Lecture 4:
CUDA C

Modern Parallel Computing
John Owens
EEC 289Q, UC Davis, Winter 2018

Recap: Programming Model Big Idea #1

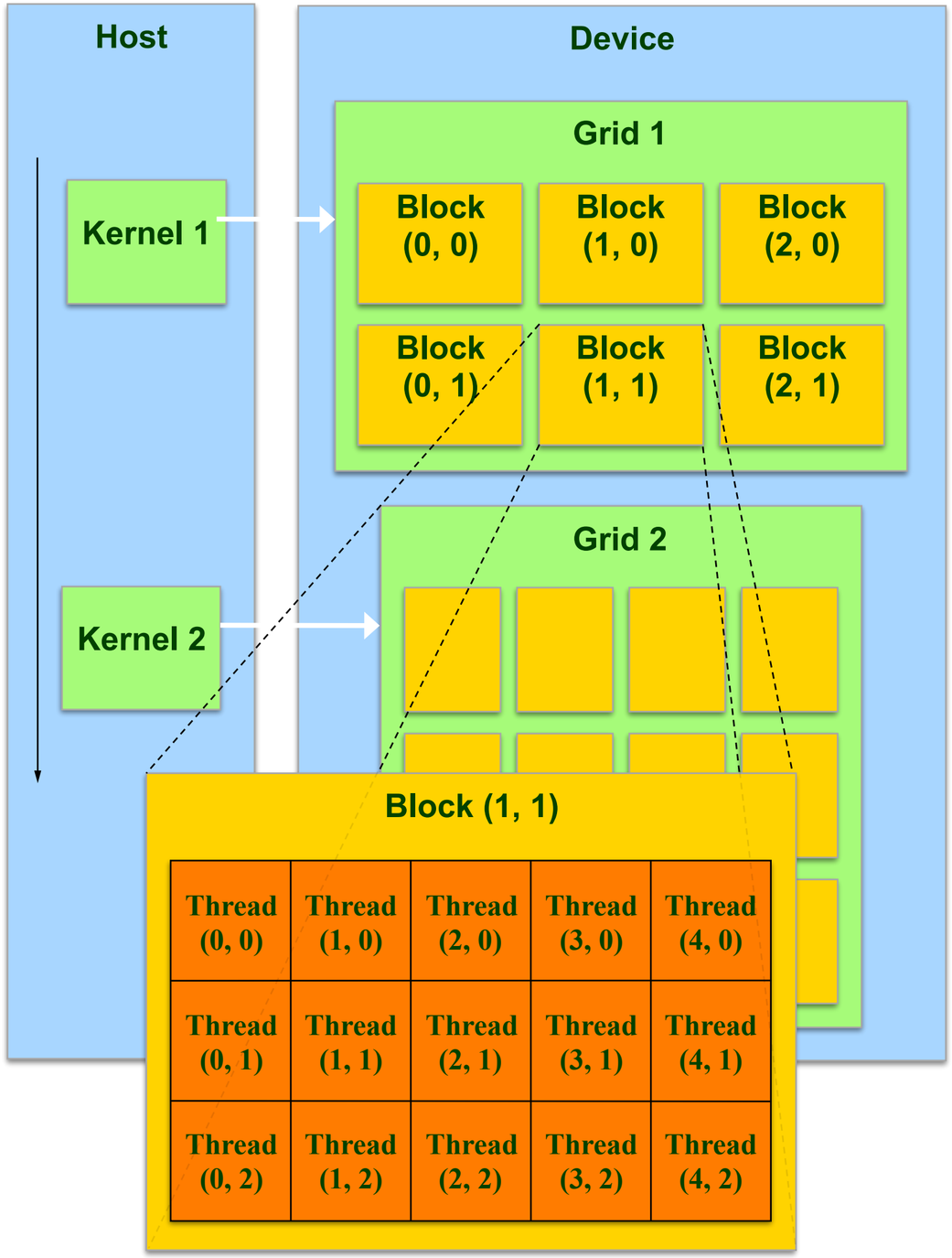
- One thread per data element.
- Doesn't this mean that large problems will have millions of threads?

Recap: Programming Model Big Idea #2

- Write one program.
- That program runs on ALL threads in parallel.
- Software terminology here is "SPMD": single-program, multiple-data.
- Hardware terminology here is "SIMT": single-instruction, multiple-thread.
  - Roughly: SIMD means many threads run in lockstep; SIMT means that some divergence is allowed and handled by the hardware

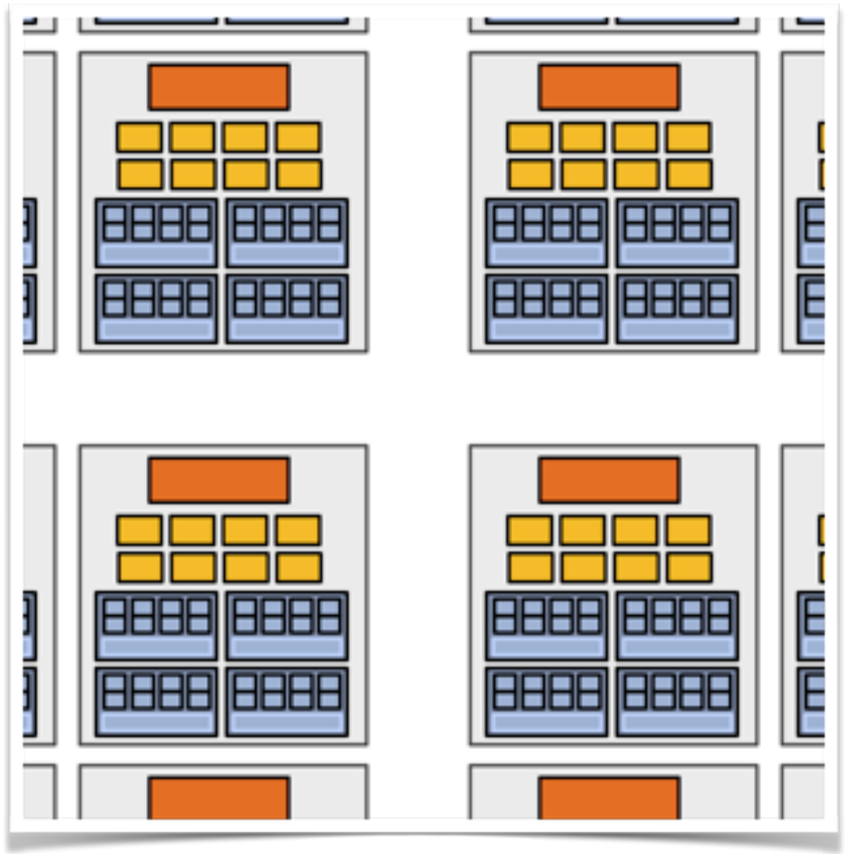
Recap: Programming Model

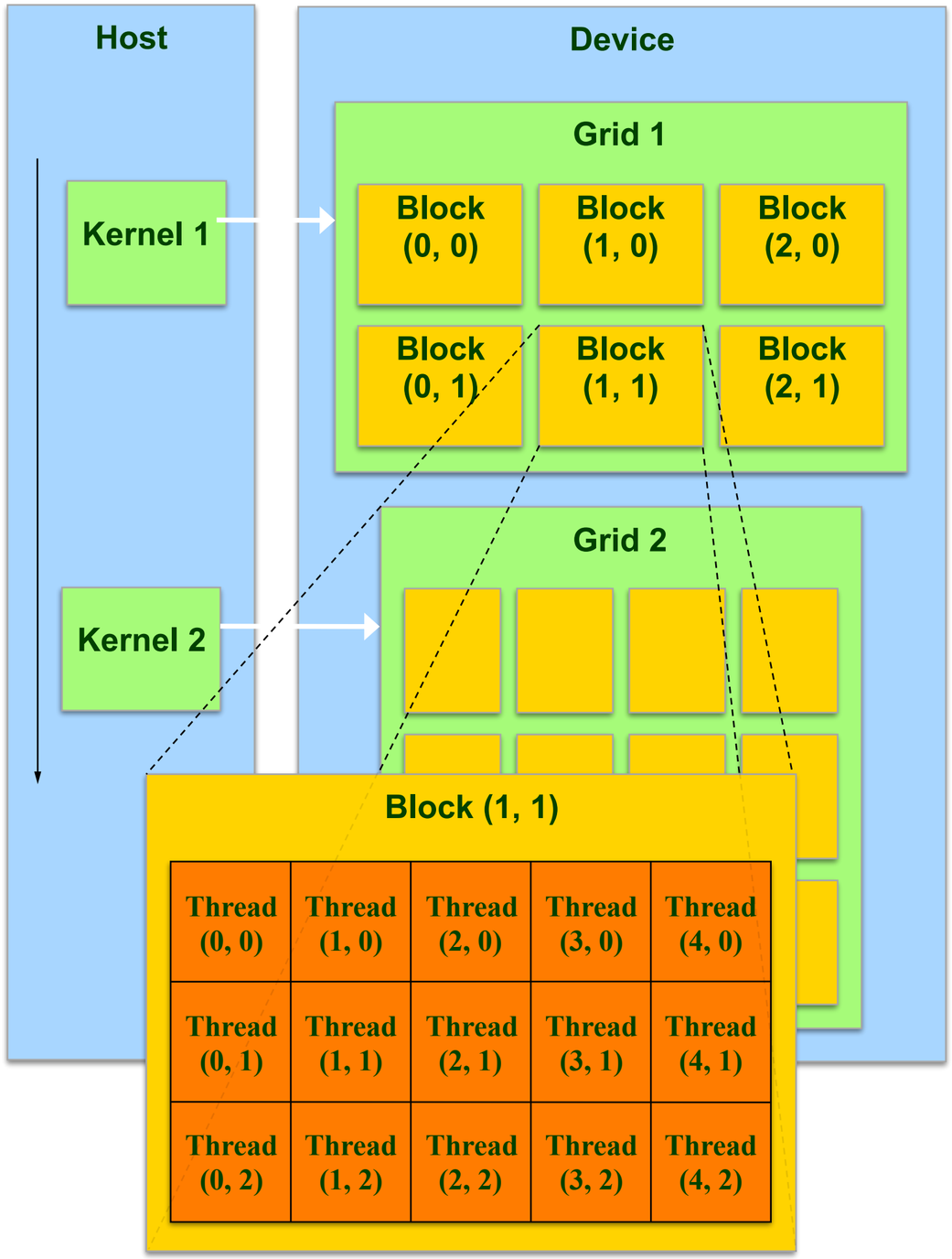
- A kernel is executed as a grid of thread blocks
- The programmer specifies the dimensions of the grid and thread block
- A thread block is a batch of threads that can cooperate with each other (shared memory, synchronization)
- Two threads from two different blocks cannot cooperate
  - Blocks are independent



Recap: What The Hardware Does

- Grids run on the entire machine
- Blocks are mapped to cores
- Threads are mapped to scalar processors





Programming Model Big Idea #3

- *Scalable execution*
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

This is very important!

- Hardware distributes independent blocks to SMs as available

Blocks must be independent

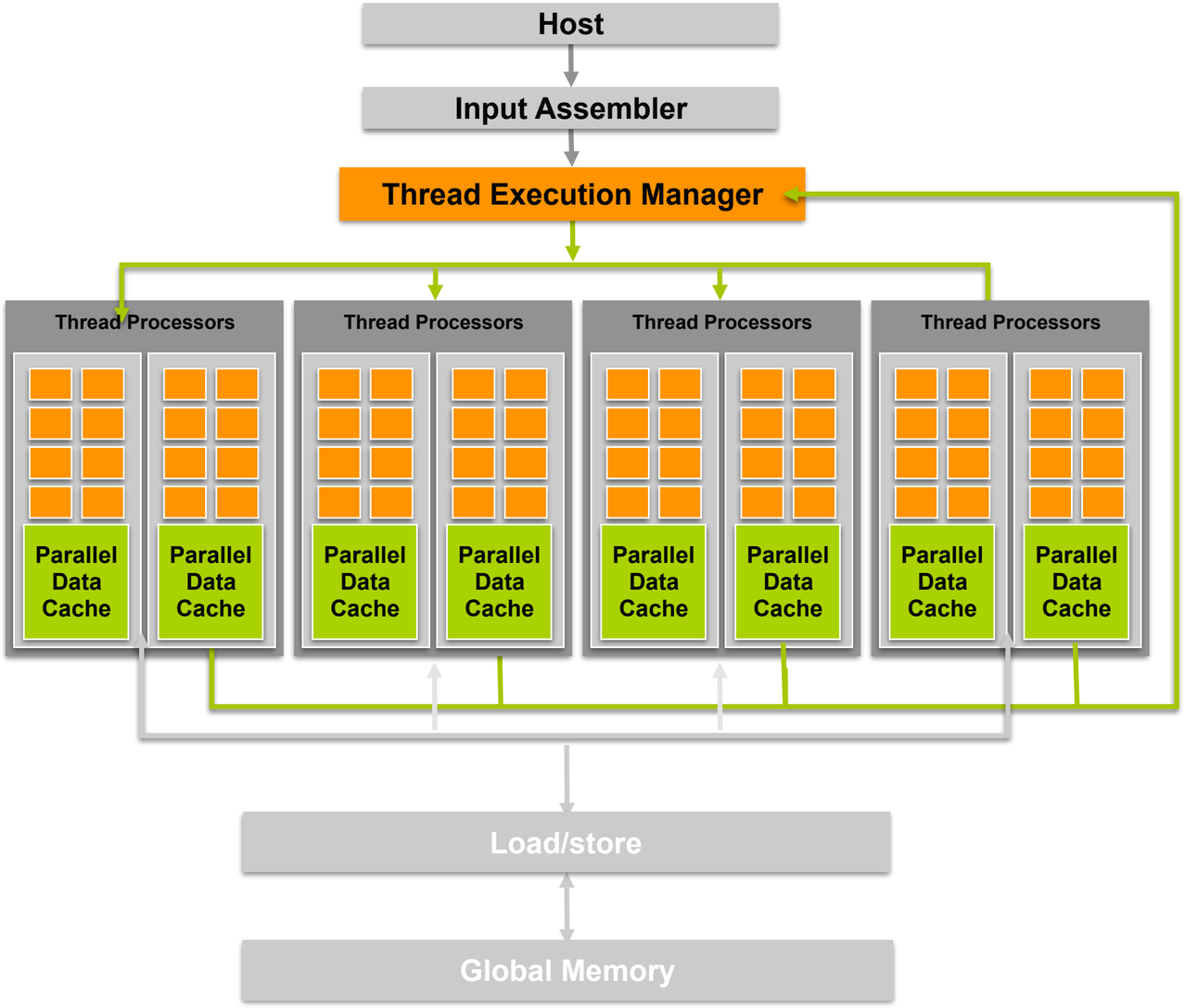
- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may *coordinate* but not *synchronize*
  - shared queue pointer: OK
  - shared lock (A locks, B unlocks): BAD … can easily deadlock
- Independence requirement gives scalability

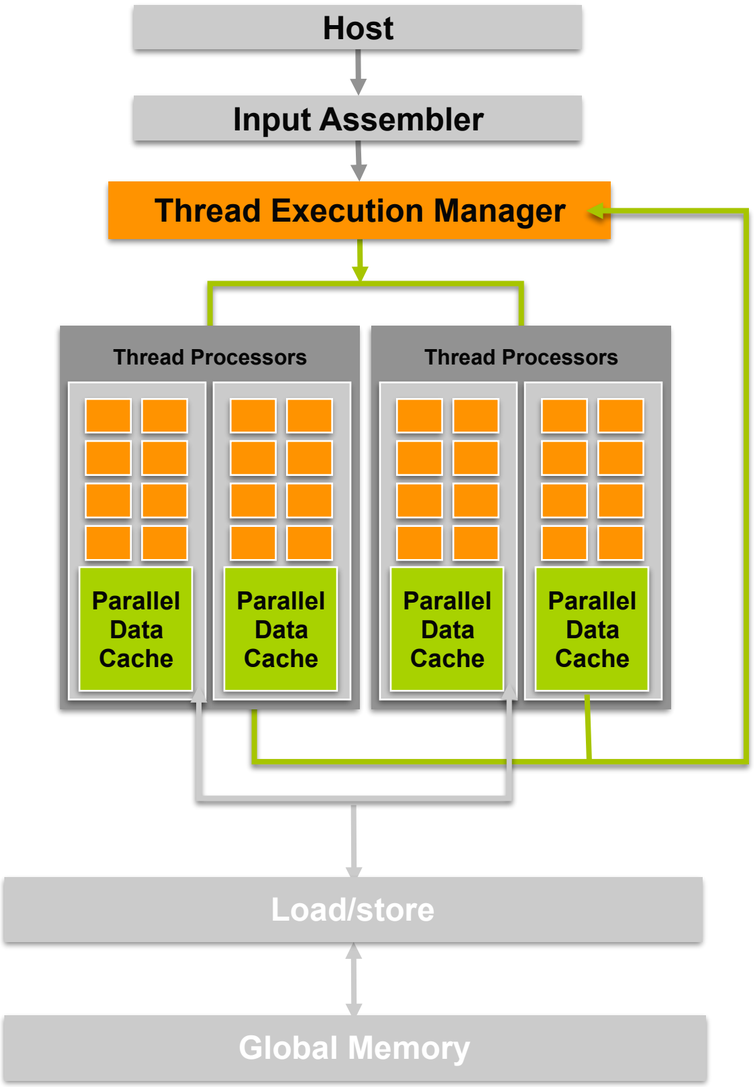
Big Idea #4

- Organization into independent blocks allows scalability / different hardware instantiations
  - If you organize your kernels to run over many blocks ...
  - ... the same code will be efficient on hardware that runs one block at once and on hardware that runs many blocks at once

Scaling the Architecture

- Same program
- Scalable performance

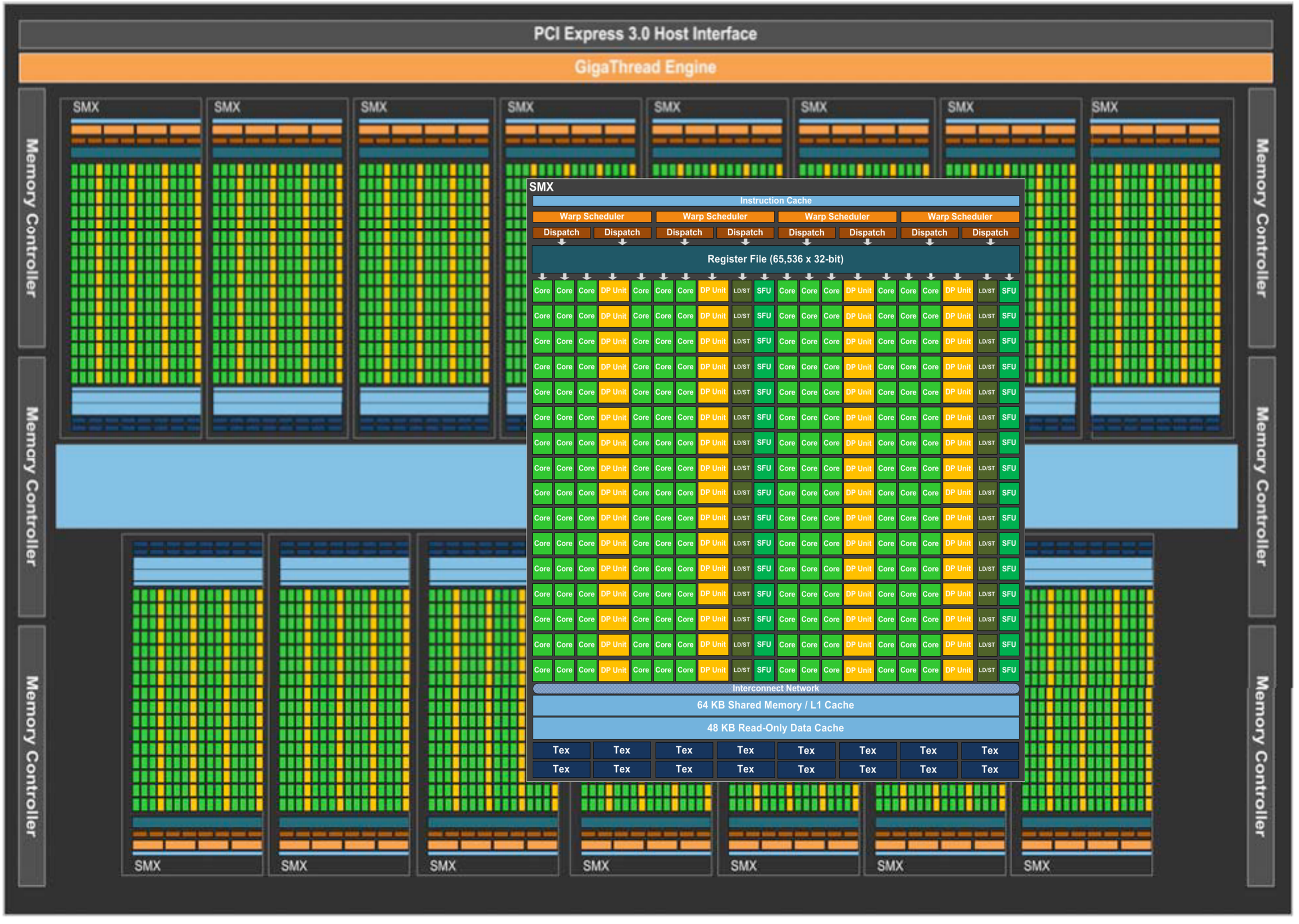




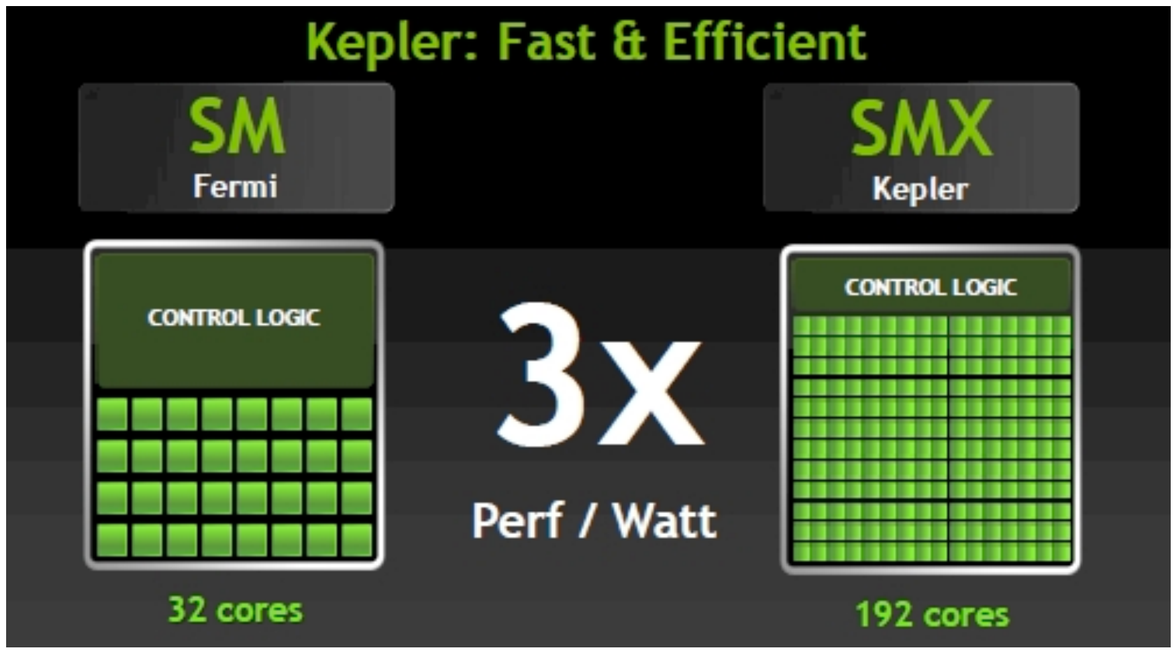
There's lots of dimensions to scale this processor with more resources. What are some of those dimensions? If you had twice as many transistors, what could you do with them?

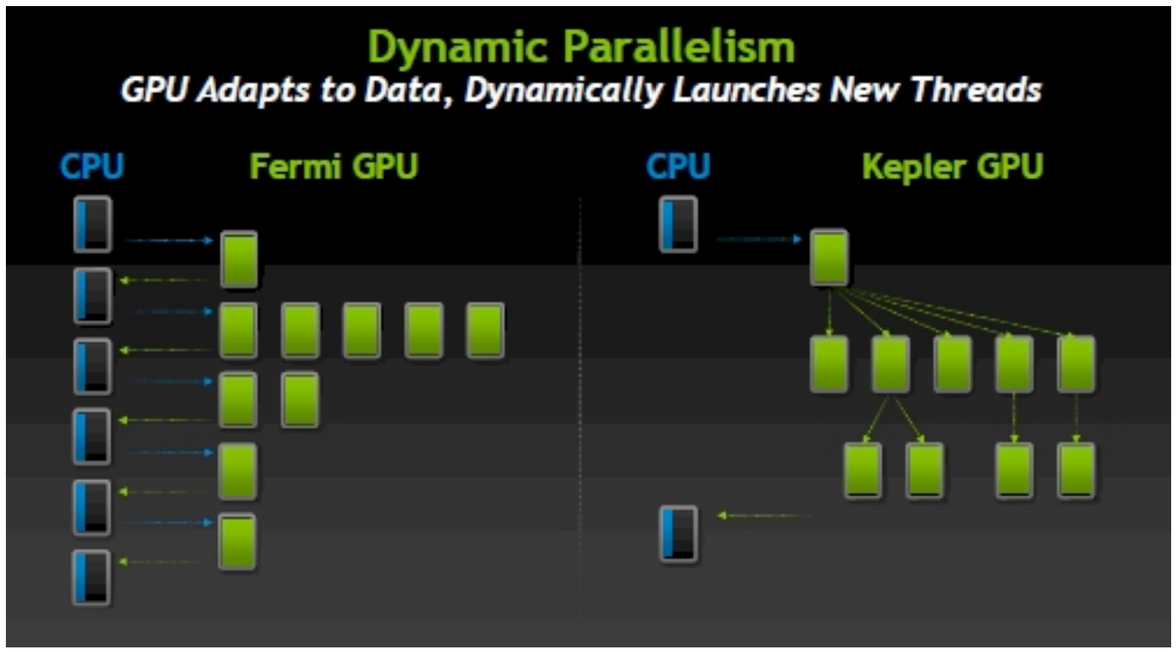
What *should* you do with more resources? In what dimension do you think NVIDIA will scale future GPUs?

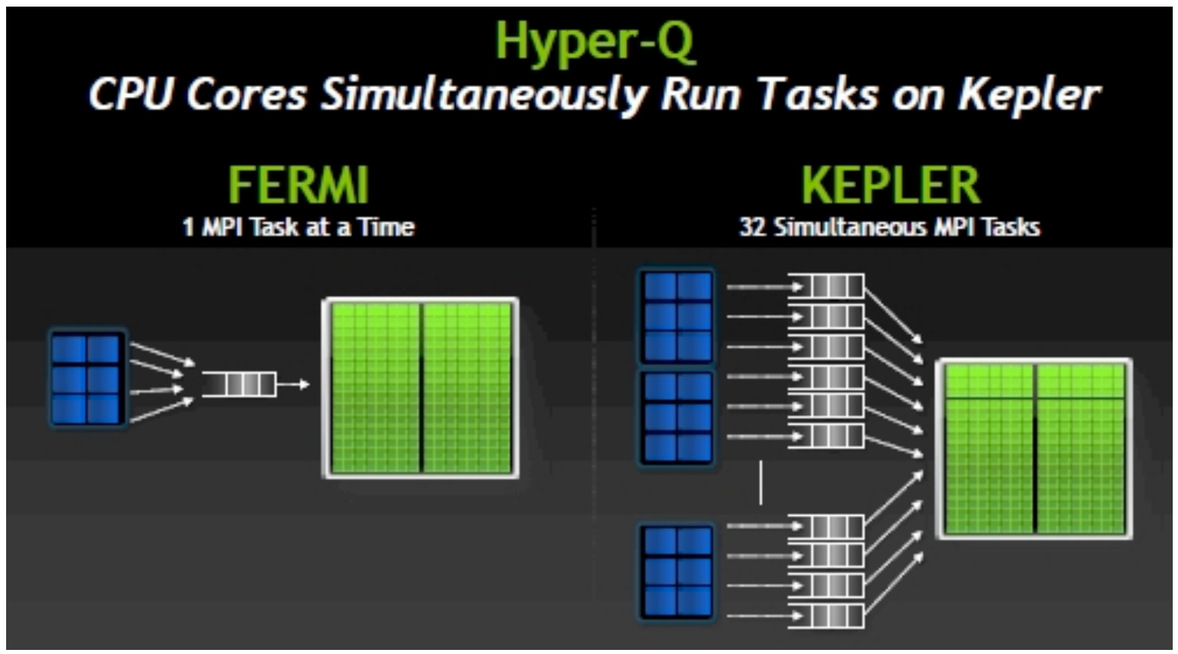
Kepler GK110 block diagram

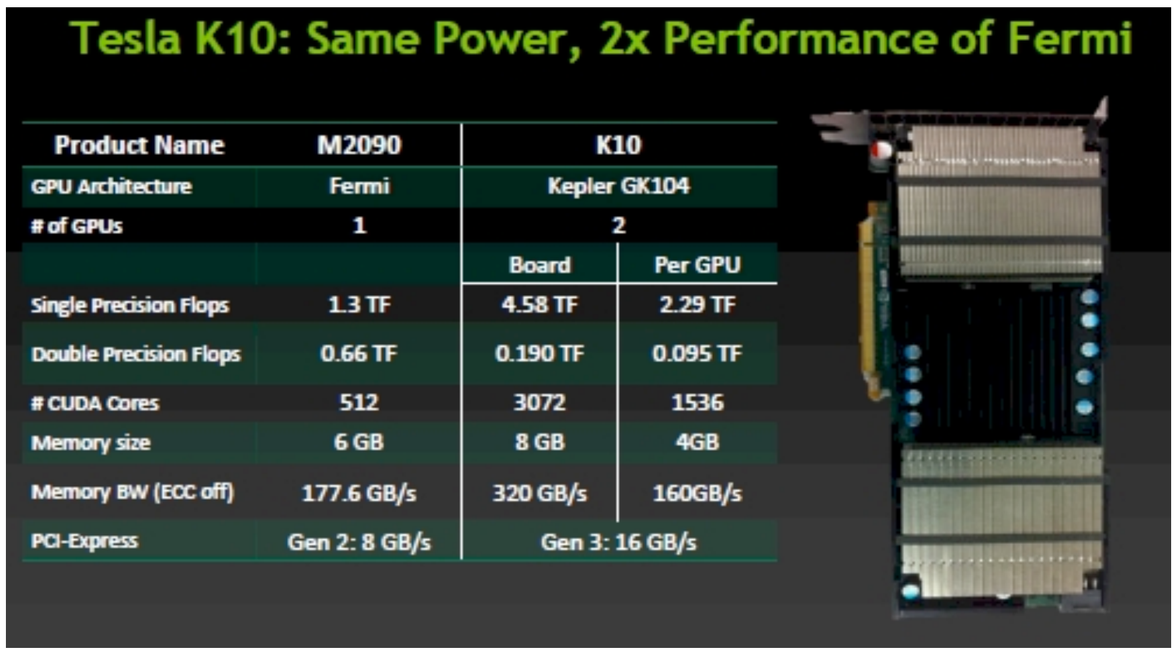


NVIDIA Kepler









Product Name	M2090	K10	
GPU Architecture	Fermi	Kepler GK104	
# of GPUs	1	2	
		Board	Per GPU
Single Precision Flops	1.3 TF	4.58 TF	2.29 TF
Double Precision Flops	0.66 TF	0.190 TF	0.095 TF
# CUDA Cores	512	3072	1536
Memory size	6 GB	8 GB	4GB
Memory BW (ECC off)	177.6 GB/s	320 GB/s	160GB/s
PCI-Express	Gen 2: 8 GB/s	Gen 3: 16 GB/s	

http://www.theregister.co.uk/2012/05/15/nvidia_kepler_tesla_gpu_revealed/page2.html

Kepler review

- Both Kepler and Fermi schedulers contain similar hardware units to handle scheduling functions, including, (a) register scoreboarding for long latency operations (texture and load), (b) inter-warp scheduling decisions (e.g., pick the best warp to go next among eligible candidates), and (c) thread block level scheduling (e.g., the GigaThread engine); however, Fermi's scheduler also contains a complex hardware stage to prevent data hazards in the math datapath itself. A multi-port register scoreboard keeps track of any registers that are not yet ready with valid data, and a dependency checker block analyzes register usage across a multitude of fully decoded warp instructions against the scoreboard, to determine which are eligible to issue.
- For Kepler, we realized that since this information is deterministic (the math pipeline latencies are not variable), it is possible for the compiler to determine up front when instructions will be ready to issue, and provide this information in the instruction itself. This allowed us to replace several complex and power-expensive blocks with a simple hardware block that extracts the pre-determined latency information and uses it to mask out warps from eligibility at the inter-warp scheduler stage.
- The short story here is that, in Kepler, the constant tug-of-war between control logic and FLOPS has moved decidedly in the direction of more on-chip FLOPS. The big question we have is whether Nvidia's compiler can truly be effective at keeping the GPU's execution units busy. Then again, it doesn't have to be perfect, since Kepler's increases in peak throughput are sufficient to overcome some loss of utilization efficiency. Also, as you'll soon see, this setup obviously works pretty well for graphics, a well-known and embarrassingly parallel workload. We are more dubious about this arrangement's potential for GPU computing, where throughput for a given workload could be highly dependent on compiler tuning. That's really another story for another chip on another day, though, as we'll explain shortly.

http://techreport.com/articles.x/22653

CUDA Software Development Kit

CUDA Optimized Libraries:
math.h, FFT, BLAS, …

Integrated CPU + GPU
C Source Code

NVIDIA C Compiler (LLVM-based)

NVIDIA Assembly
for Computing (PTX)

CPU Host Code

CUDA
Driver

Debugger
Profiler

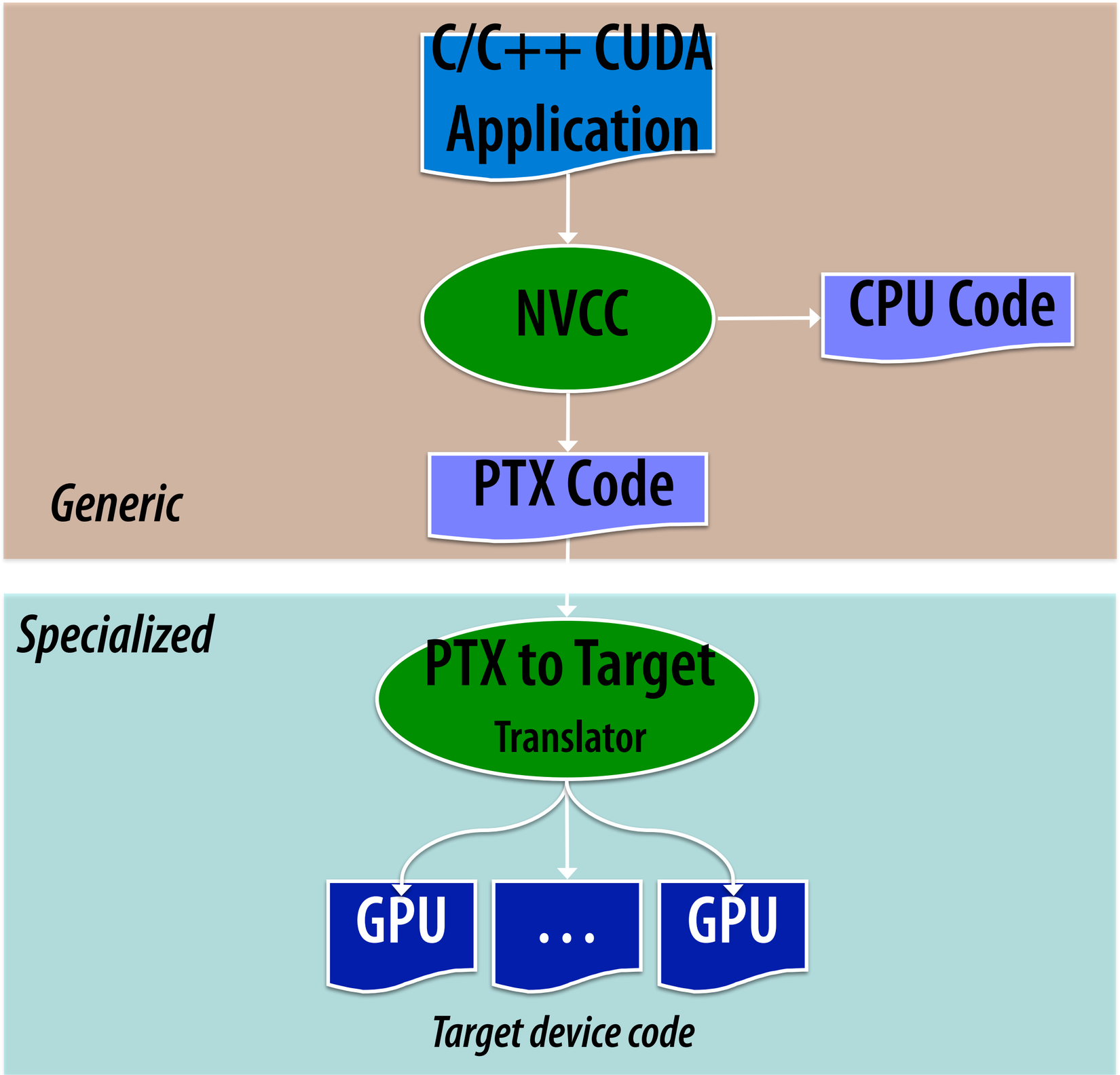
Standard C Compiler

SASS (GPU machine code)

GPU

CPU

Compiling CUDA for GPUs



OpenCL vs. CUDA

- OpenCL: Controlled by Khronos Group (consortium)
- Supported on:
  - GPUs (AMD, Intel, NVIDIA, Apple, Qualcomm, Samsung, etc.)
  - CPUs (Intel, AMD, etc.)
  - DSPs, FPGAs
- Very similar programming model (SPMD kernels, grid/blocks/threads, etc.)
- CUDA has single-program model. OpenCL separates programs into host vs. kernel code.
- Fewer {language, functional} features than CUDA.
- OpenCL programs have more boilerplate (harder for beginners)
- Generally *functionally* compatible across devices, but not necessarily *performance*-compatible

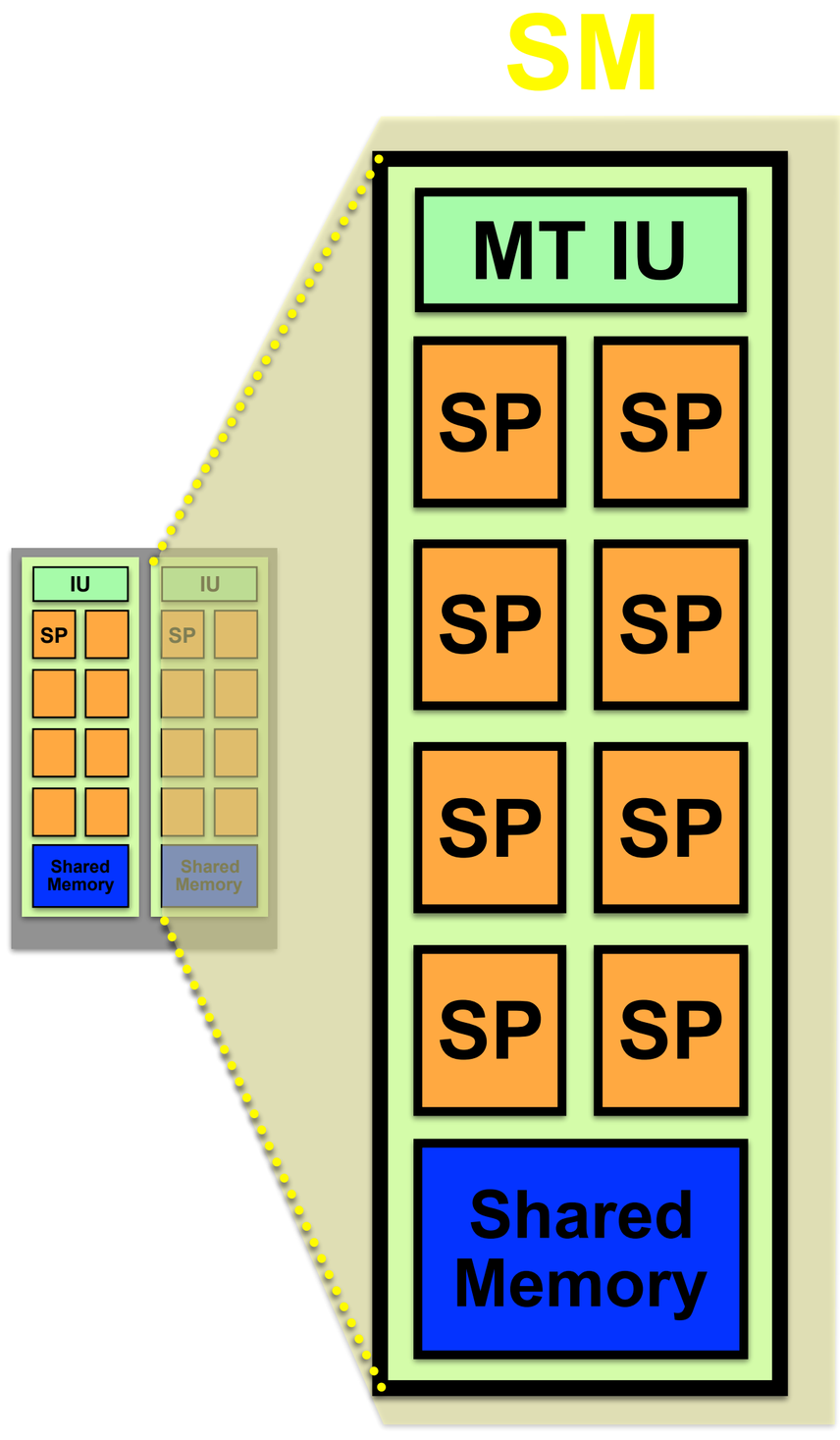
CUDA vs. OpenCL terminology

CUDA term	OpenCL term
GPU	Device
(Streaming) Multiprocessor	Compute unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or local)	Private memory
Block	Work-group
Thread	Work-item

[Matt Harvey, "Experiences porting from CUDA to OpenCL"]

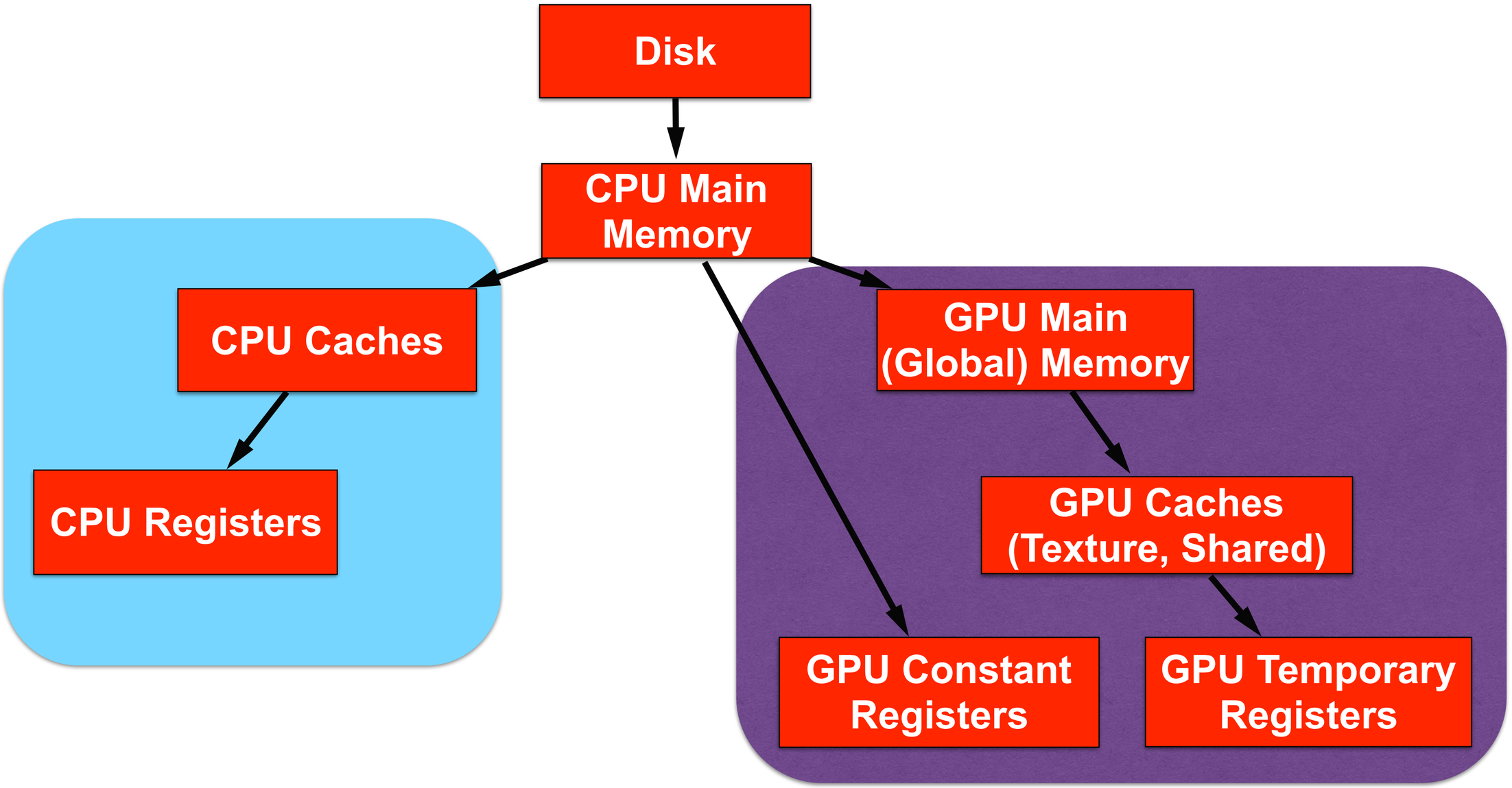
SM Multithreaded Multiprocessor

- Each SM runs a *block* of threads
- SMs have 8, 16, or 32 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hw multithreaded (1024 in newer hw)
- 16KB Shared Memory (64KB in newer hw)
  - Concurrent threads share data
  - Low latency load/store
- 32 elements run at same time (SIMD) as a *warp*



What do you think goes in a single thread processor? How would you design it?

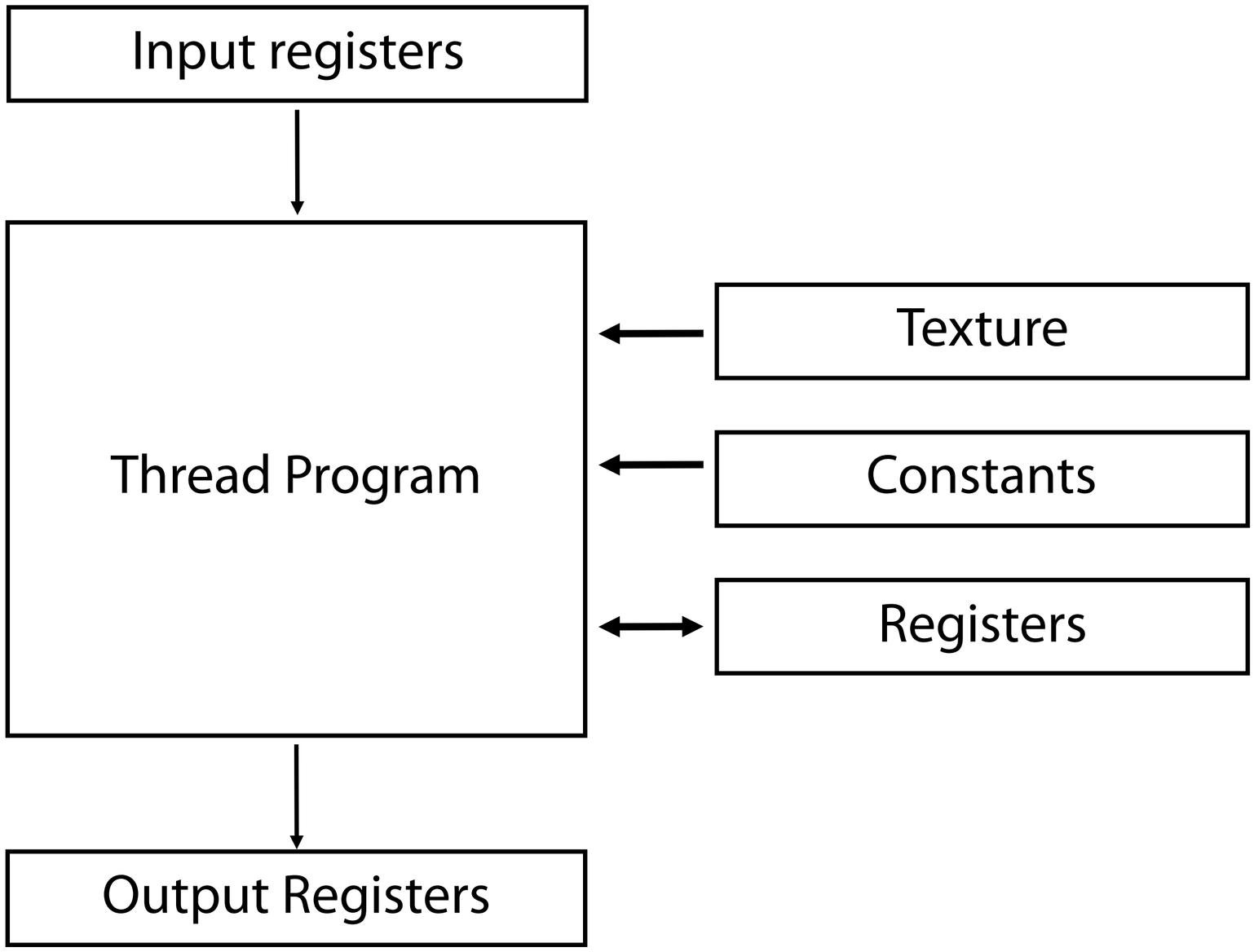
CPU/GPU Memory Hierarchy



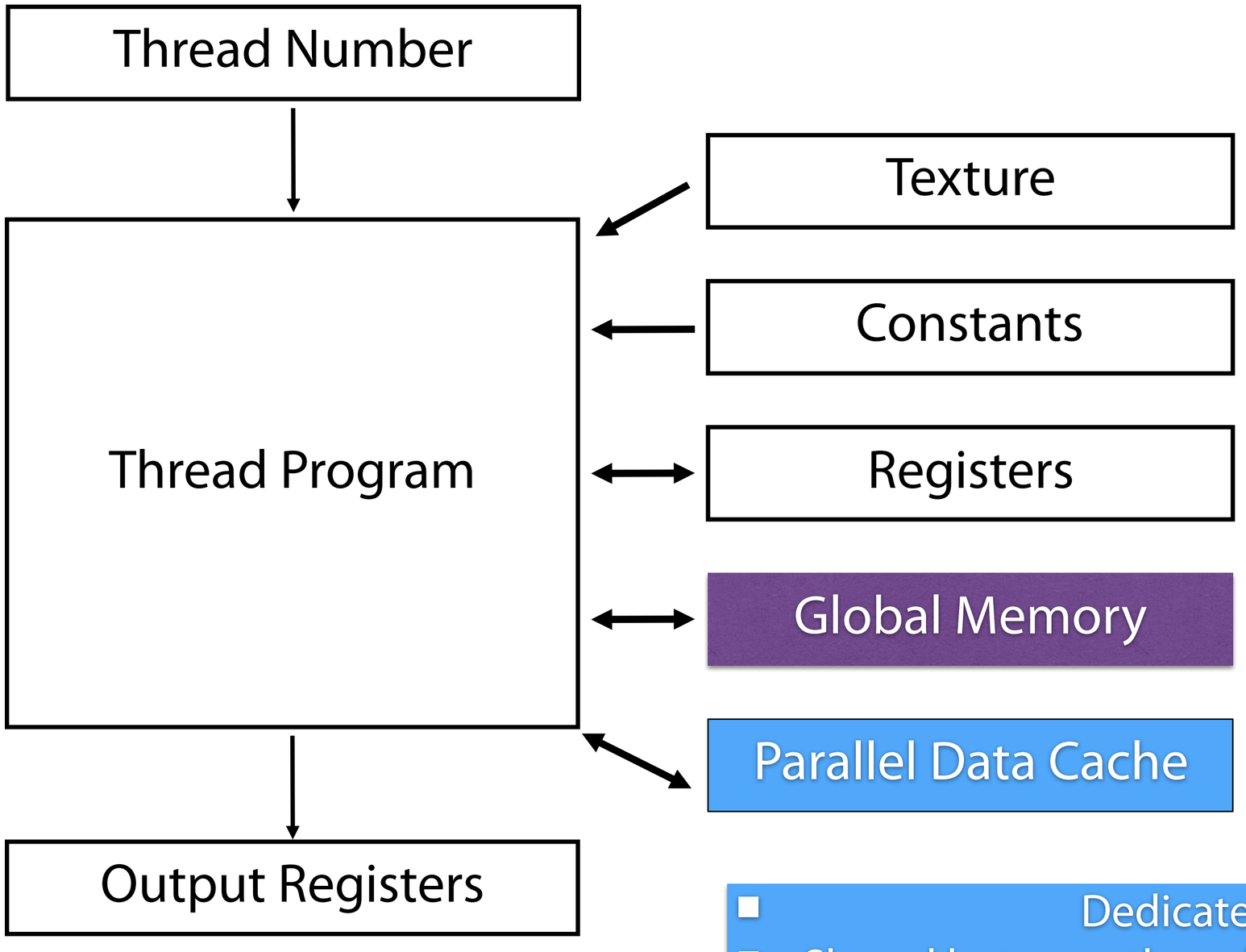
What about hybrid (CPU/GPU same die) processors?

Graphics Programs

- **Features**
  - **Millions of instructions**
  - **Full integer and bit instructions**
  - **No limits on branching, looping**



GPU Memory Spaces

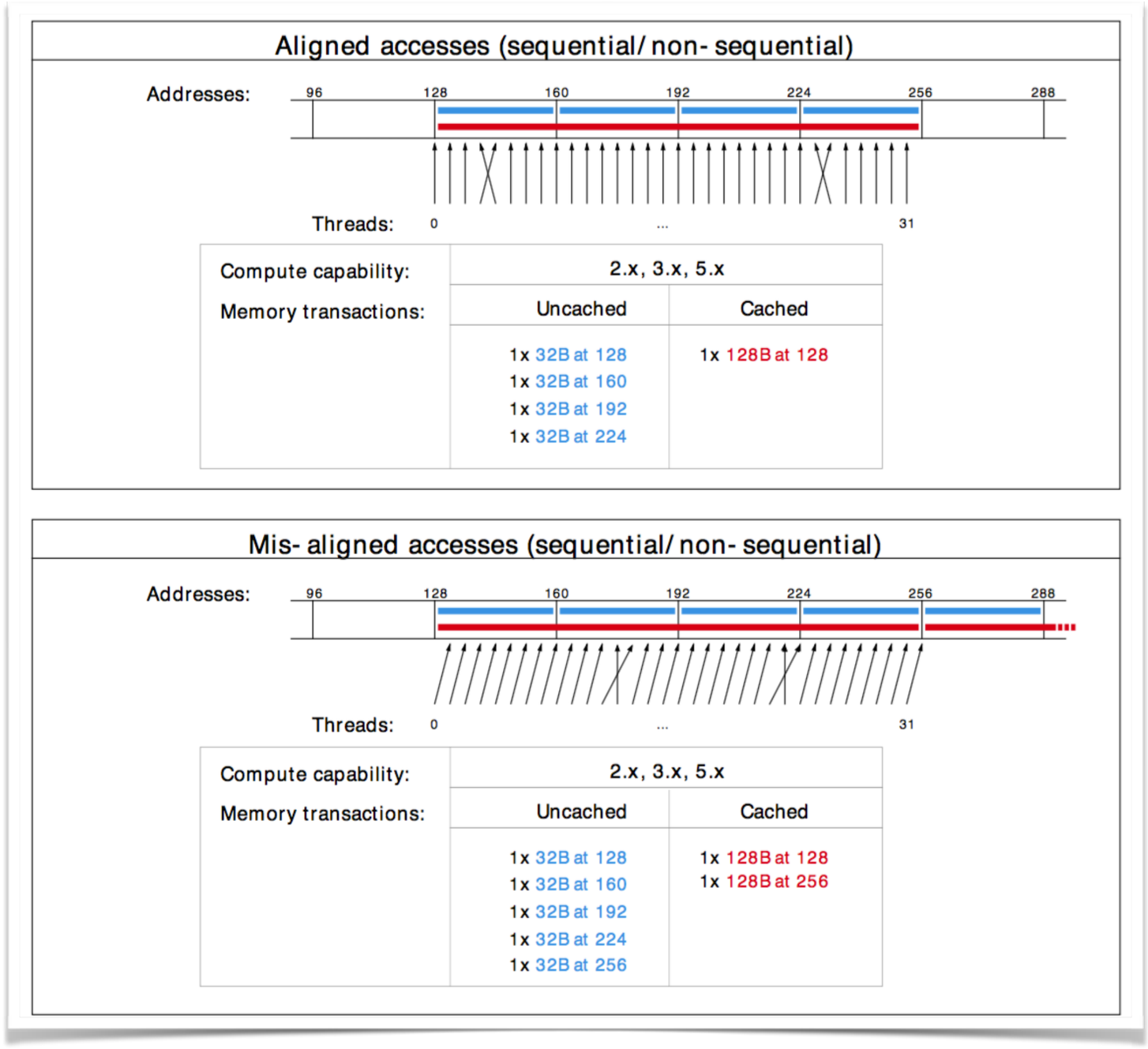


- 1D, 2D, or 3D thread ID allocation
- Fully general load/store to GPU memory: Scatter/Gather
- Programmer flexibility on how memory is accessed
- Untyped, not limited to fixed texture types
- Pointer support

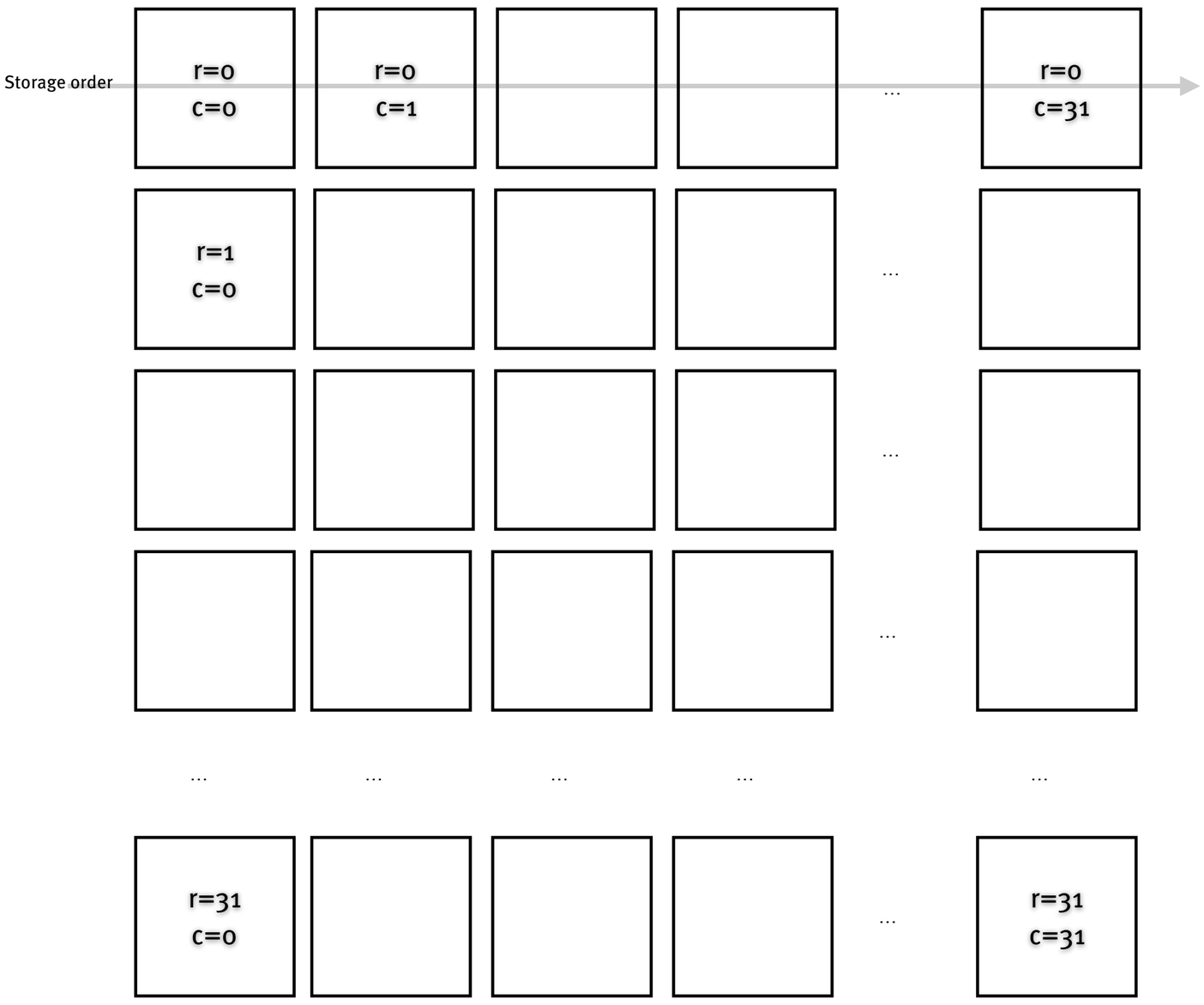
- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed
- (Much) closer to register speed than memory speed

Memory Coalescing in Global Memory

- **Your mental model should be that:**
  - **Once you touch anything in a memory chunk, every other item in that chunk is free**
  - **Your memory cost is the cost of all chunks touched**



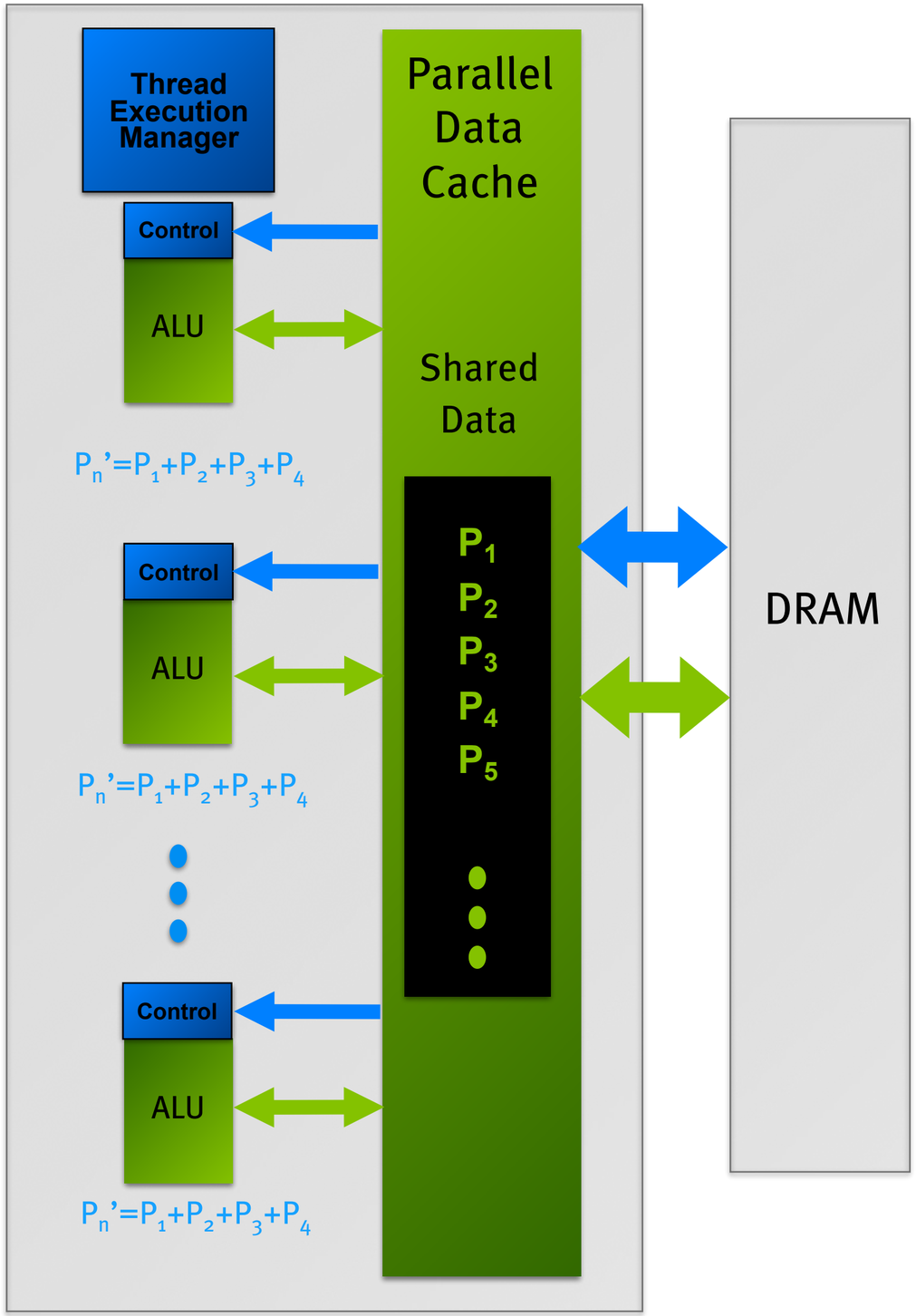
Memory Coalescing Example



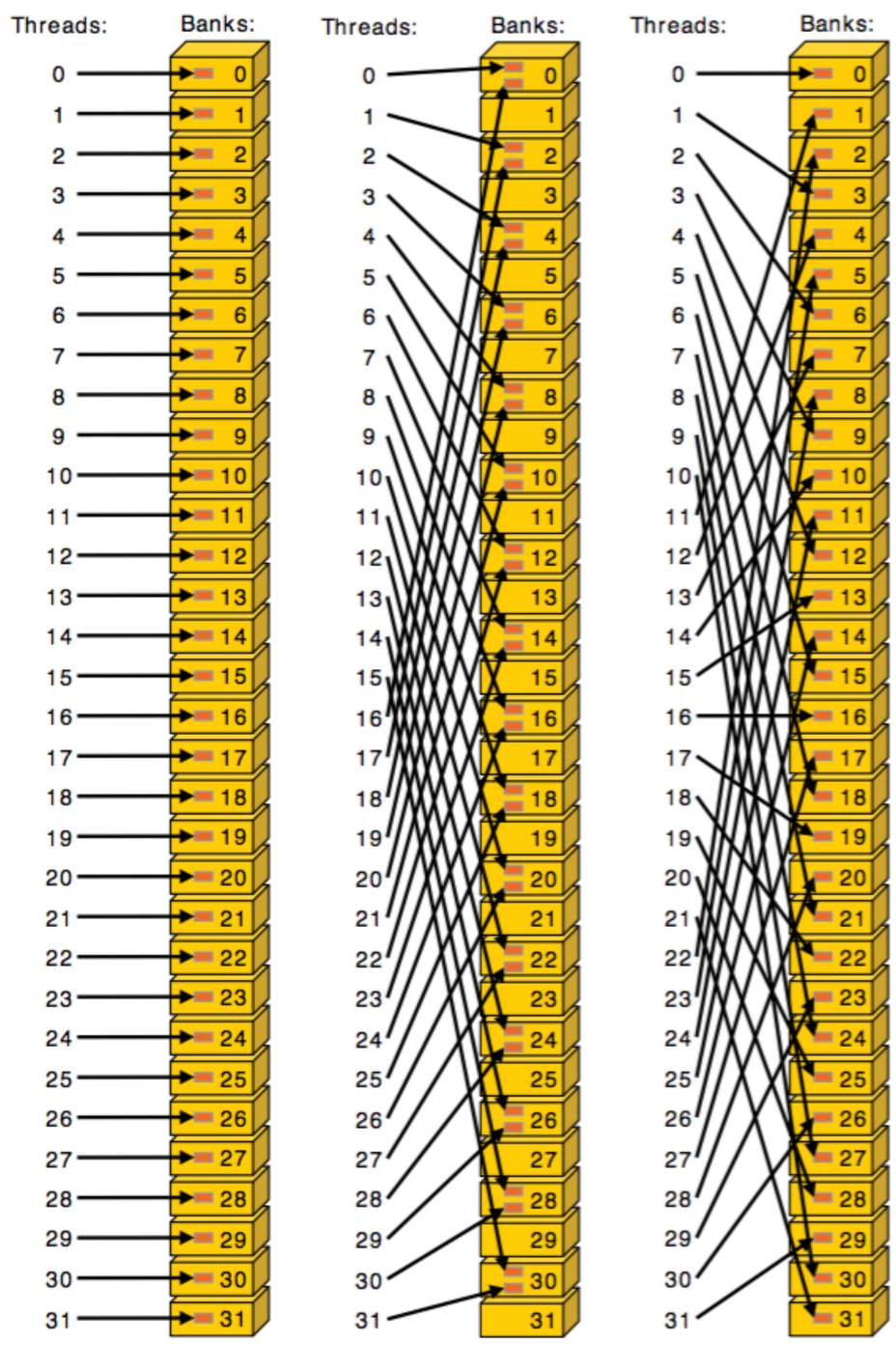
- Read a 32x32 block of memory, stored in row-major order, from DRAM
- Read it row-major: thread i reads (r=i/32, c=i%32):
  - Fully coalesced
  - 32 memory transactions
- Read it column-major: thread i reads (r=i%32, c=i/32):
  - Fully uncoalesced
  - 32x32 memory transactions

Parallel Data Cache

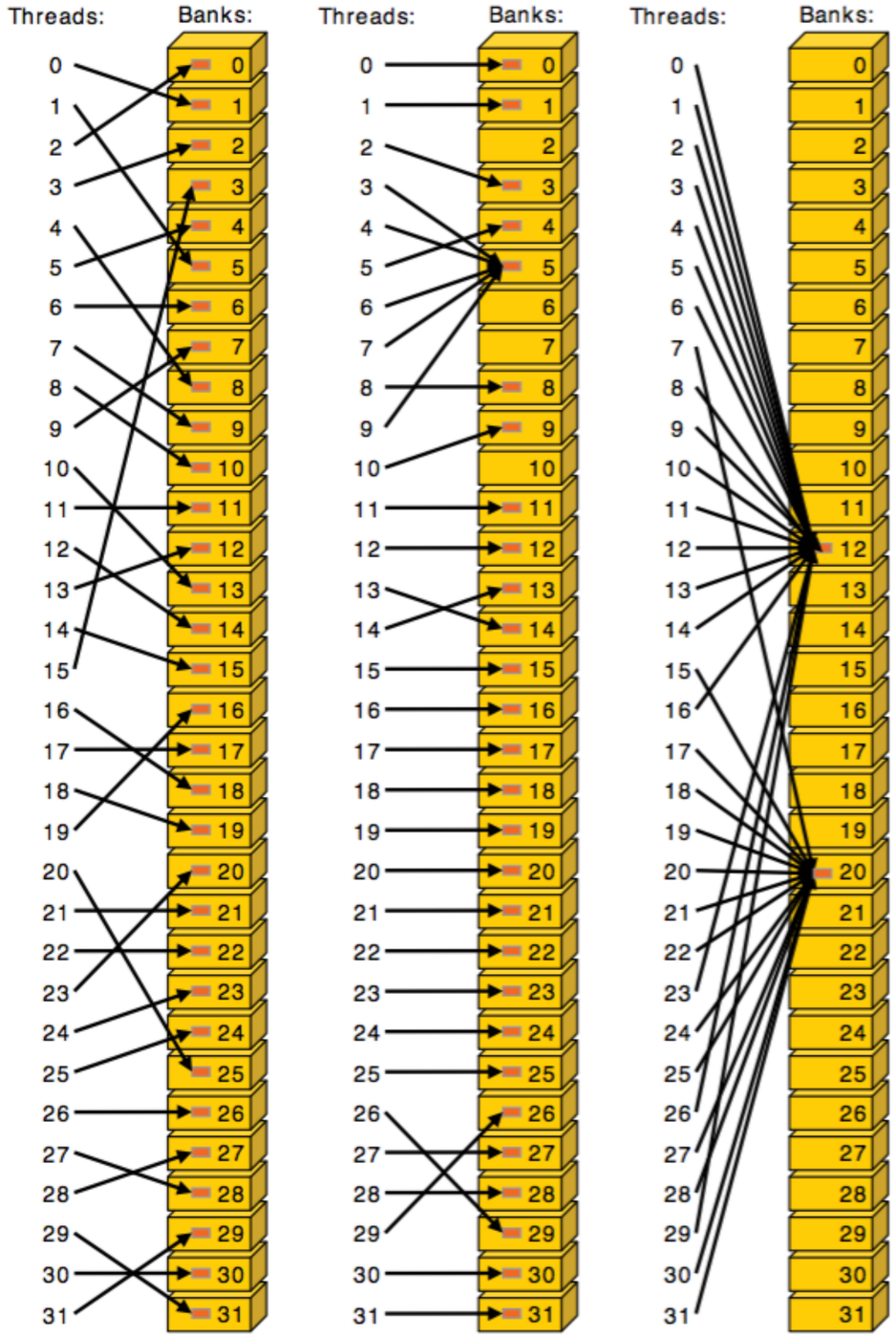
- Addresses a fundamental problem of stream computing
- Bring the data closer to the ALU
- Stage computation for the parallel data cache
- Minimize trips to external memory
- Share values to minimize overfetch and computation
- Increases arithmetic intensity by keeping data close to the processors
- User managed generic memory, threads read/write arbitrarily



Shared memory has banks



Left
Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle
Linear addressing with a stride of two 32-bit words (two-way bank conflict).
Right
Linear addressing with a stride of three 32-bit words (no bank conflict).



Left
Conflict-free access via random permutation.
Middle
Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right
Conflict-free broadcast access (threads access the same word within a bank).

This parallel data cache seems pretty cool. Since we can have tens of thousands of threads active at any time, why can't all of them communicate through it?

What did we learn about the hardware?

- Base unit of the hardware is a SIMD (lockstep) warp of 32 threads
  - Avoid branching within a warp
- GPU "SMs" (cores) run blocks of threads
  - SMs run warps when they are ready.
  - Warps may be from the same or different blocks.
  - The hardware schedules blocks onto SMs if sufficient SM resources.
- Memory hierarchy, {small, fast} -> {big, slow}: registers, shared memory + L1, L2, global memory (DRAM)

Hardware design points

- {More, weaker} cores vs. {fewer, more powerful} cores (SMs)
- {More, weaker} thread processors vs. {fewer, more powerful}
- Registers per thread
- Managing branch divergence & warp width
- Caches, and user-managed vs. hardware-managed
- Allowable communication mechanisms between {threads, warps, blocks, GPUs}
- Atomics/synchronization and where they can be used

Execution Model

- Kernels are launched in grids
  - One kernel executes at a time
- A block executes on one multiprocessor
  - Does not migrate, runs to completion
- Several blocks can reside concurrently on one multiprocessor (SM)
  - Control limitations (~current GPUs, compute version 5.0):
    - At most 32 concurrent blocks per SM
    - At most 2048 concurrent threads per SM
  - Number is further limited by SM resources
    - Register file (64–128k) is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks

What is a thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be physical threads
  - as on NVIDIA GPUs
- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU
  - Interesting research on this topic

What is a thread block?

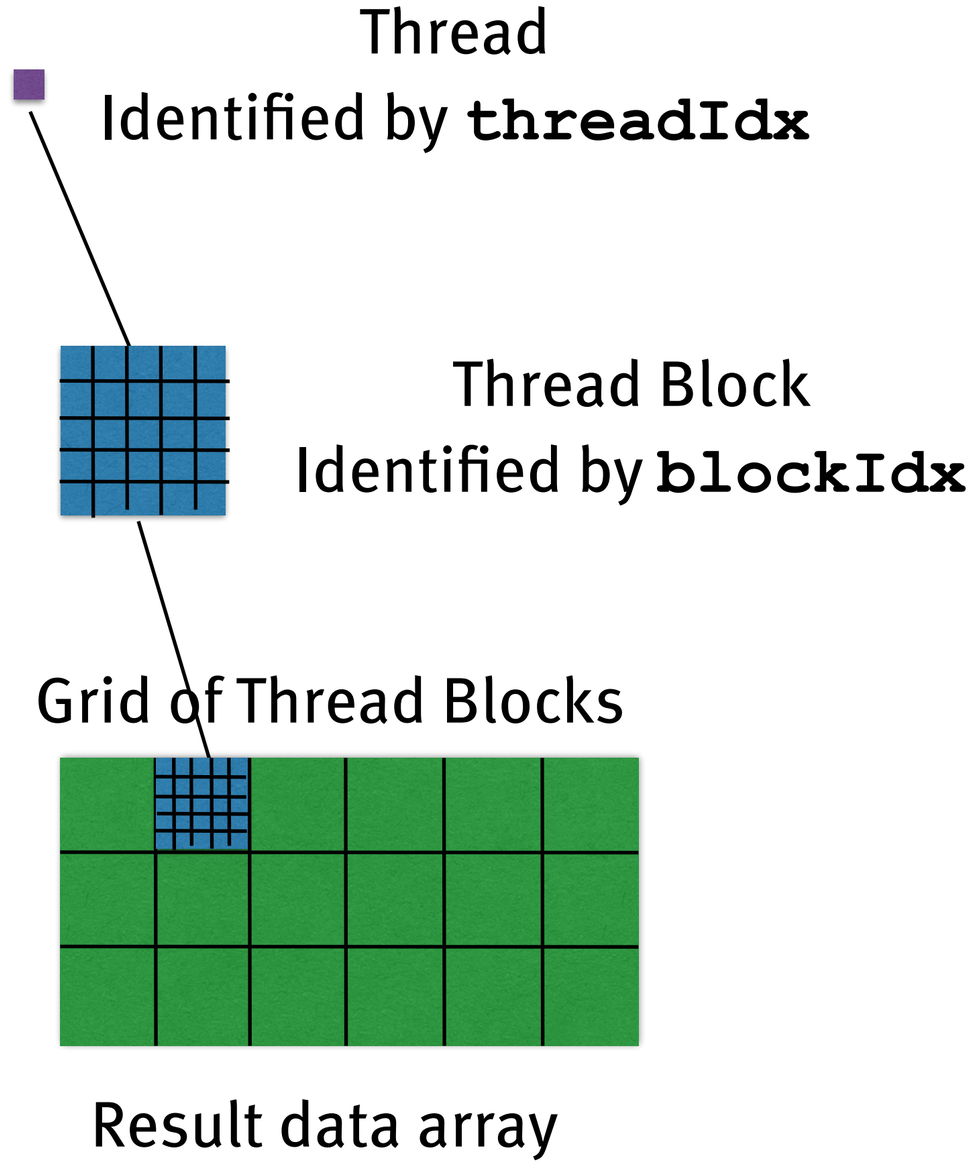
- Thread block = virtualized multiprocessor
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be independent tasks
  - program valid for any interleaving of block executions

Levels of parallelism

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

Some CUDA high-level concepts

- Multiple levels of parallelism
- Thread block
  - Up to 2048 threads per block
  - Communicate through shared memory
  - Threads guaranteed to be resident
    - `threadIdx, blockIdx`
    - `__syncthreads()`
- Grid of thread blocks
    - `f<<<nblocks, nthreads>>>(a,b,c)`



Key Parallel Abstractions in CUDA

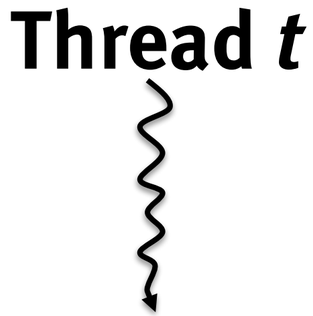
- Hierarchy of concurrent threads

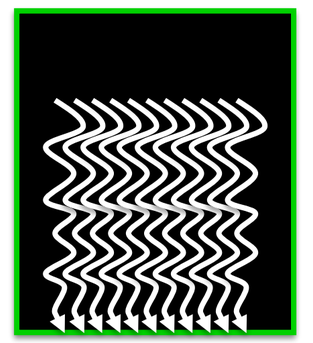
- Lightweight synchronization primitives

- Shared memory model for cooperating threads

Hierarchy of concurrent threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program
  - (This is "SPMD")
- Threads are grouped into thread blocks
  - threads in the same block can cooperate
- Threads/blocks have unique IDs
  - Each thread knows its "address" (thread/block ID)





CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);   // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

CUDA: Features available on GPU

- Standard mathematical functions
  - `sinf, powf, atanf, ceil, min, sqrtf, etc.`
- Atomic memory operations
  - `atomicAdd, atomicMin, atomicAnd, atomicCAS,` etc.
- Texture accesses in kernels
  - `// declare texture reference`
    `texture<float,2> my_texture;`
  - `float4 texel = texfetch(my_texture, u, v);`

Example: Vector Addition Kernel

- Compute vector sum C = A+B means:
- n = length(C)
-
```
for i = 0 to n-1:
    C[i] = A[i] + B[i]
```
- So C[0] = A[0] + B[0], C[1] = A[1] + B[1], etc.

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

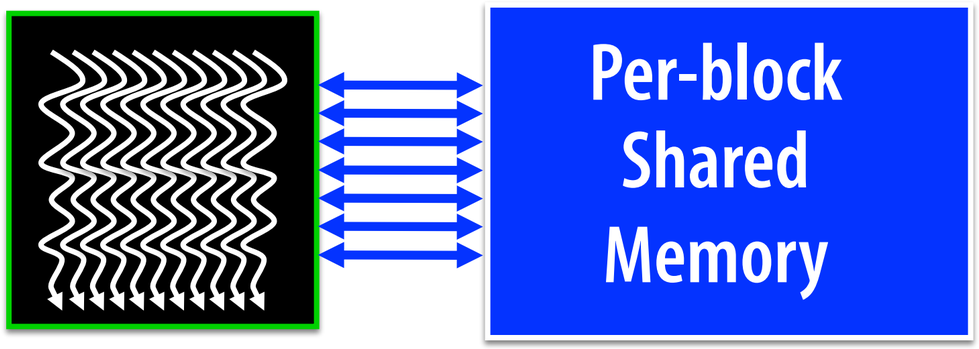
Host Code

Memory model

Thread

This is either
– Registers (up to 255/thread)
– Spill to DRAM (“local memory”)

Block



SM 5.0 GPUs:
48 kB/thread block

Synchronization of blocks

- Threads within block may synchronize with barriers
- Why is this necessary? Your mental model needs to be:
  - Any warp can run at any time
  - Warps can and will run until they hit a barrier
- Consider 128 threads in a block where you want thread t's register *y* = thread 127-t's register *x* (reversal):

```
shared[t] = register_x
register_y = shared[127-t]
```

Synchronization of blocks

- Threads *within* block may synchronize with barriers
  - … Step 1 …

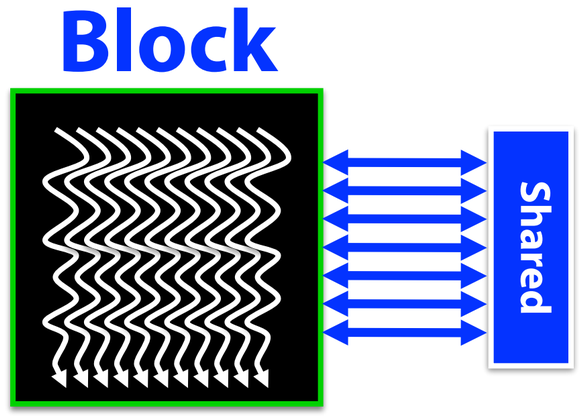
    `__syncthreads();`

    … Step 2 …
- Blocks *coordinate* via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`
  - but don't *cooperate* between blocks!
    - e.g., block 1 spins waiting for block 0 to complete
- Implicit barrier between dependent kernels
  - `vec_minus<<<nblocks, blksize>>>(a, b, c);`
    `vec_dot<<<nblocks, blksize>>>(c, c);`

Using per-block shared memory

- Variables shared across block

```
__shared__ int *begin, *end;
```



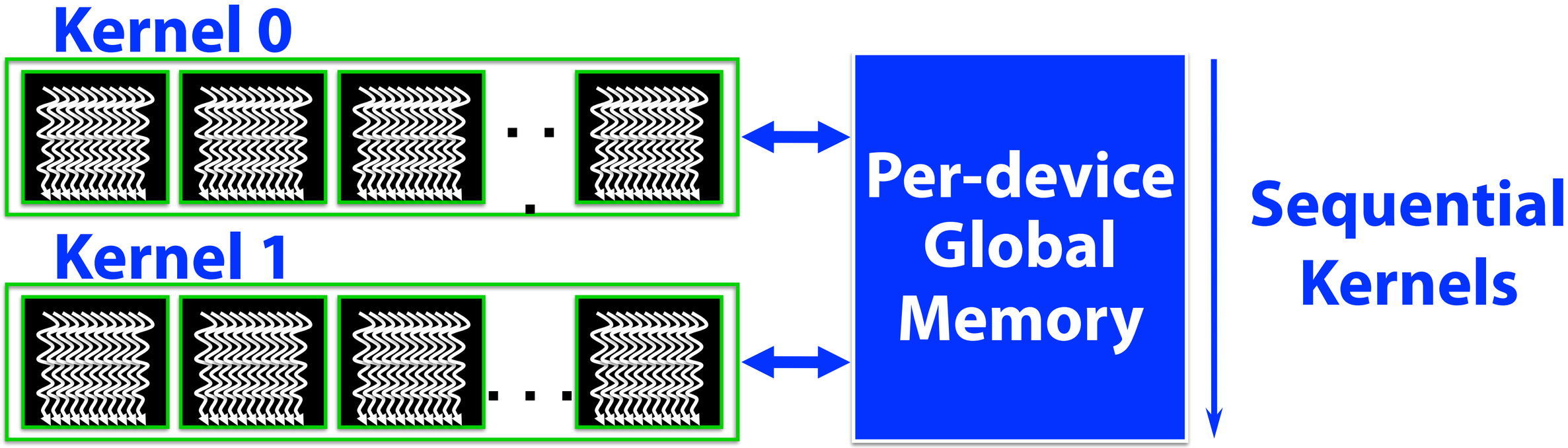
- Scratchpad memory

```
__shared__ int scratch[blocksize];

scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

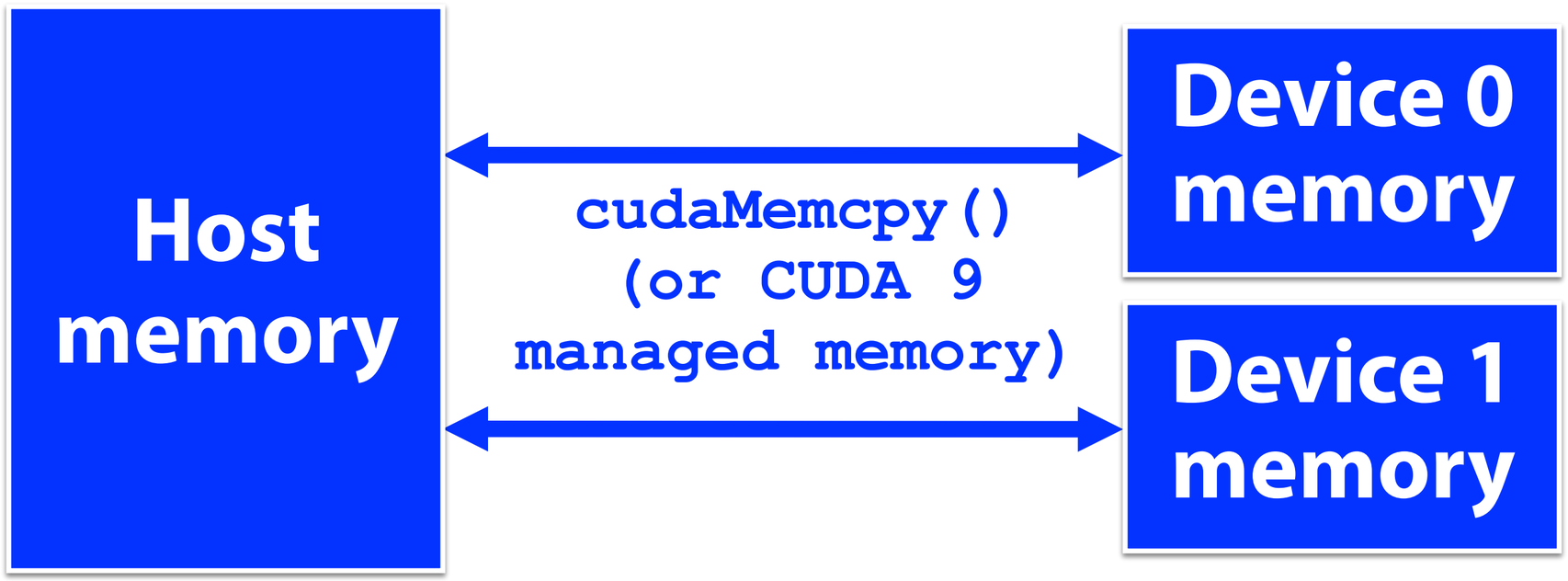
- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```

Memory model



Memory model



CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory

```
cudaMalloc(), cudaMallocManaged(),
cudaFree()
```

- Explicit memory copy for host ↔ device, device ↔ device, device ↔ host

```
cudaMemcpy(),cudaMemcpy2D(),...
```

- Texture management

```
cudaBindTexture(),cudaBindTextureToArray(),...
```

- OpenGL & DirectX interoperability

```
cudaGLMapBufferObject(),
cudaD3D9MapVertexBuffer(),...
```

Example: Vector Addition Kernel

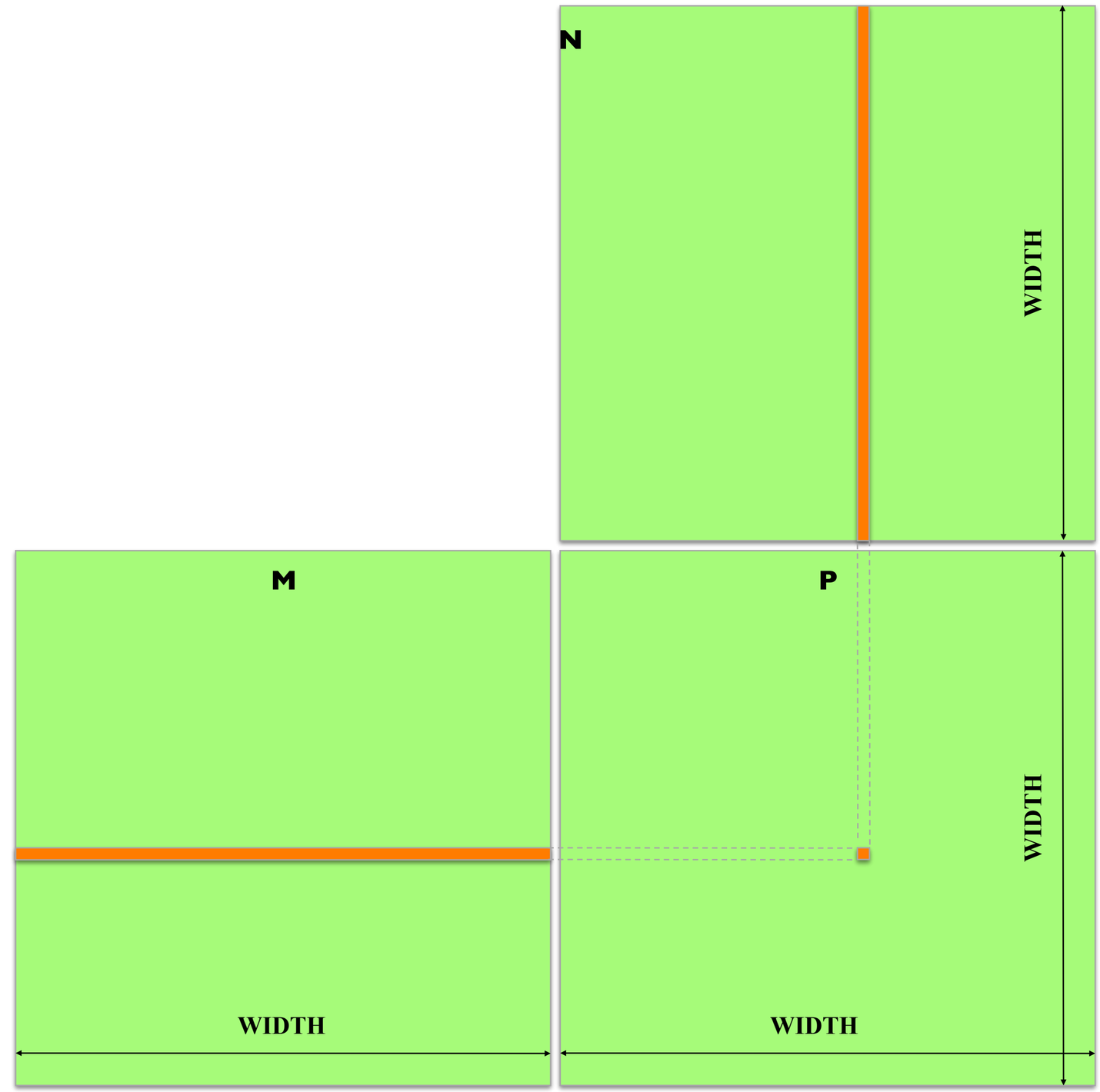
```
// Compute vector sum C = A+B

// Each thread performs one pair-wise addition

__global__ void vecAdd(float* A, float* B, float* C){

    int i = threadIdx.x + blockDim.x * blockIdx.x;

    C[i] = A[i] + B[i];

}


int main(){

    // Run N/256 blocks of 256 threads each

    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);

}
```

Example: Host code for `vecAdd`

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

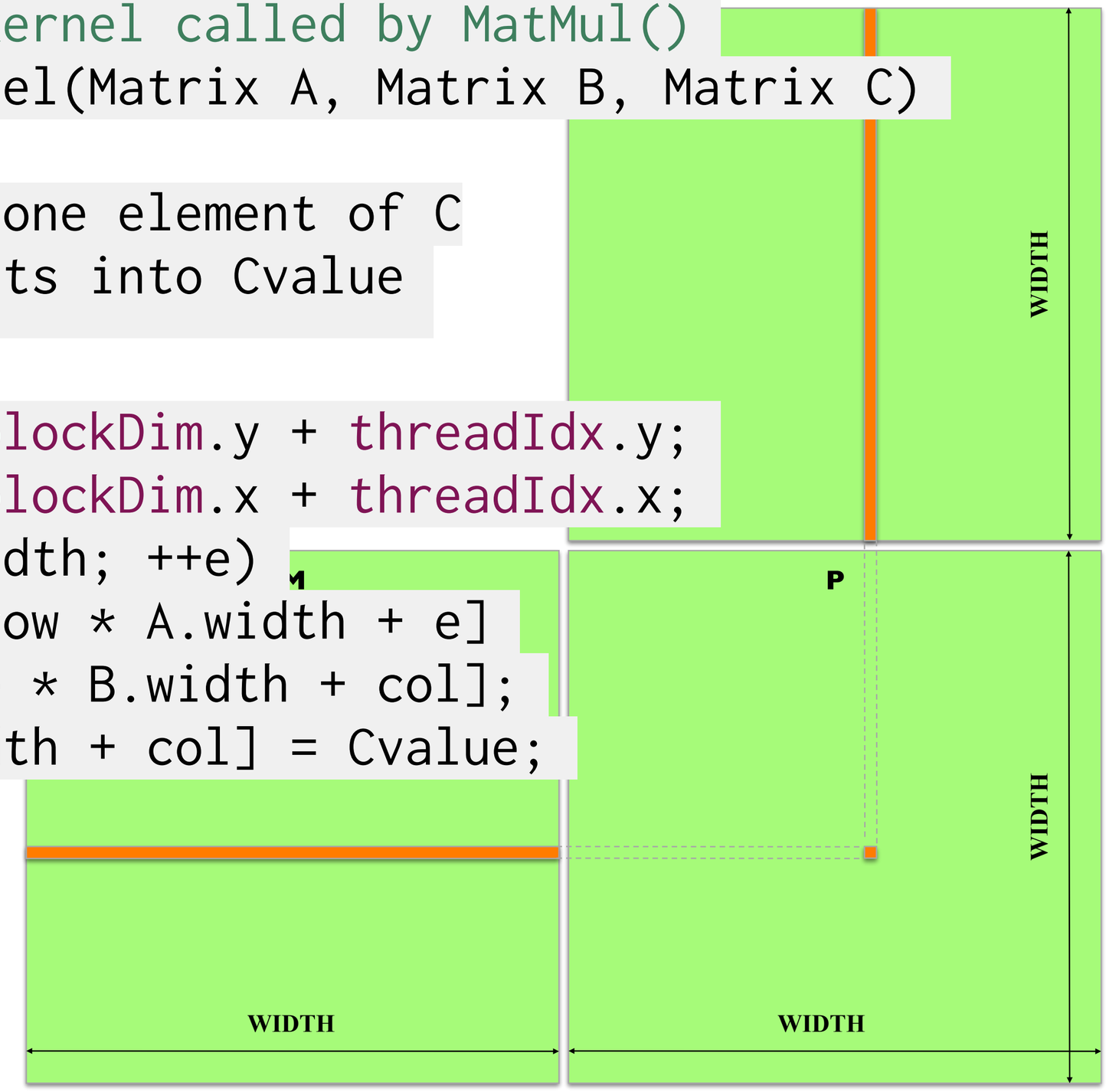
Dense Matrix Multiplication

- for all elements E in destination matrix P
  - $P_{r,c} = M_r \bullet N_c$



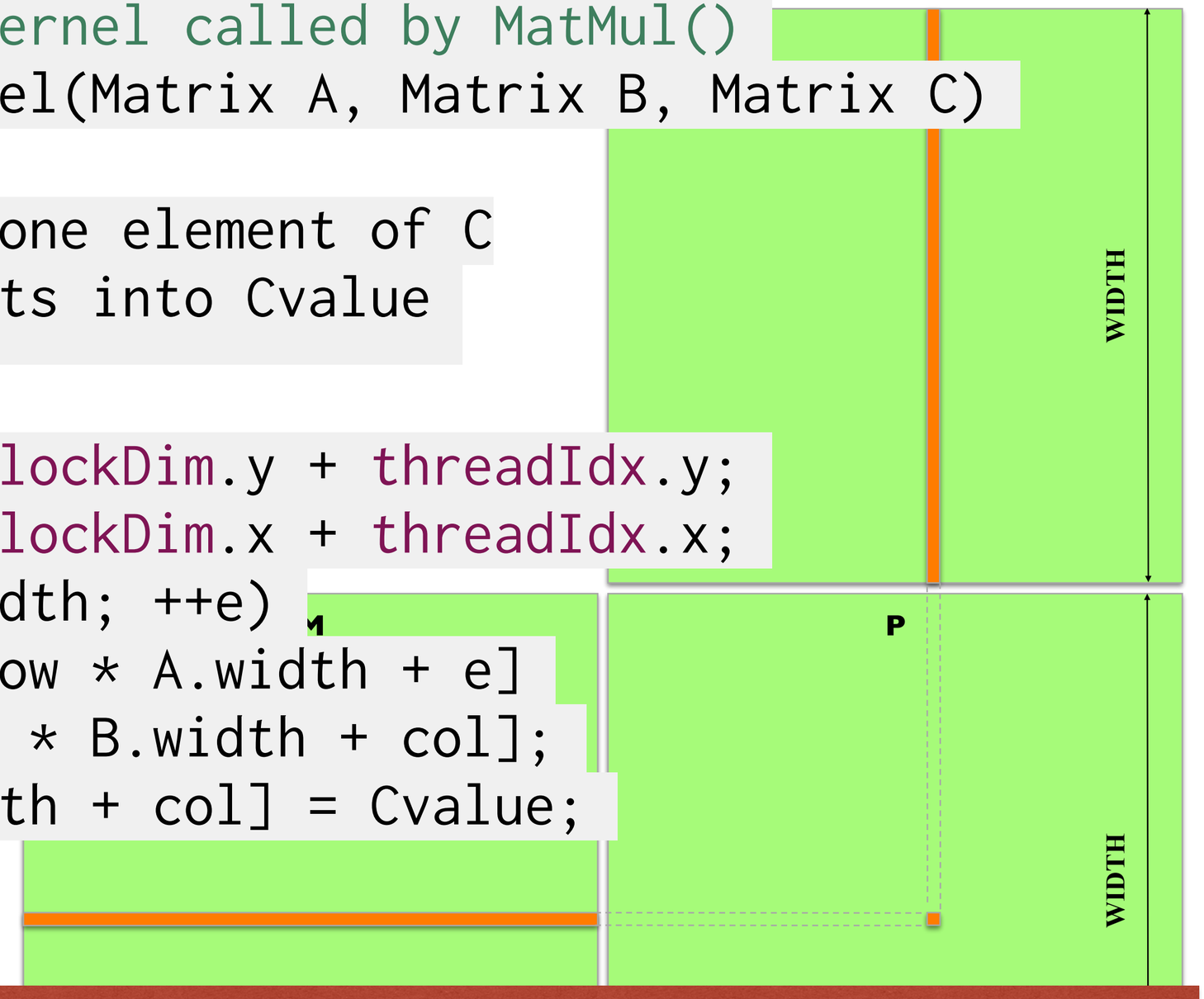
Dense Matrix Multiplication

```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue

  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
            * B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Dense Matrix Multiplication

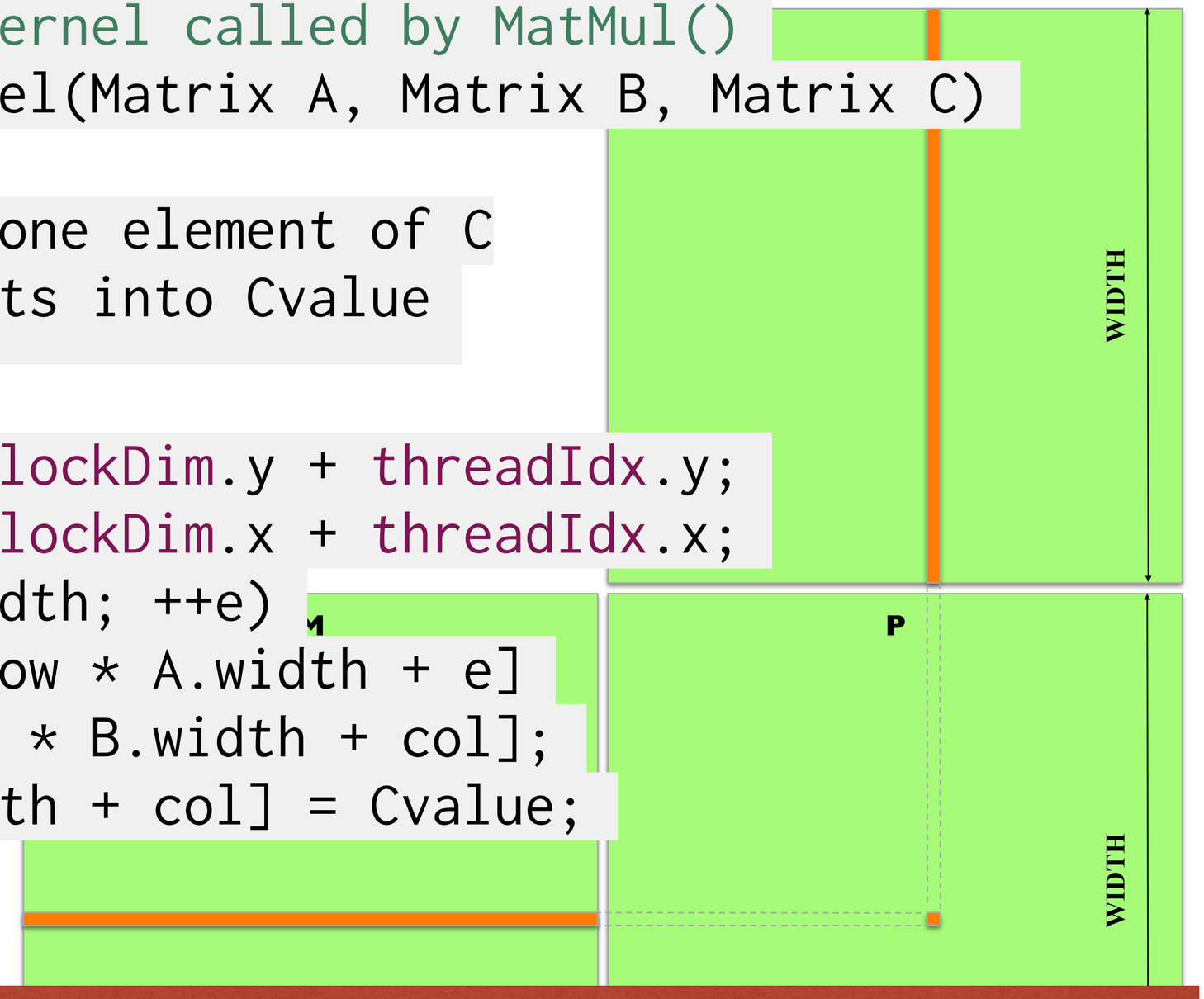
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue

  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
            * B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Q1: Discounting addressing arithmetic, how many multiplications are we going to do?

Dense Matrix Multiplication

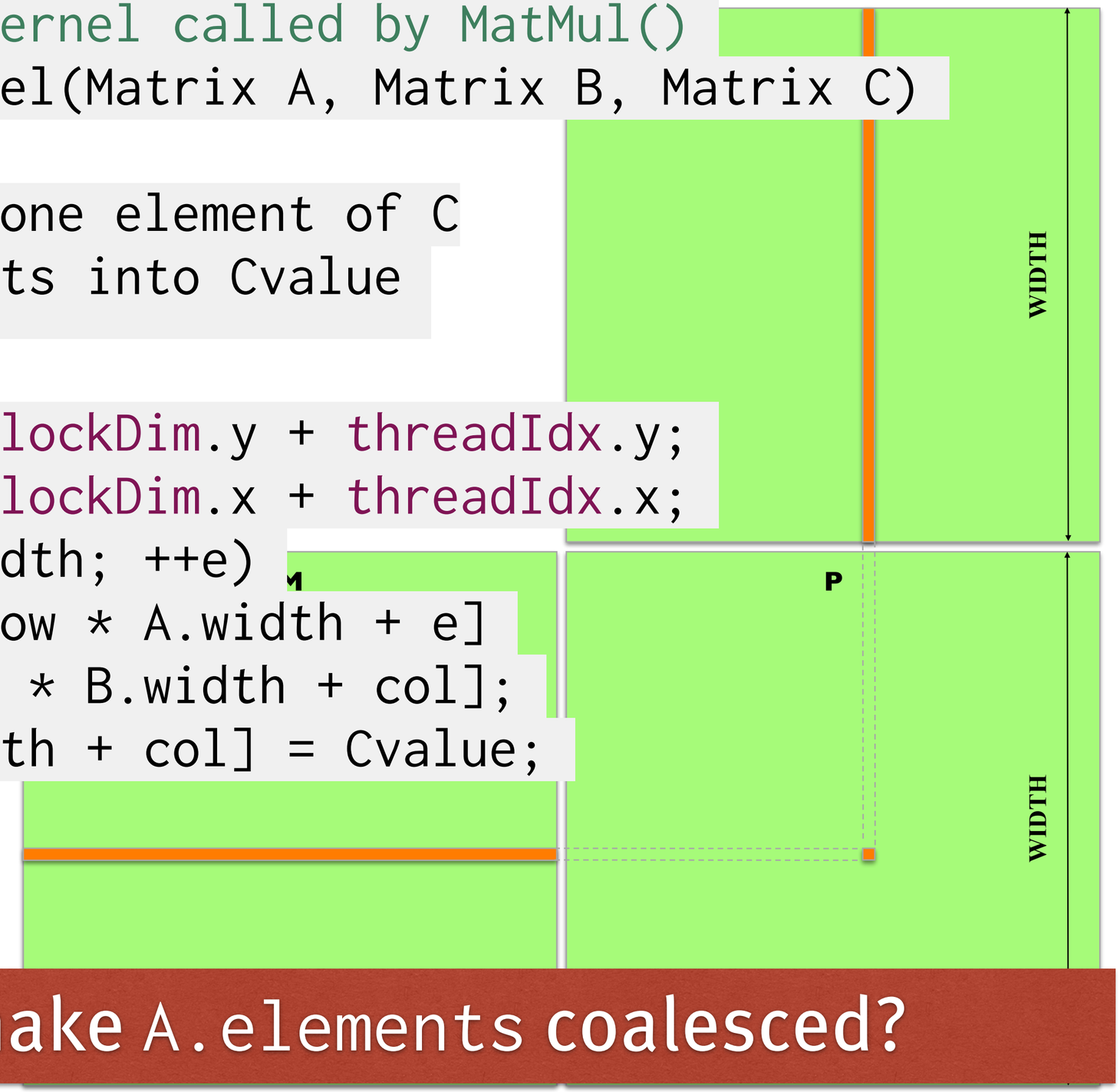
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue

  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
            * B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Q2: Out of the 3 global memory reads—
{A,B,C}.elements—which accesses are coalesced?

Dense Matrix Multiplication

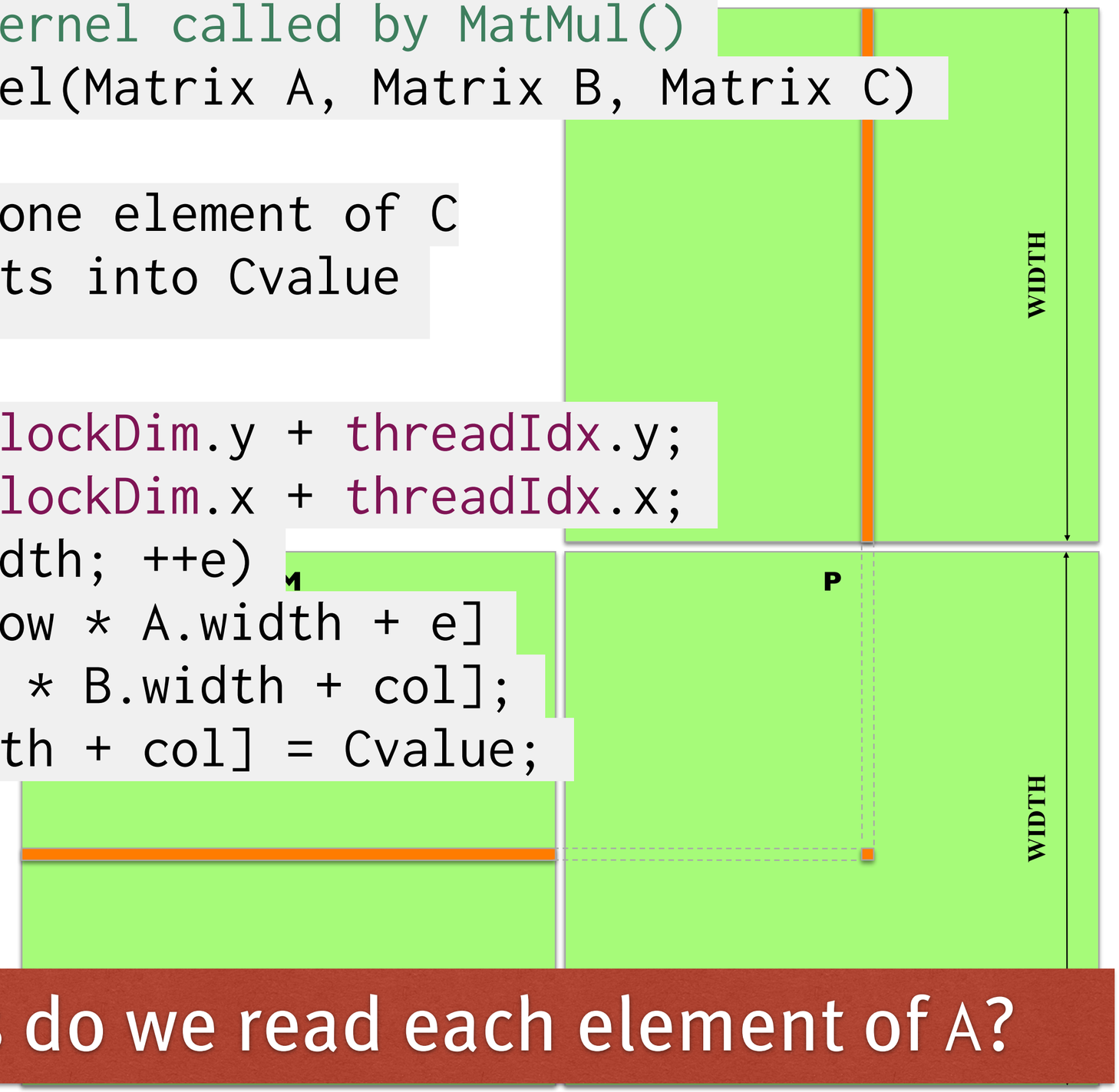
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue

  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
            * B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Q3: How do we make `A.elements` coalesced?

Dense Matrix Multiplication

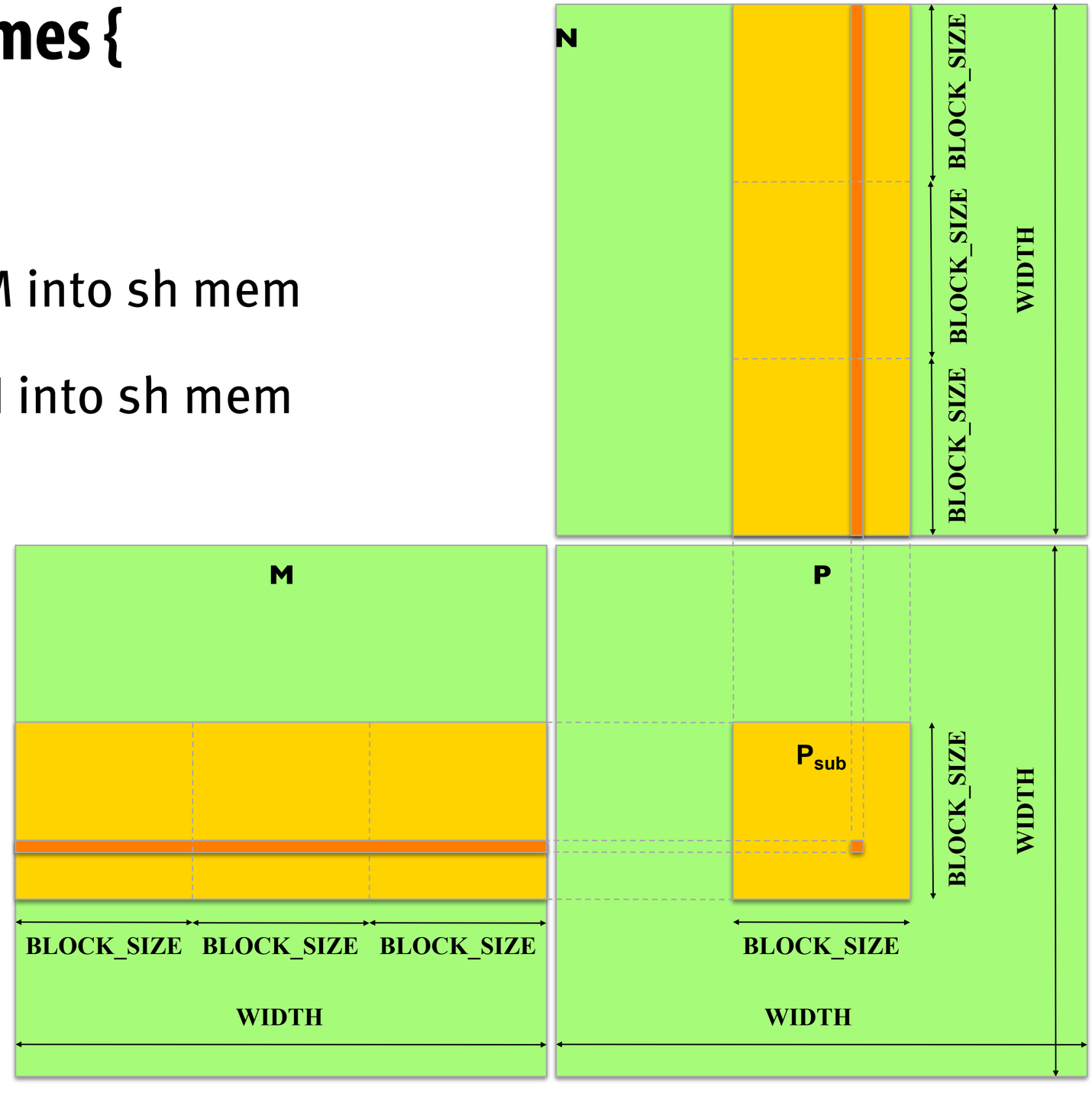
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue

  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  for (int e = 0; e < A.width; ++e)
    Cvalue += A.elements[row * A.width + e]
            * B.elements[e * B.width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```



Q4: How many times do we read each element of A?

Dense Matrix Multiplication

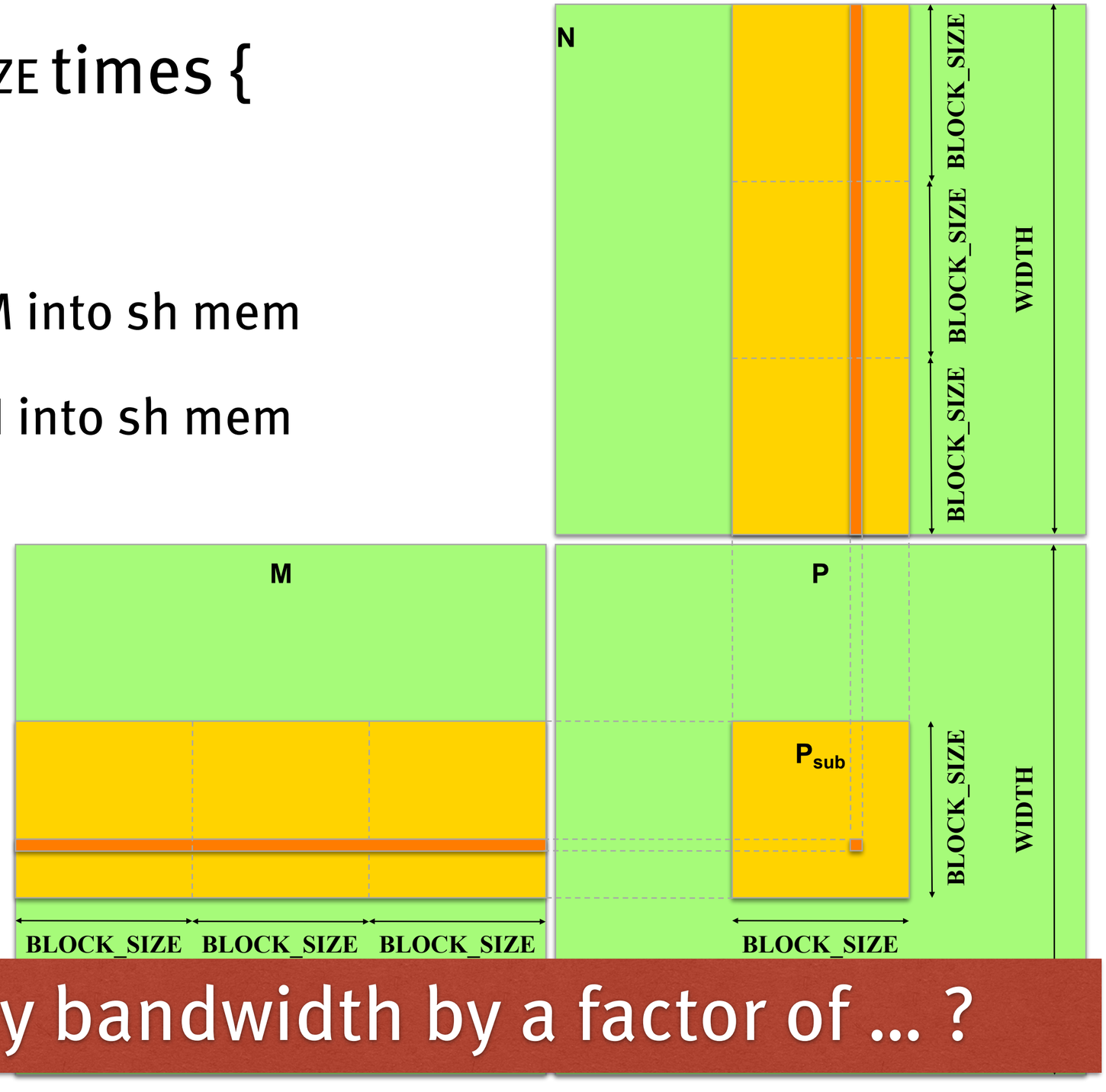
- **P = M * N of size WIDTH x WIDTH**
- **Repeat WIDTH / BLOCK_SIZE times {**
  - Each thread block:
    - Reads a subblock from M into sh mem
    - Reads a subblock from N into sh mem
  - Each thread:
    - Computes a mini-dot-product
- **Output total sum**



```
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);
        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);
        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();
        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];
        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }
```

Dense Matrix Multiplication

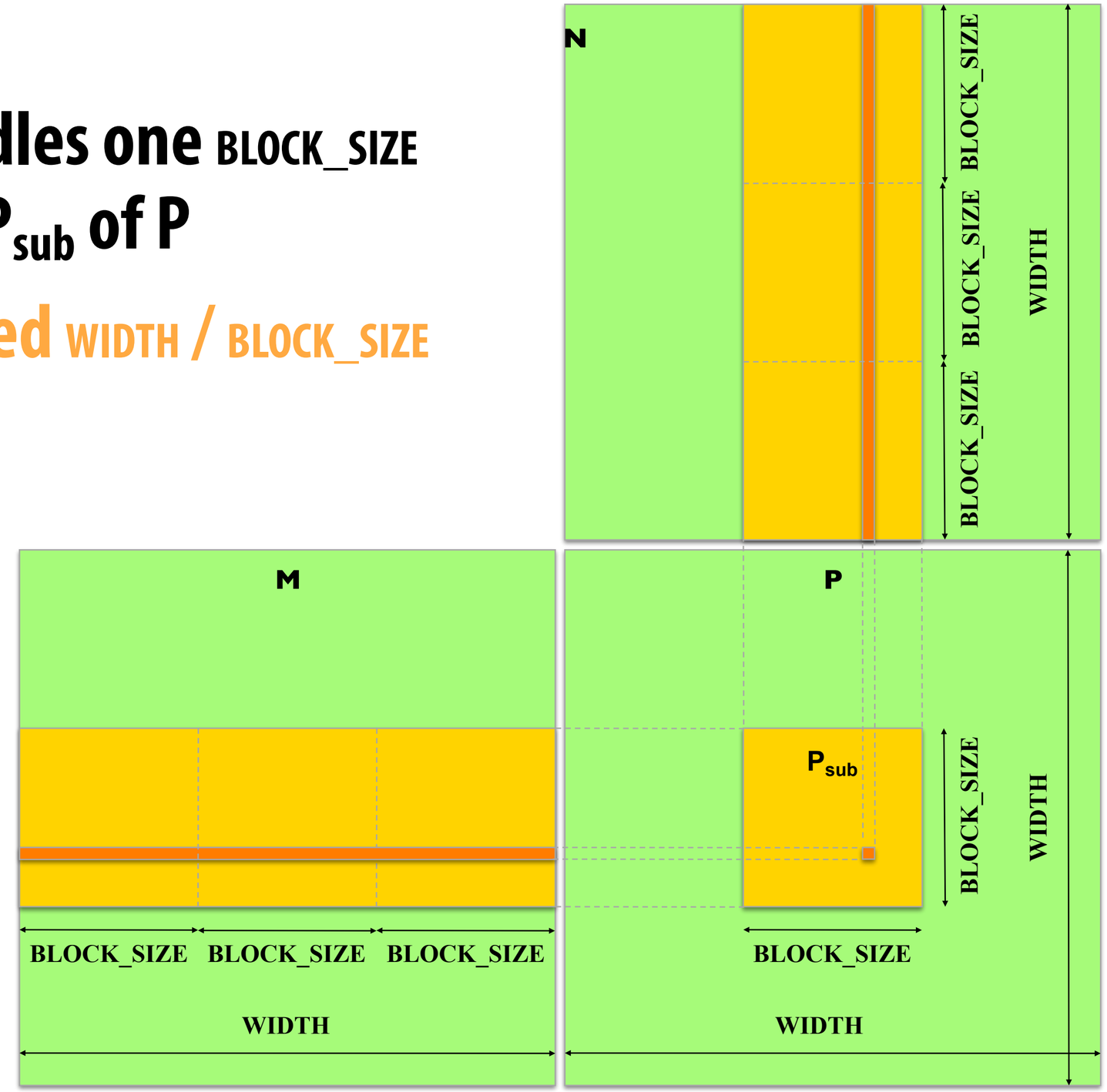
- P = M * N of size WIDTH x WIDTH
- Repeat WIDTH / BLOCK_SIZE times {
  - Each thread block:
    - Reads a subblock from M into sh mem
    - Reads a subblock from N into sh mem
  - Each thread:
    - Computes a mini-dot-product
- Output total sum



Q5: We save memory bandwidth by a factor of … ?

Dense Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- With blocking:
  - One thread block handles one BLOCK_SIZE x BLOCK_SIZE sub-matrix P_{sub} of P
  - M and N are only loaded WIDTH / BLOCK_SIZE times from global memory
- Great saving of memory bandwidth!



Basic Efficiency Rules

- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
- Exploit per-block shared memory as scratchpad
- Expose enough parallelism

Summing Up

- CUDA = C + a few simple extensions
  - makes it easy to start writing basic parallel programs

- Three key abstractions:
  - hierarchy of parallel threads
  - corresponding levels of synchronization
  - corresponding memory spaces

- Supports massive parallelism of manycore GPUs