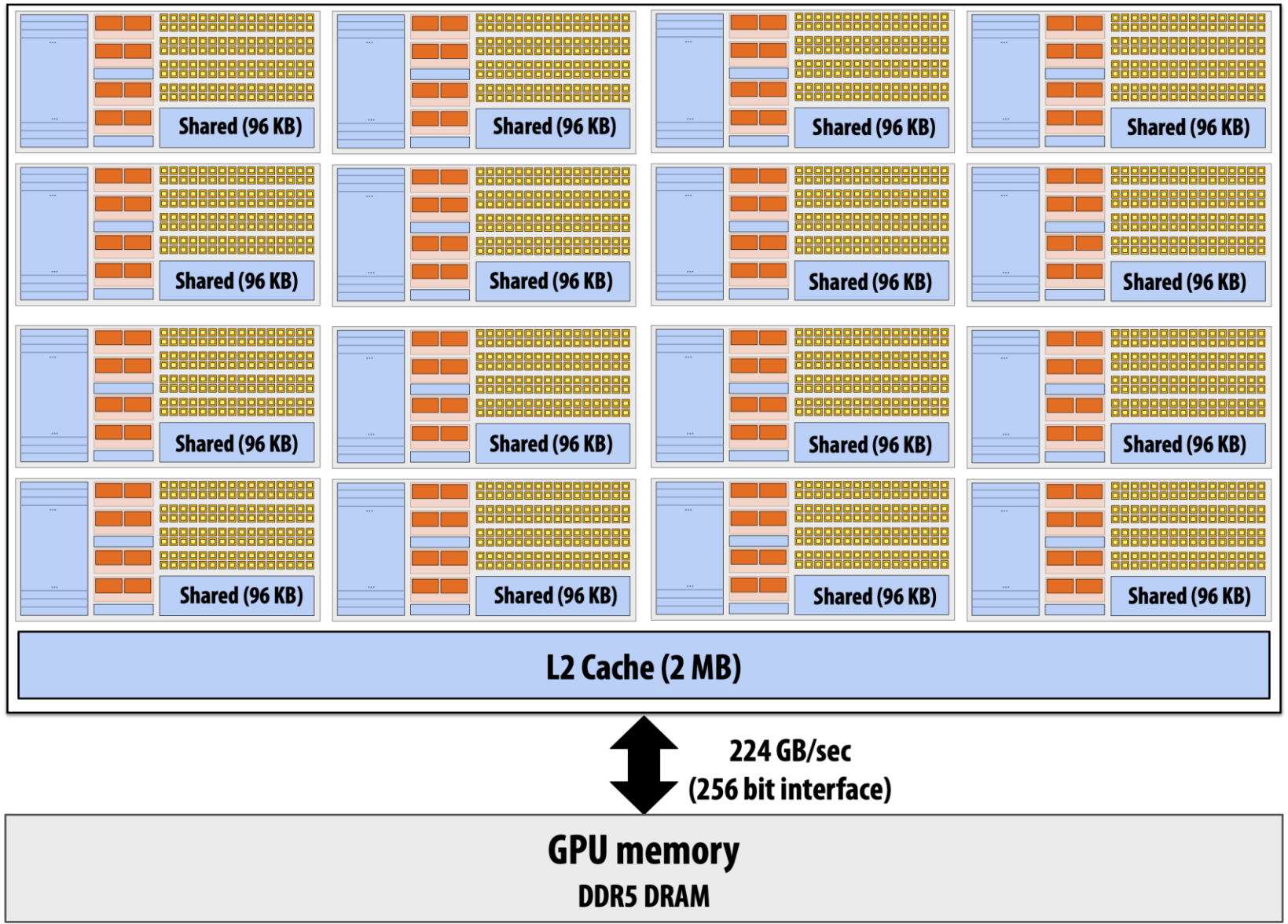


Recall: an SMM on a NVIDIA GTX 980 (2014)

- SMM resource:
  - Map warp execution contexts: 64 (64 * 32 = 2048 total CUDA threads)
  - 96 KB of shared memory



NVIDIA GTX 980 contains 16 SMMs



1.1 GHz clock

16 SMM cores per chip

16 x 4 warps x 32 threads / warp
= 2048 SIMD mul-add ALUs
= 4.6 TFLOPs

Up to 16 x 64 = 1024 interleaved warps per chip
(32,768 CUDA threads / chip)

GTX 980 (2014) -> H100 (2022)

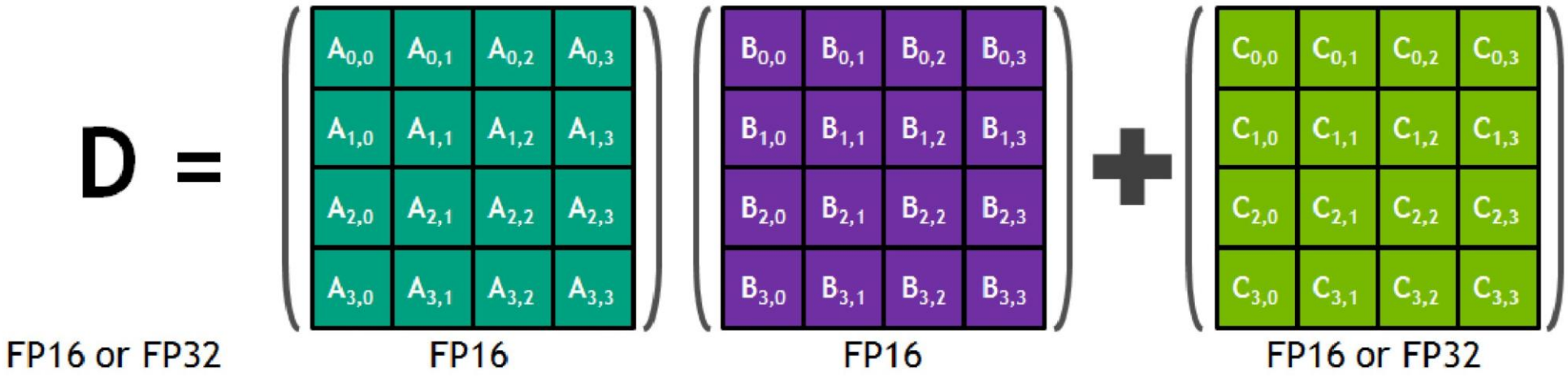
- SMMs remain the same
  - Clock speed: 1064 MHz -> 1110 MHz
  - Map warps per SMM: 64 -> 64
  - Threads per warp: 32 -> 32
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)

- Streaming multiprocessors: 16 SMMs -> 132 SMMs
- Peak performance 4.6 TFLOPs -> 1000 TFLOPs (mainly because of tensor cores)

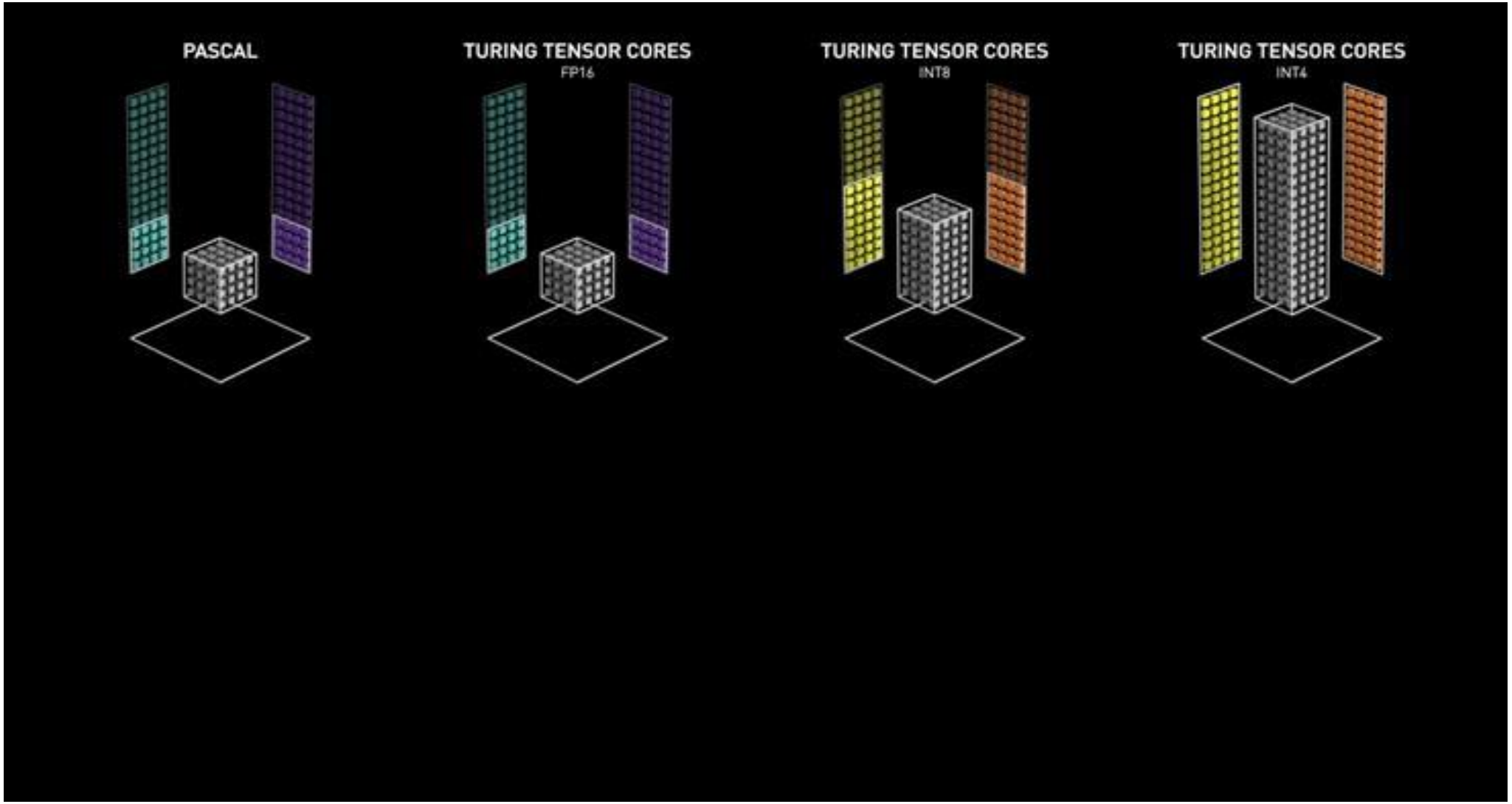
H100 architecture with tensor cores

Tensor cores

- Matrix multiplication unit in SMM



Tensor cores

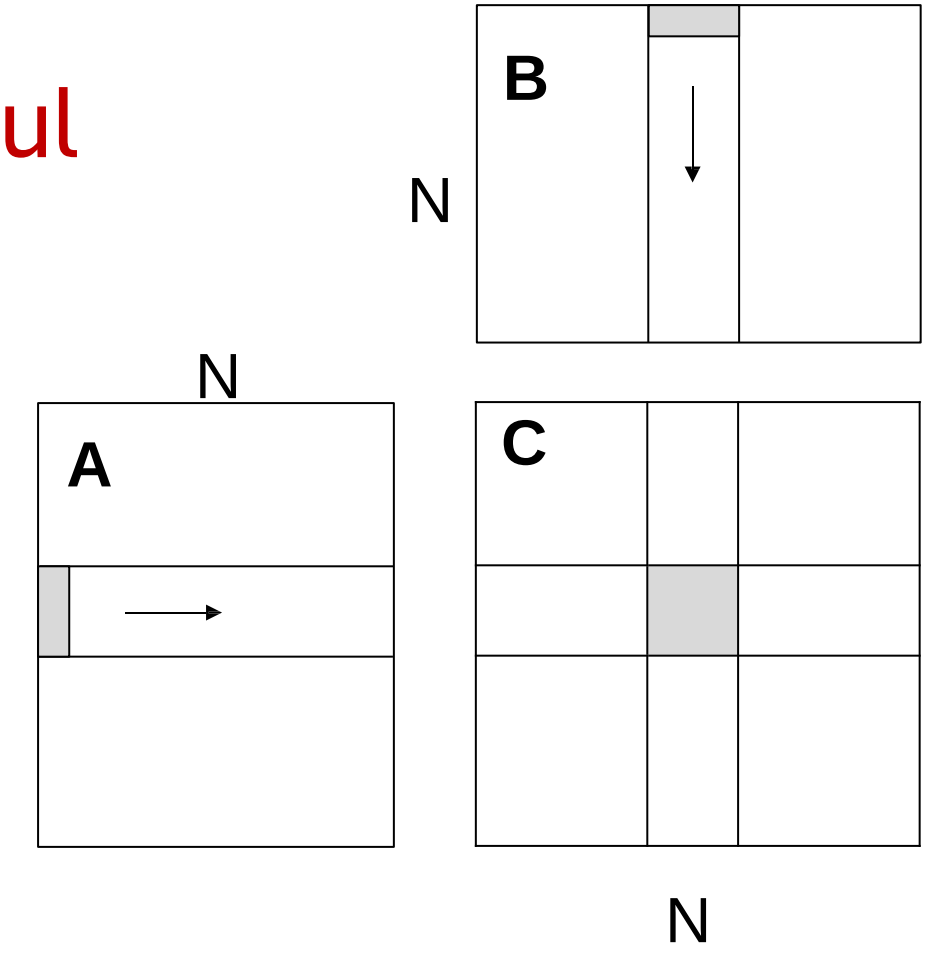


Vanilla implementation of Matmul

- Compute $C = A \times B$
- 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;
}
```

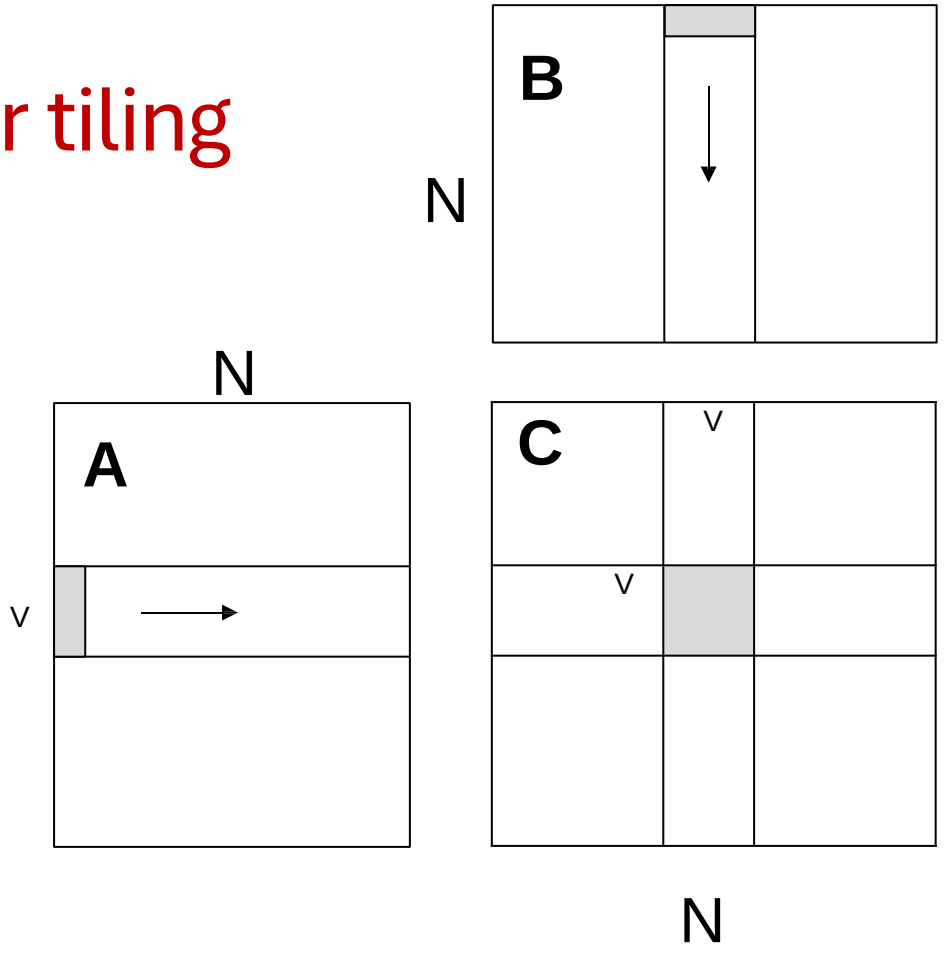


Global memory access per thread: 2*N
Number of threads: N^2
Total global memory access: $2N^3$

Optimization 1: thread-level register tiling

- Compute $C = A \times B$
- Each thread computes a V x V submatrix

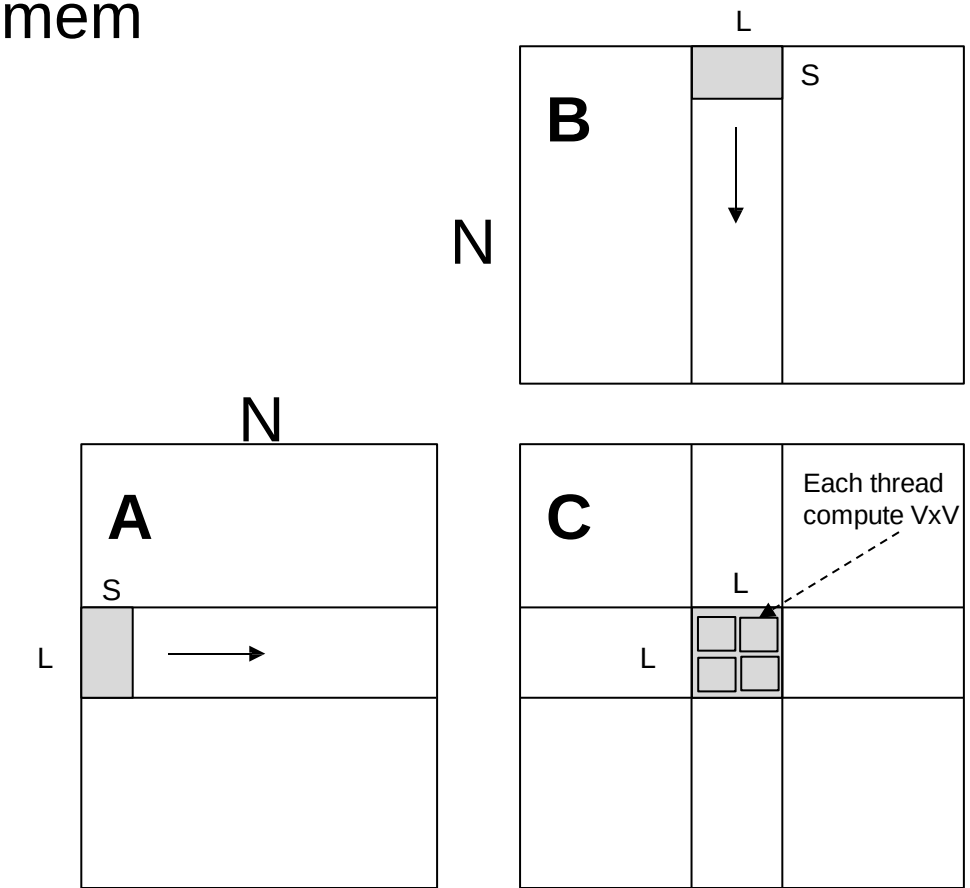
```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  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[:];
}
```



Global memory access per thread: $2NV$
Number of threads: N^2 / V^2
Total global memory access: $2N^3 / V$

Optimization 2: block-level shared memory tiling

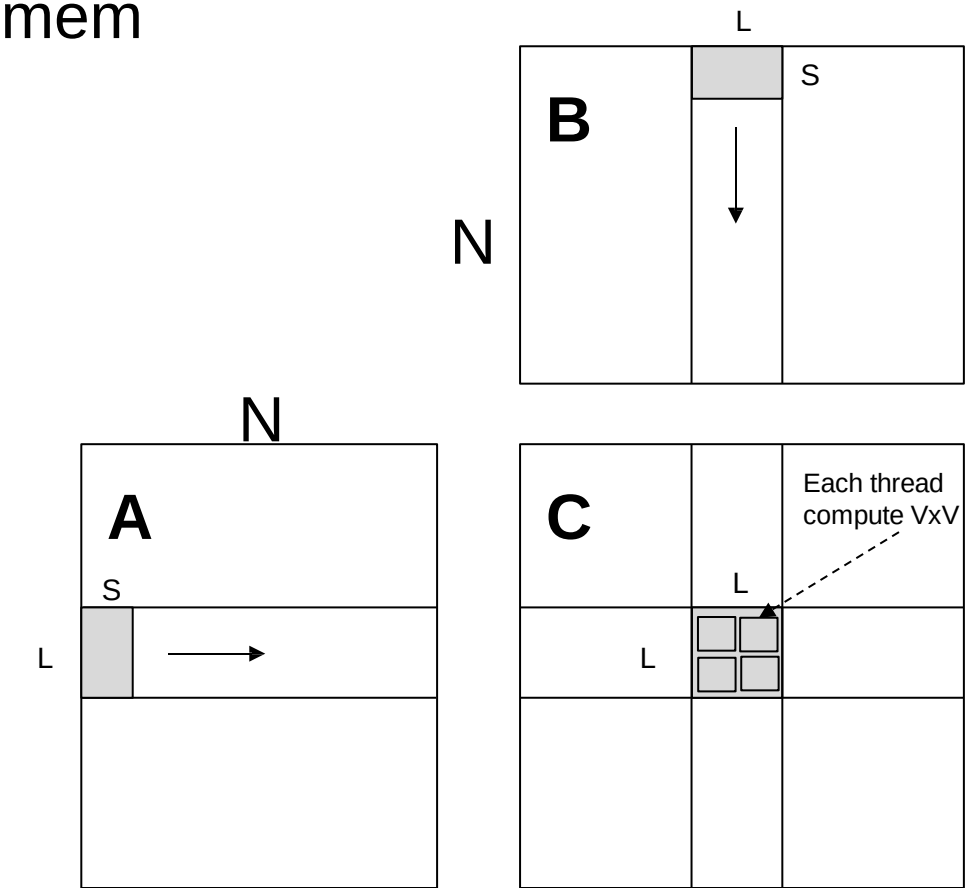
- A block computes a L x L submatrix
- A thread computes a V x V submatrix and reuses the matrices in shared mem




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

Optimization 2: block-level shared memory tiling

- 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) {
    }
  }
  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[:];
}
```

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: $2N^3/V$

More software/hardware optimizations

- CUDA compute graph
- Libraries: Eigen, MKL, BLAS, cuDNN, cuBLAS
- Google TPUs
- ML compilers

Papers to read for next lecture

- Scaling Distributed Machine Learning with the Parameter Server. OSDI 2014

- Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022