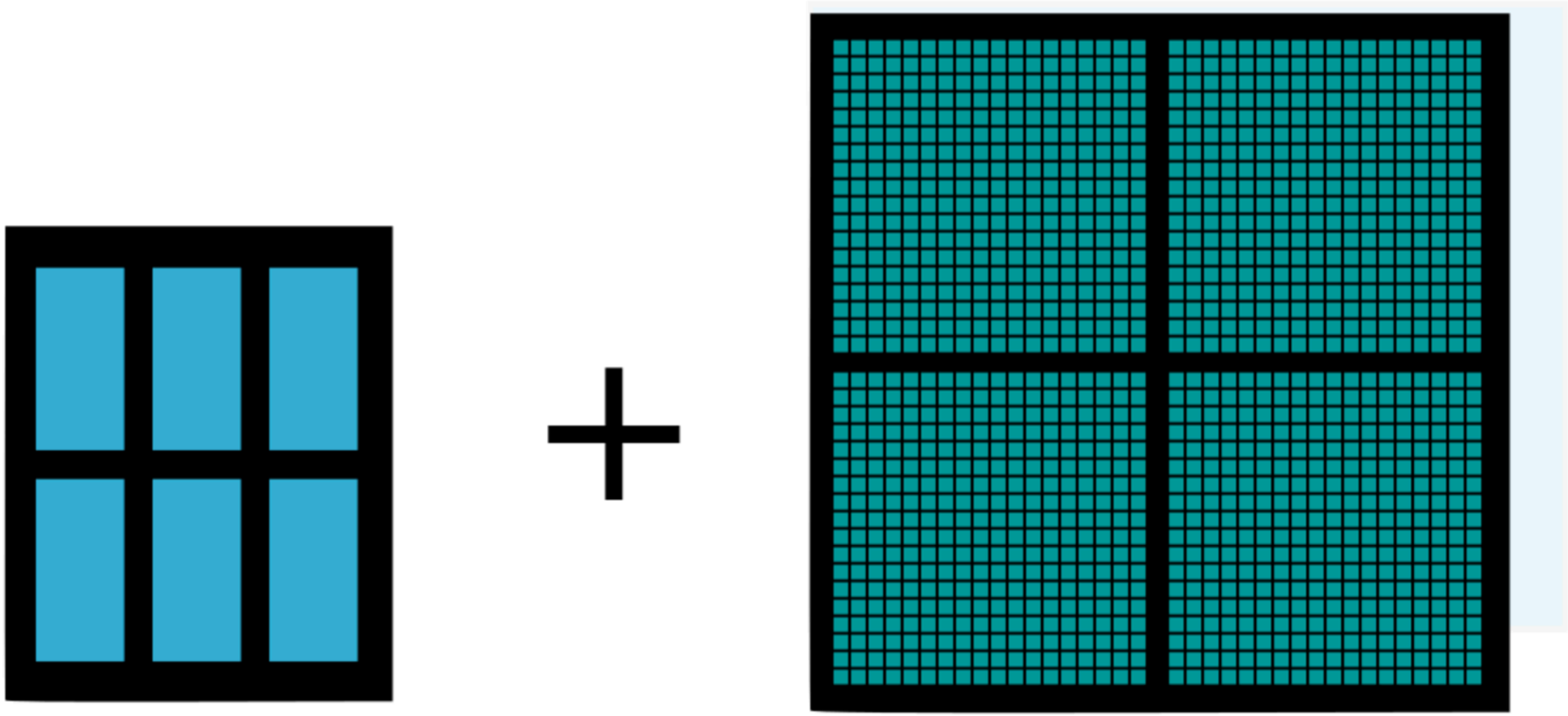
CMOR 421/521
GPU parallelism

Jesse Chan

Why have GPUs become so popular?

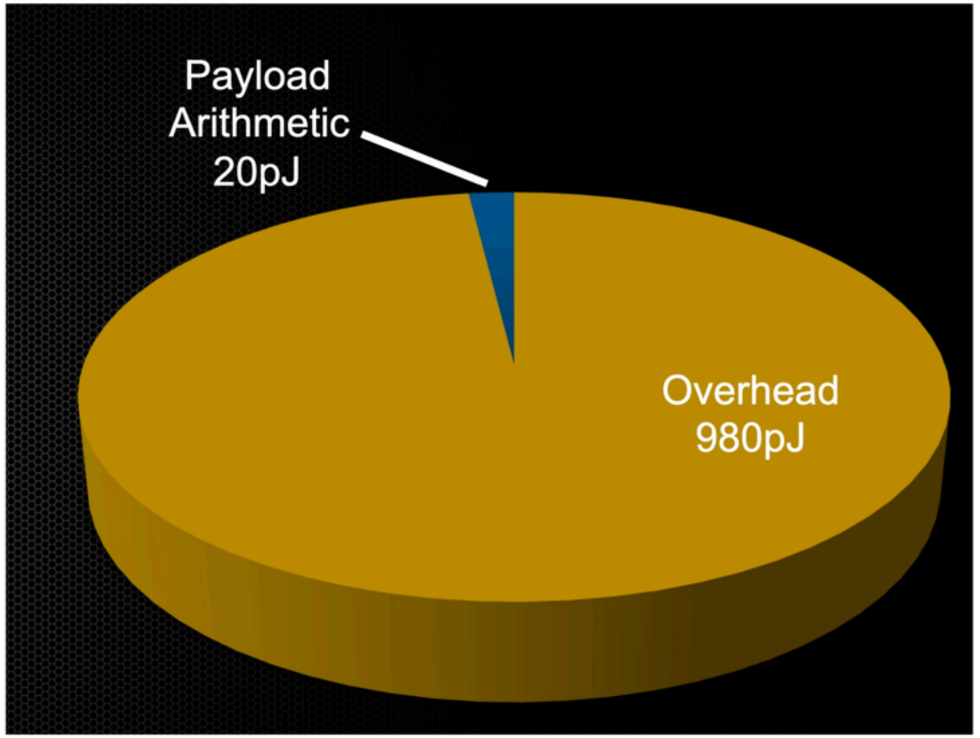
An oversimplified explanation

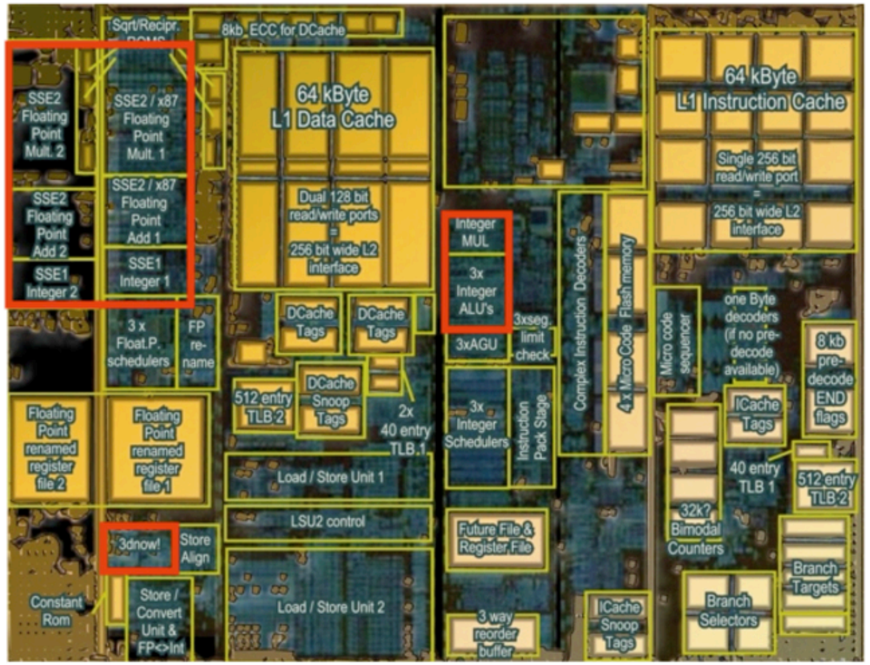


CPU
Multiple Cores

GPU
Thousands of Cores

CPU design is more complex





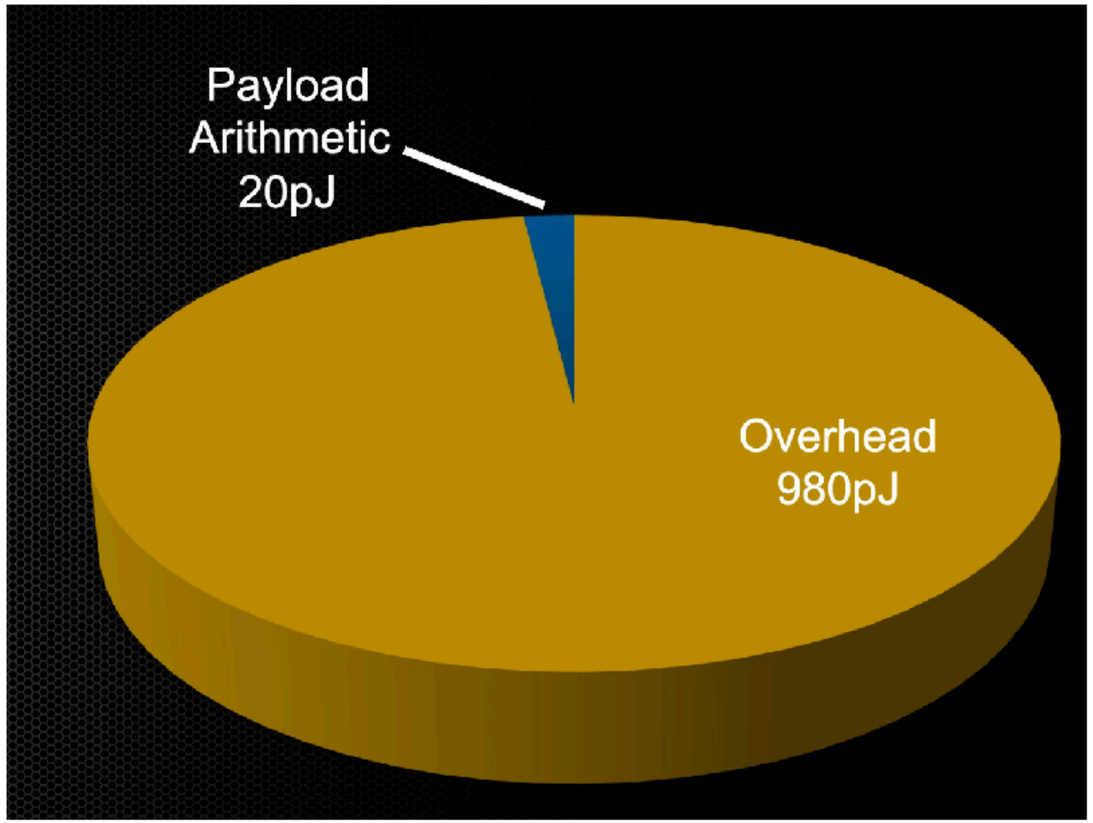
chip-architect.com

AMD K7 “Deerhound”

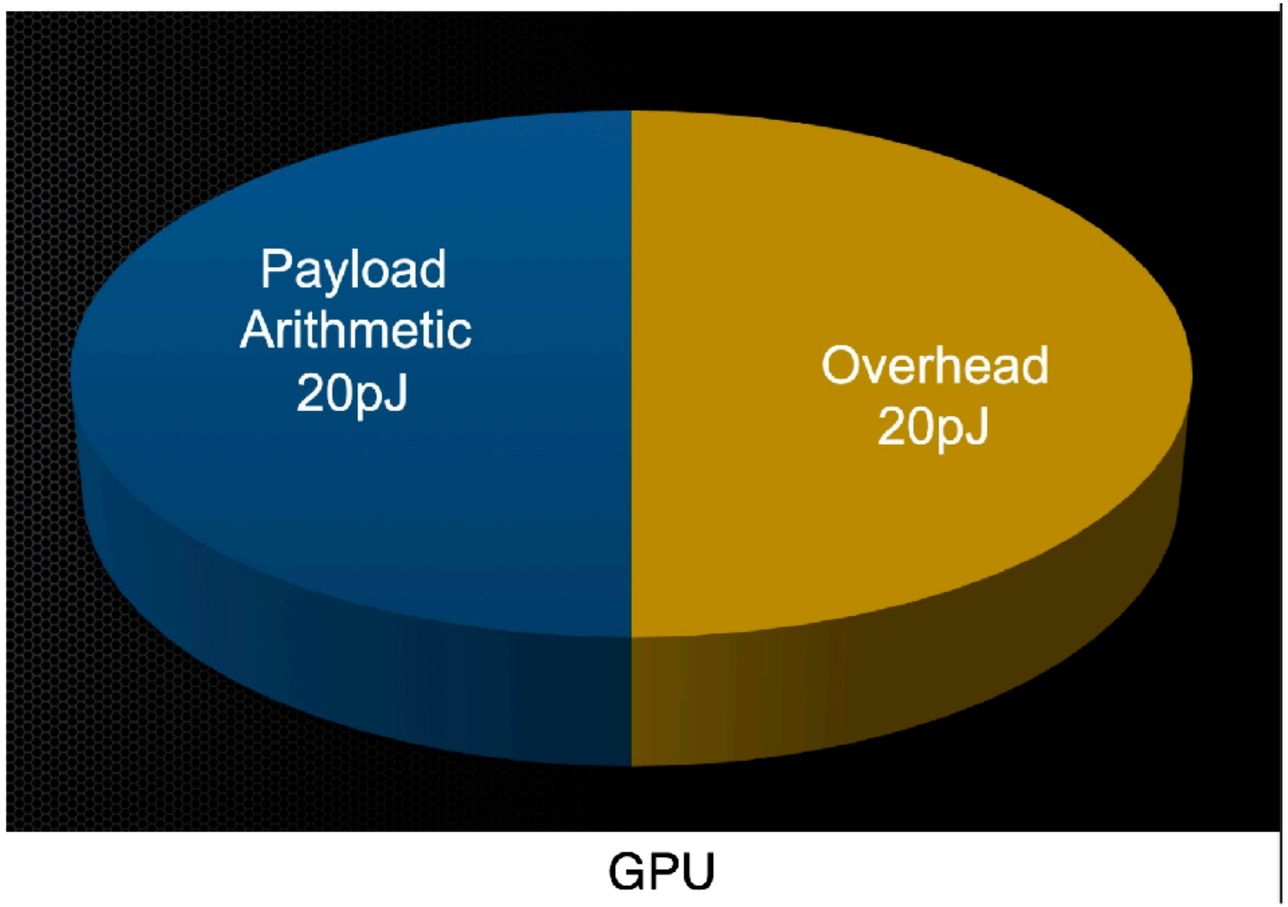
- Most CPU space and energy is not actually used to perform computations
- A lot of the CPU is dedicated to instructions, scheduling, etc.

Figure from Aydin Buluc, Bill Dally (2015)

CPU vs GPU energy expenditure



CPU



GPU

GPUs address energy constraints

- Larger supercomputers need more and more energy, starting to hit physical limitations as number of transistors increases (Dennard scaling)

> "A modern supercomputer usually consumes between 4 and 6 megawatts—enough electricity to supply something like 5000 homes."

> If you tried to achieve an exaflops-class supercomputer by simply scaling Blue Waters up 100 times, it would take 1.5 gigawatts of power to run it, more than 0.1 percent of the total U.S. power grid. You'd need a good-size nuclear power plant next door. That would be absurd, of course, ...

https://spectrum.ieee.org/nextgeneration-supercomputers

GPUs: higher FLOPS for the same energy

- GPUs can execute a much higher number of FLOPs relative to a single CPU core
- About 10x speedup over modern multicore CPUs
- Note: Bill Dally works for Nvidia (beware marketing)

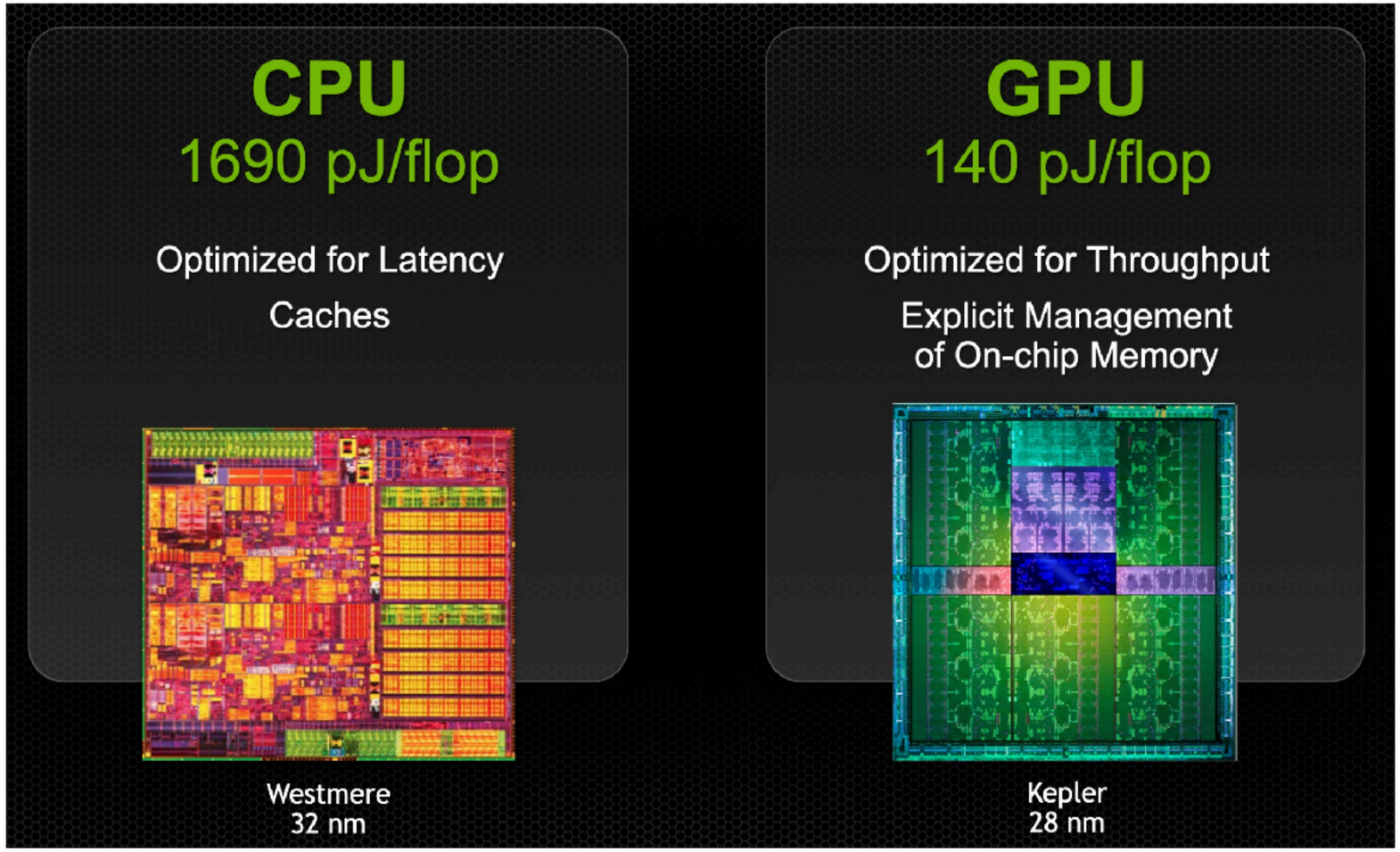


Figure from Bill Dally (2015)

Why are CPUs so complicated?

- CPUs have evolved as designers added features to speed things up.
- Left: CPU hardware used for actual computations
- Right side: CPU hardware that was introduced to minimize latency
  - Pack instruction cycles, Avoid memory latency

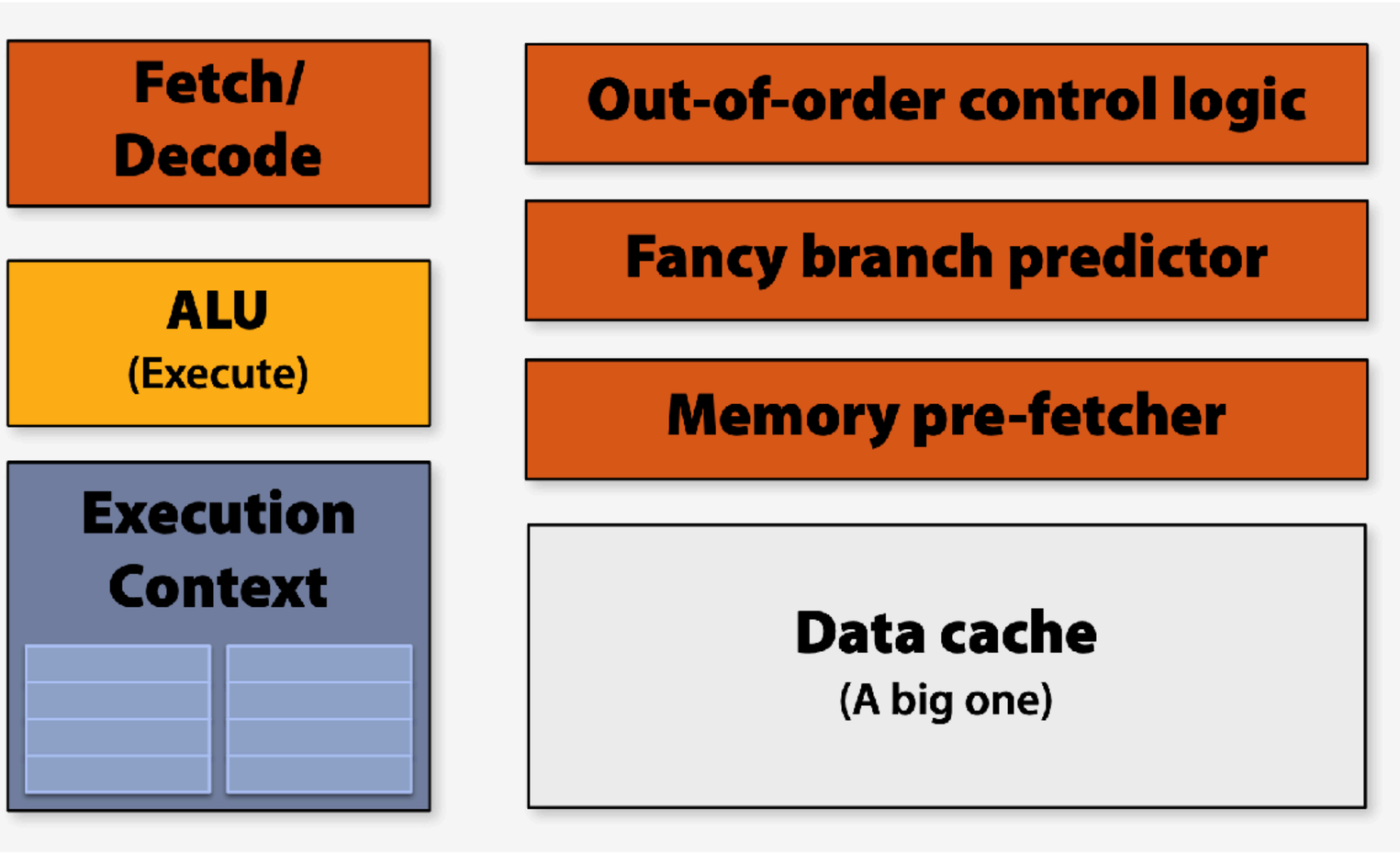


Figure from Aydin Buluc

From CPUs to GPUs: hardware idea 1

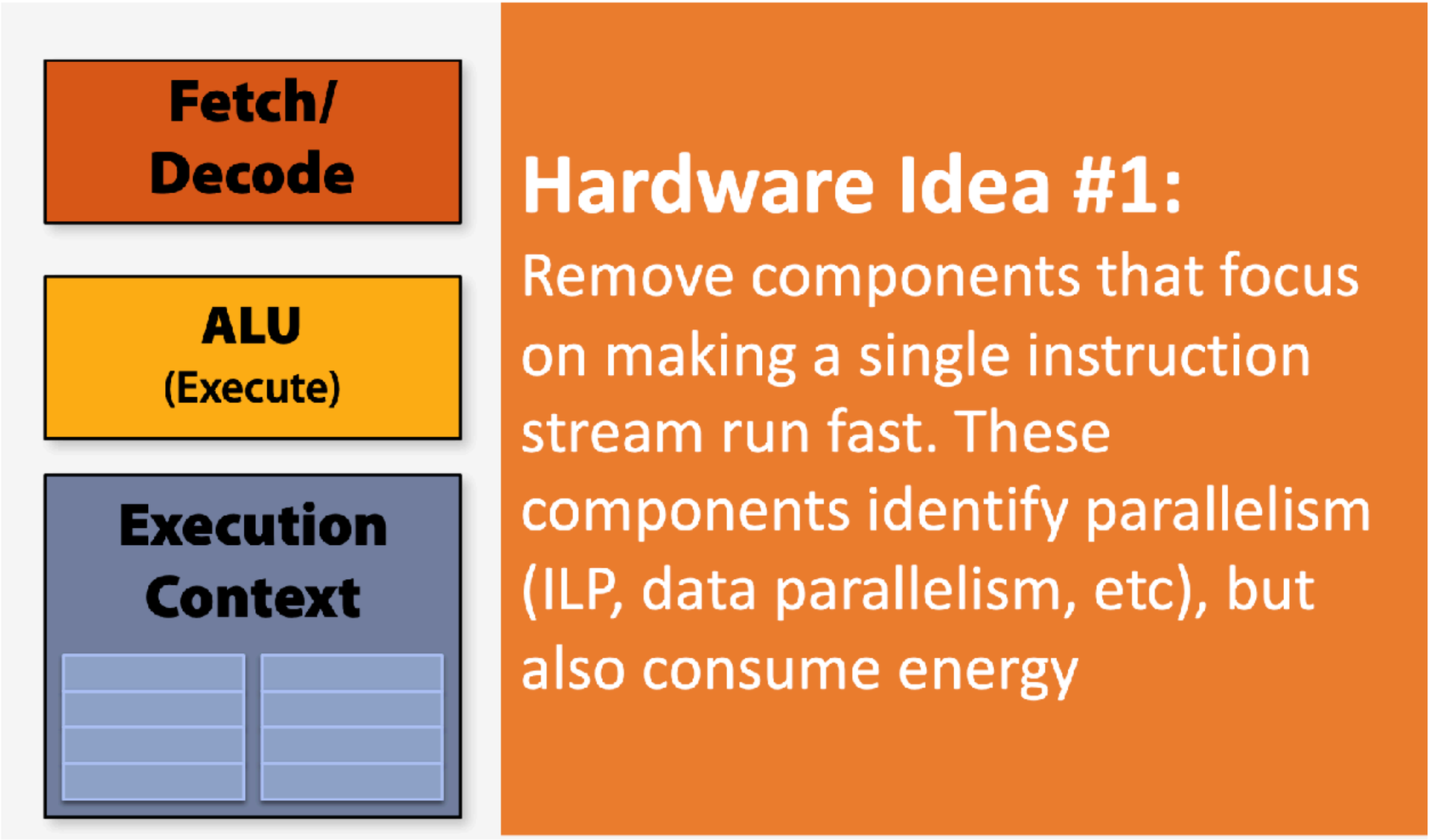
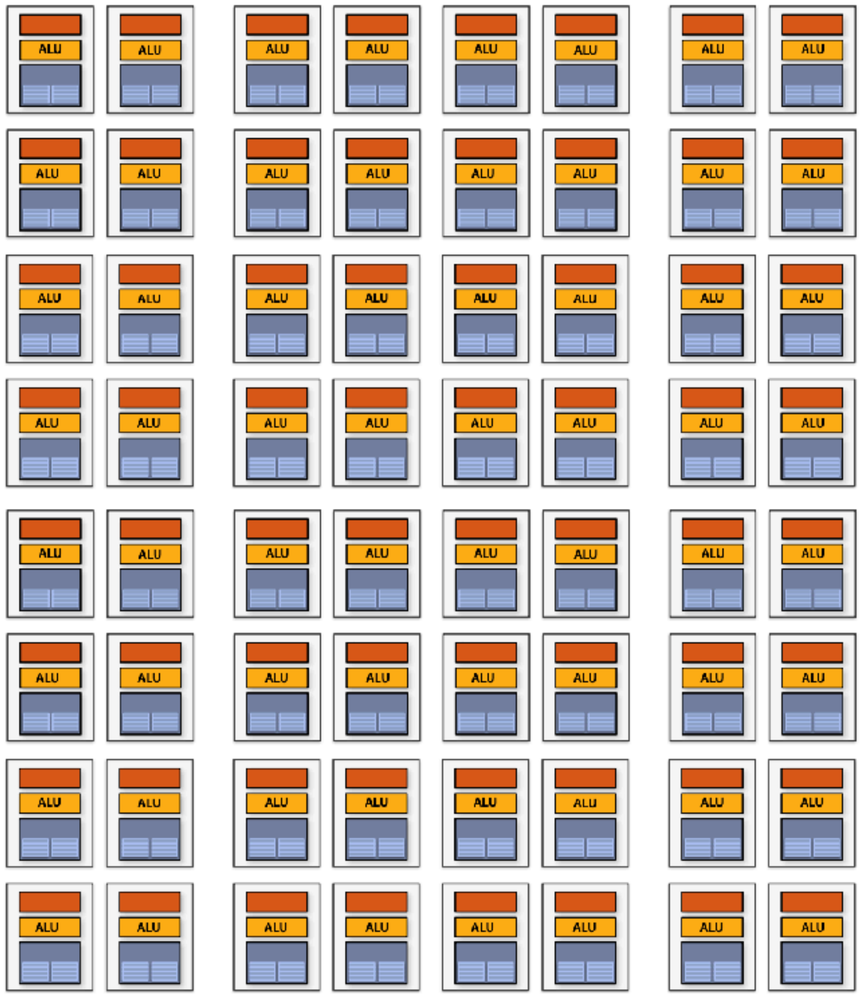


Figure from Aydin Buluc

From CPUs to GPUs: hardware idea 2



Hardware Idea #2:

- A larger number of (smaller and simpler) cores
- Encourages data parallelism (same operation applied to multiple instances of data)

Figure from Aydin Buluc

From CPUs to GPUs: hardware idea 3

- Multicore CPUs: each thread = core gets one set of instructions
- GPU “threads” reduce hardware by sharing instructions
  - A “warp” (currently 32) of threads on a GPU all must execute the same instruction simultaneously.

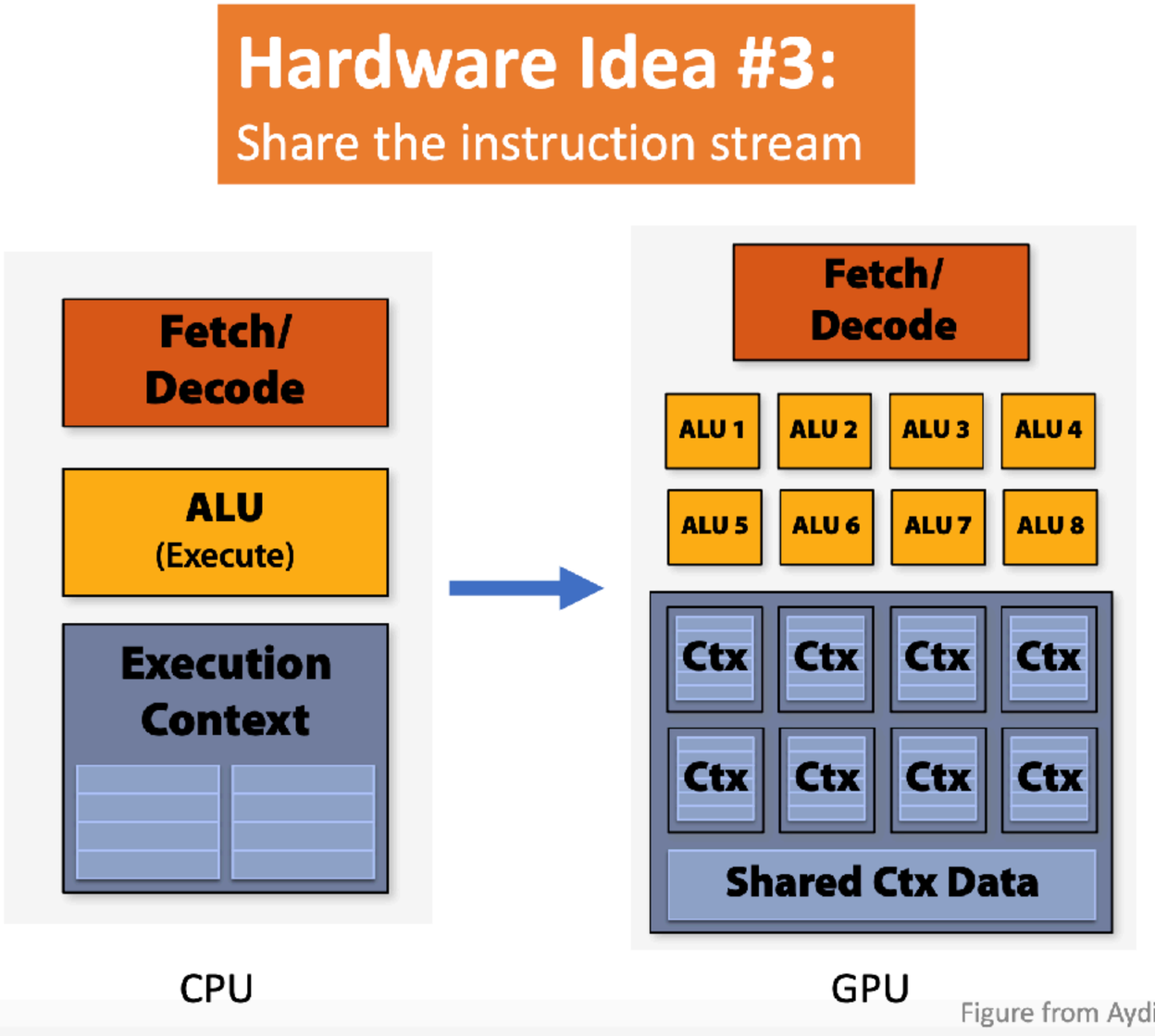
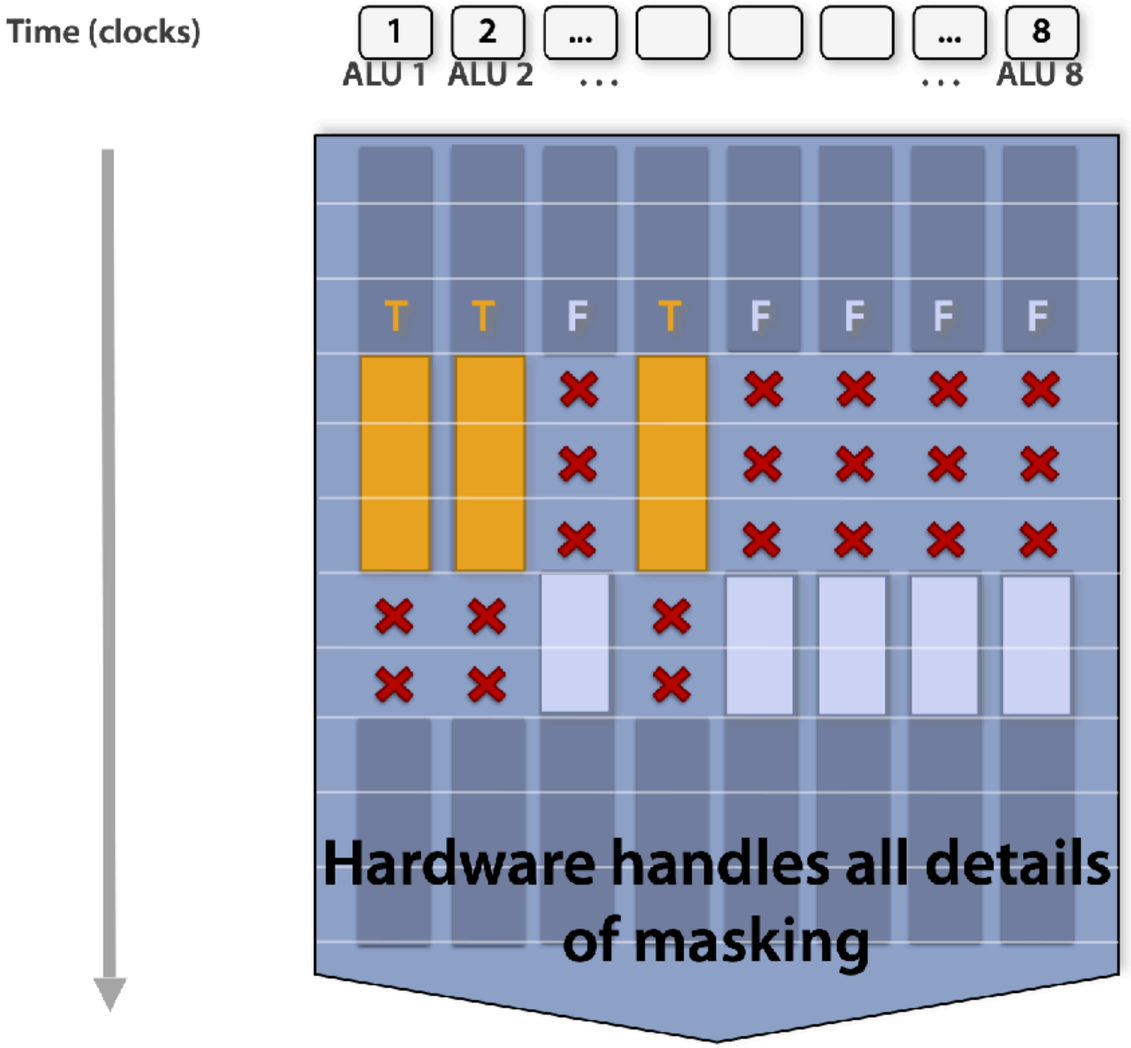


Figure from Aydin Buluc

From CPUs to GPUs: hardware idea 4

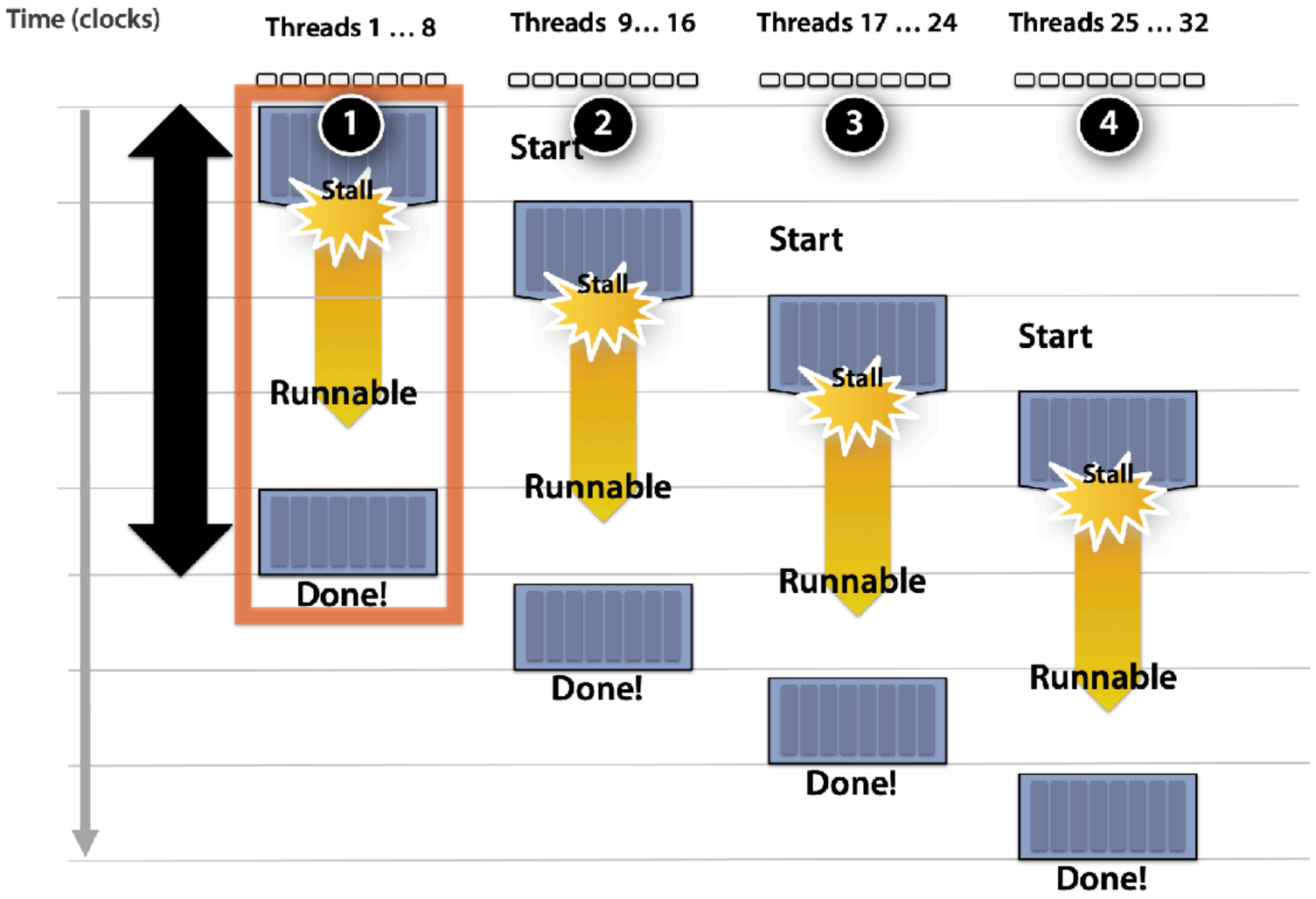



```
<unconditional
 program code>

if (x > 0) {
  y = pow(x, exp);
  y *= Ks;
  refl = y + Ka;
} else {
  refl = Ka;
}

     <resume
   unconditional
   program code>
```

Hardware Idea #4:
Replace branching with masks

Figure from Aydin Buluc

From CPUs to GPUs: hardware idea 5



Hardware idea #5:

Use threads to hide high latency operations.

- CPUs try to avoid "stalling" where one instruction waits on a previous one to complete.
- GPUs try to mitigate latency by staying busy (scheduling from the pool of "ready" thread instructions)

Figure from Aydin Buluc

If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?
-Seymour Cray, “The Father of Supercomputing”

GPUs caveats

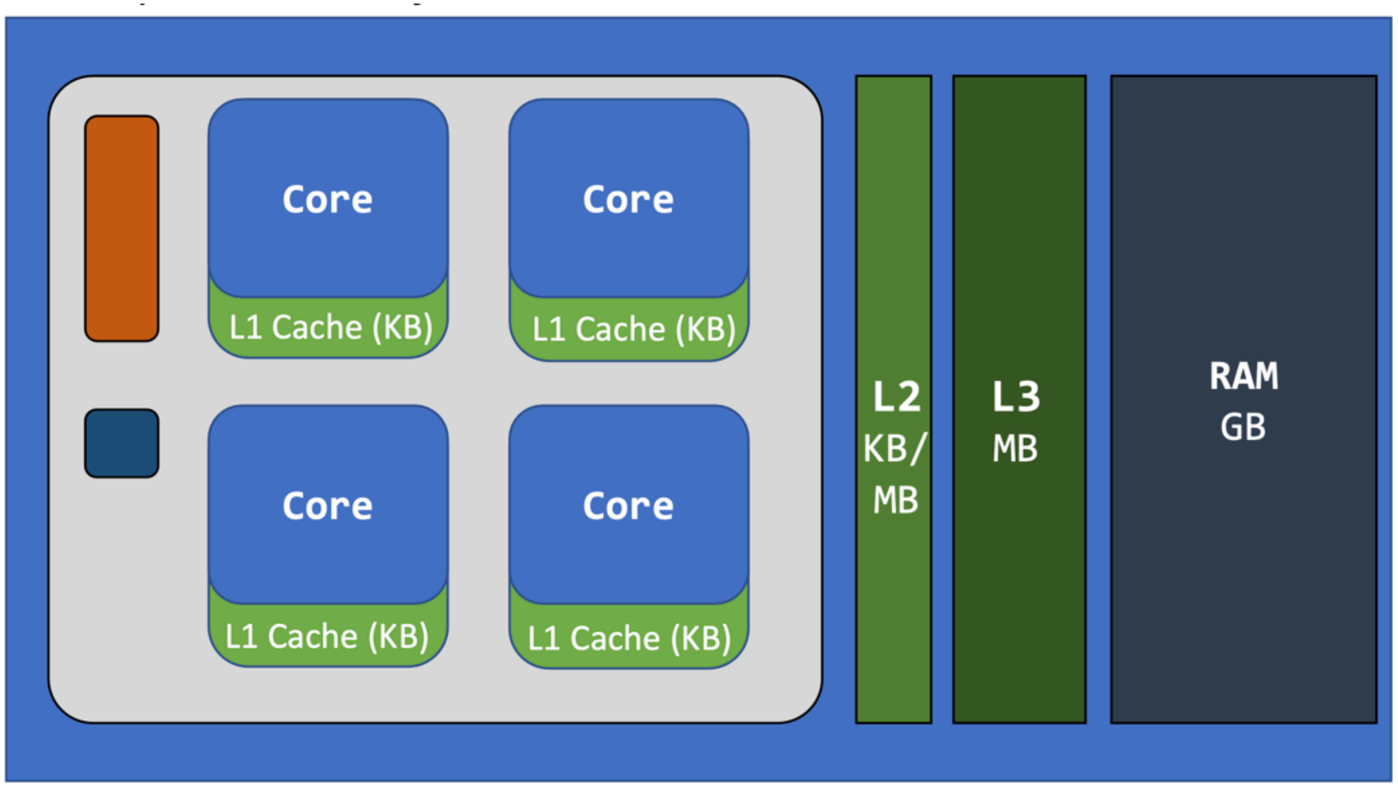
- GPUs aren’t very “smart”. They are highly efficient and specialized for a certain kind of problem —> they lack the generalizability of CPUs.
- It is often the case that:
  - The problems GPUs are good for, they are fantastic for!
  - The problems GPUs are bad at, they are ***extremely*** bad at.
  - Relative to CPUs, there is less middle ground between the two extremes
- Pretty common opinion: it’s much harder to develop on GPUs.

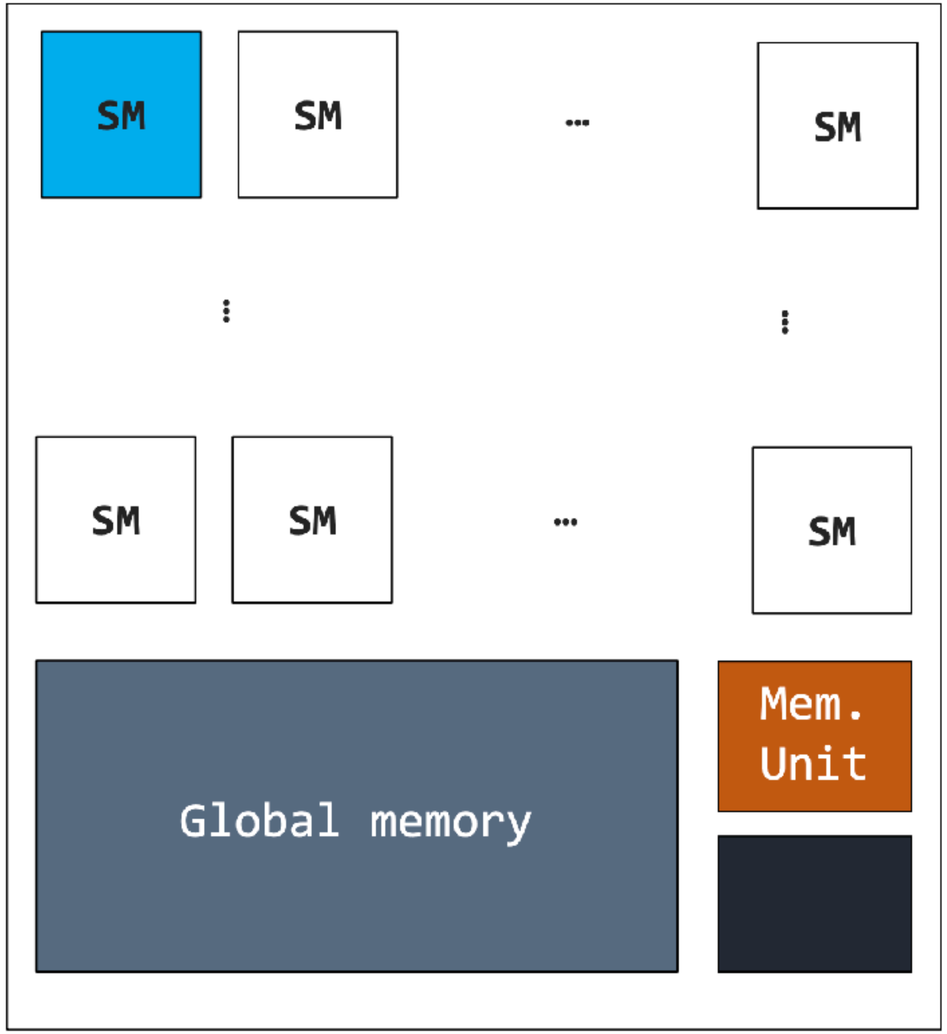
GPU disclaimers

- My familiarity with GPUs is in the context of scientific computing, **not at all for graphics.**
- There are several GPU (and GPU-like) manufacturers: Nvidia, AMD, Intel, Apple, and more.
  - They have different terminology (and different languages!) that have roughly the same features.
  - We are using Nvidia’s terminology and CUDA for this class.

CPU vs GPU architecture

CPU vs GPU architectures

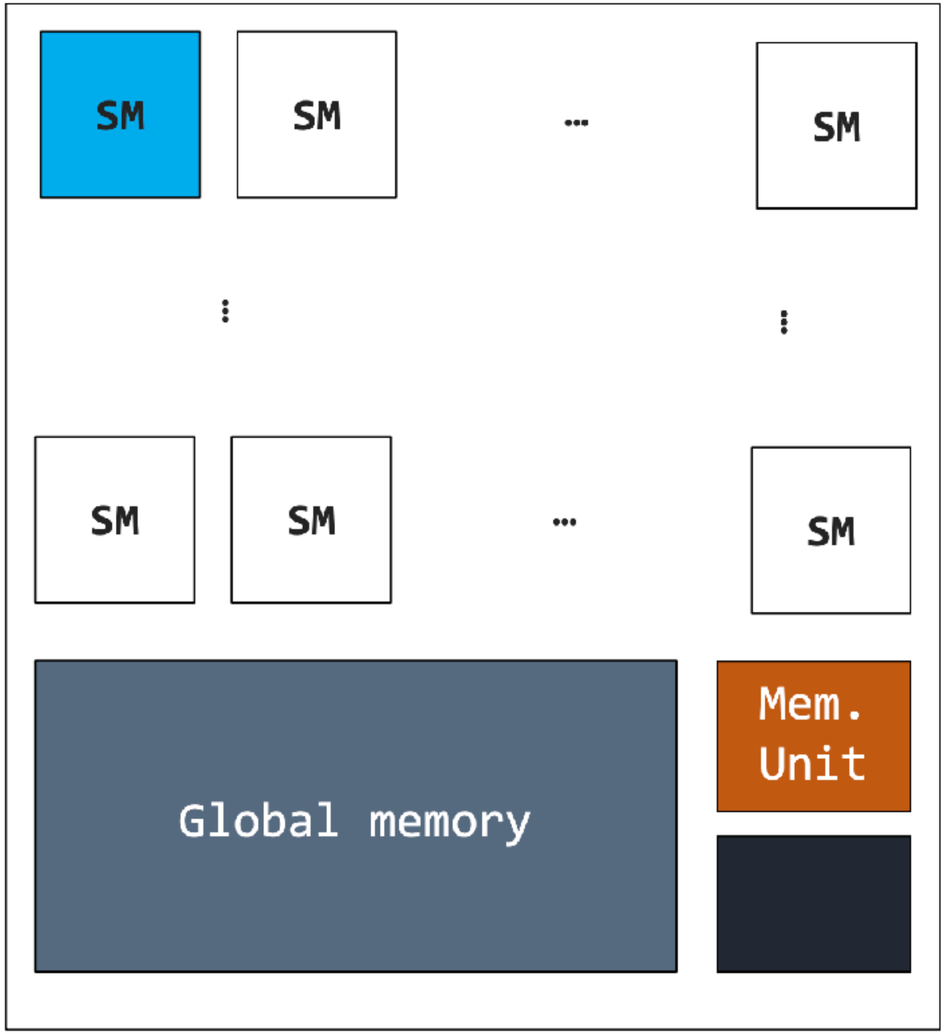




GPU architectures

GPU's have parallelism on top of parallelism

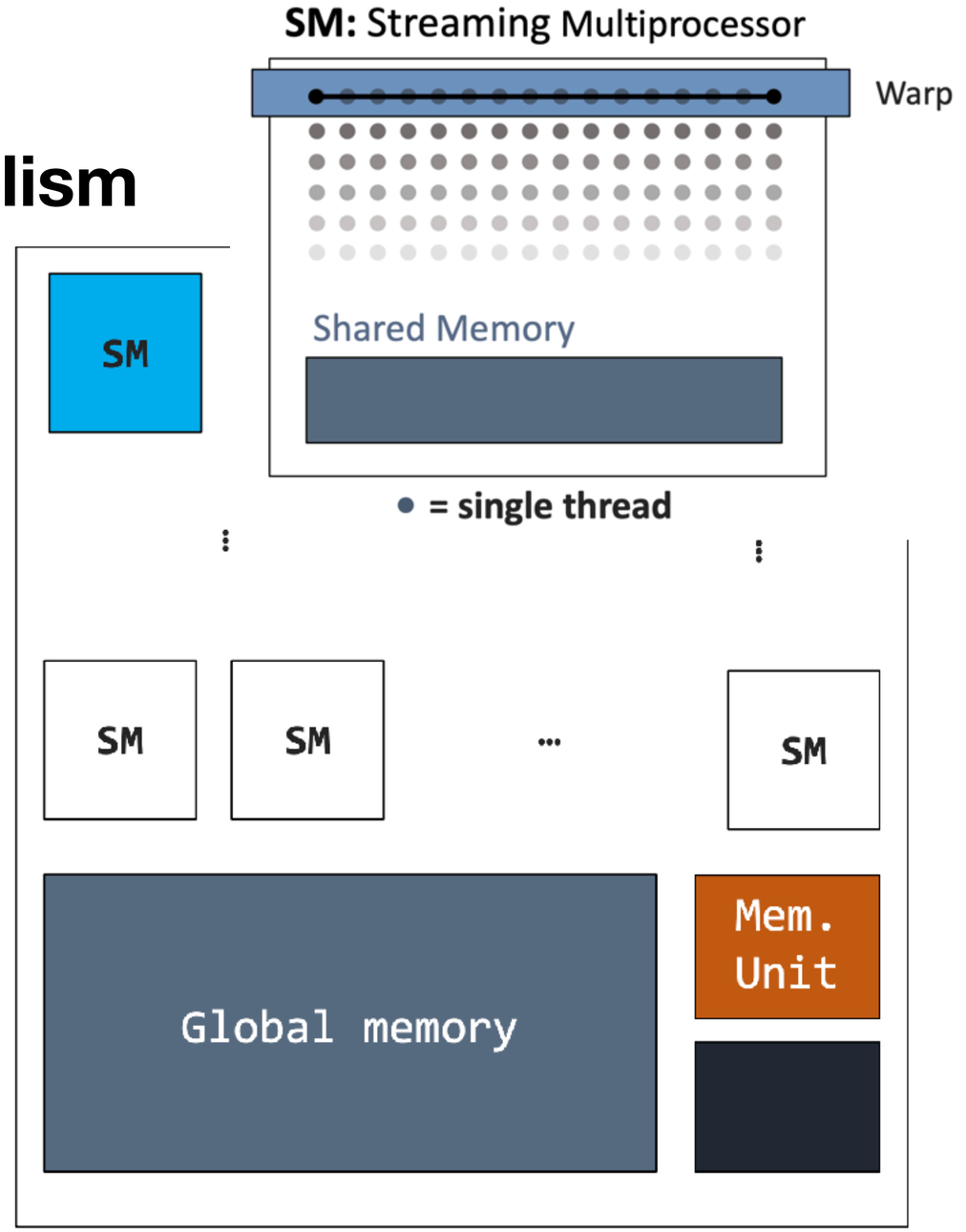
- GPUs are made up of
  - Streaming multiprocessors
  - Global memory (similar to RAM)
  - Memory (copy) unit, other stuff
- Streaming multiprocessors (SMs) perform arithmetic operations
  - SMs can access a shared memory pool, have multiple cores (organized into "warps")



GPU architectures

GPU's have parallelism on top of parallelism

- GPUs are made up of
  - Streaming multiprocessors
  - Global memory (similar to RAM)
  - Memory (copy) unit, other stuff
- Streaming multiprocessors (SMs) perform arithmetic operations
  - SMs can access a shared memory pool, have multiple cores (organized into "warps")



Some GPU numbers for scale

- Tesla K80 (2014, NOTS). Literally 2 K40 GPU glued together
  - Peak performance: **1.37 TFLOPS** (double precision), **4.1 TFLOPS** (single precision).
  - Peak bandwidth: **480 GB/s**
- Tesla P100 (2020):
  - Peak performance: **4.7 TFLOPS** (double precision), **9.3 TFLOPS** (single precision)
  - Peak bandwidth: **720 GB/s**
- For comparison, NOTS CPU performance/bandwidth: **166.4 GFLOPS, ~60 GB/s**

Comparison of CPU/GPU parallelism

- **CPU (Skylake)**
  - 40 cores (2 20-core chips)
  - 2 threads each
  - 2 x AVX-512 vectorization, so 2x8 in double precision
- ~640 way parallelism (40*2*8)

- **GPU (V100)**
  - 80 SMs
  - 64 warps per SM
  - 32 threads per warp in double precision
- ~150,000+ way parallelism (80*64*32)

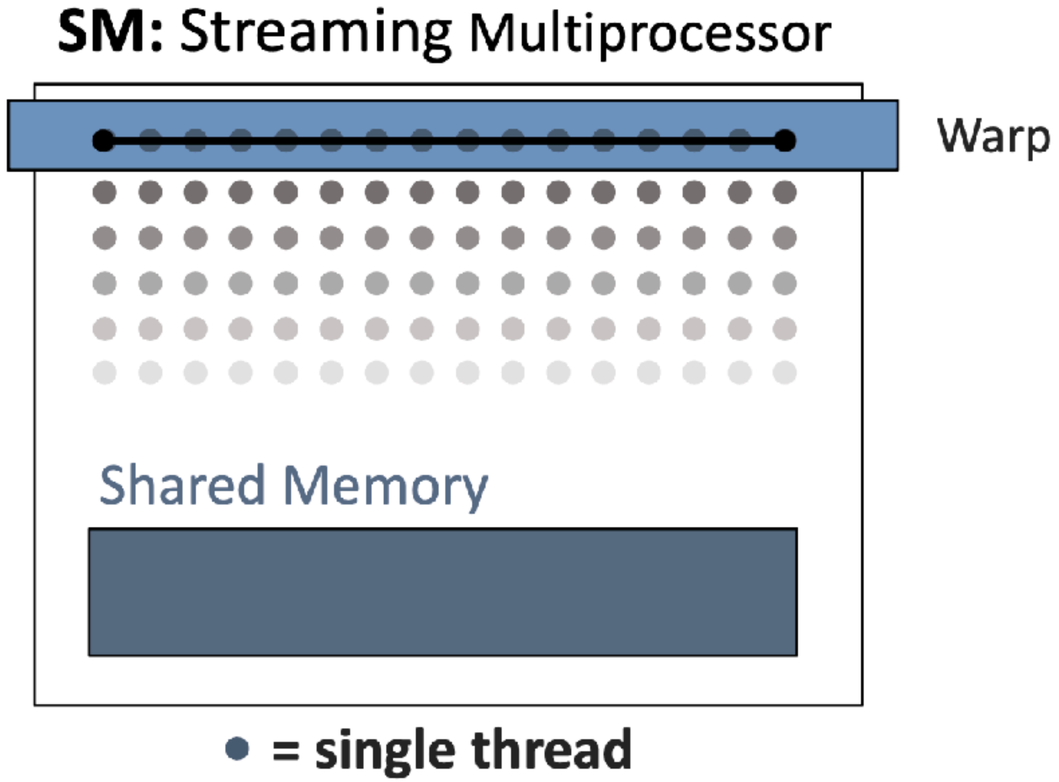
GPUs support 1000x more parallel operations than CPUs, but this is not necessarily an advantage. Insufficient parallelism under-utilizes GPUs.

How is GPU parallelism different?

- Streaming multiprocessors process commands using “warps” of threads
  - **All threads in a warp execute in lock-step (must do the same thing)**
- Threads execute using Single Instruction Multiple Thread (SIMT)
  - Execution is one “warp” at a time
  - There is a hardware limit on the total number of active warps

How is GPU parallelism different?

- Streaming multiprocessors process commands using “warps” of threads
  - **All threads in a warp execute in lock-step (must do the same thing)**
- Threads execute using Single Instruction Multiple Thread (SIMT)
  - Execution is one “warp” at a time
  - There is a hardware limit on the total number of active warps

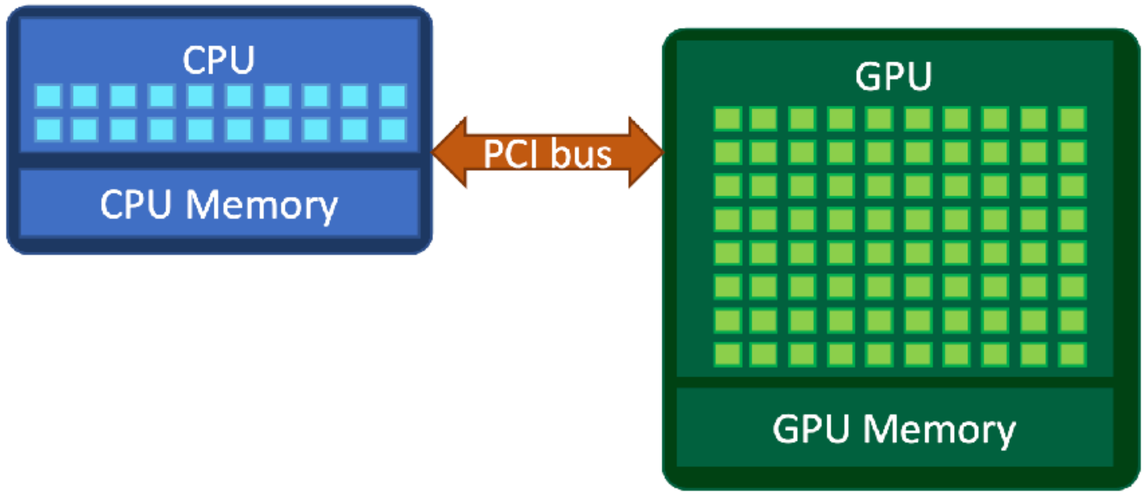


GPUs and memory

- Due to their immense computing power, GPUs are **more likely to be bottlenecked by communication/memory accesses than computation.**
- Exploiting the memory hierarchy is extremely important!
  - Accessing CPU memory from the GPU is **extremely** slow
  - Global memory (on the GPU) is slow (and should be done using *coalesced* memory accesses, similar to contiguous memory access on CPU).
  - Shared memory (local to a SM) is fast
  - Register memory (local to a core/thread) is fastest

Host-device paradigm

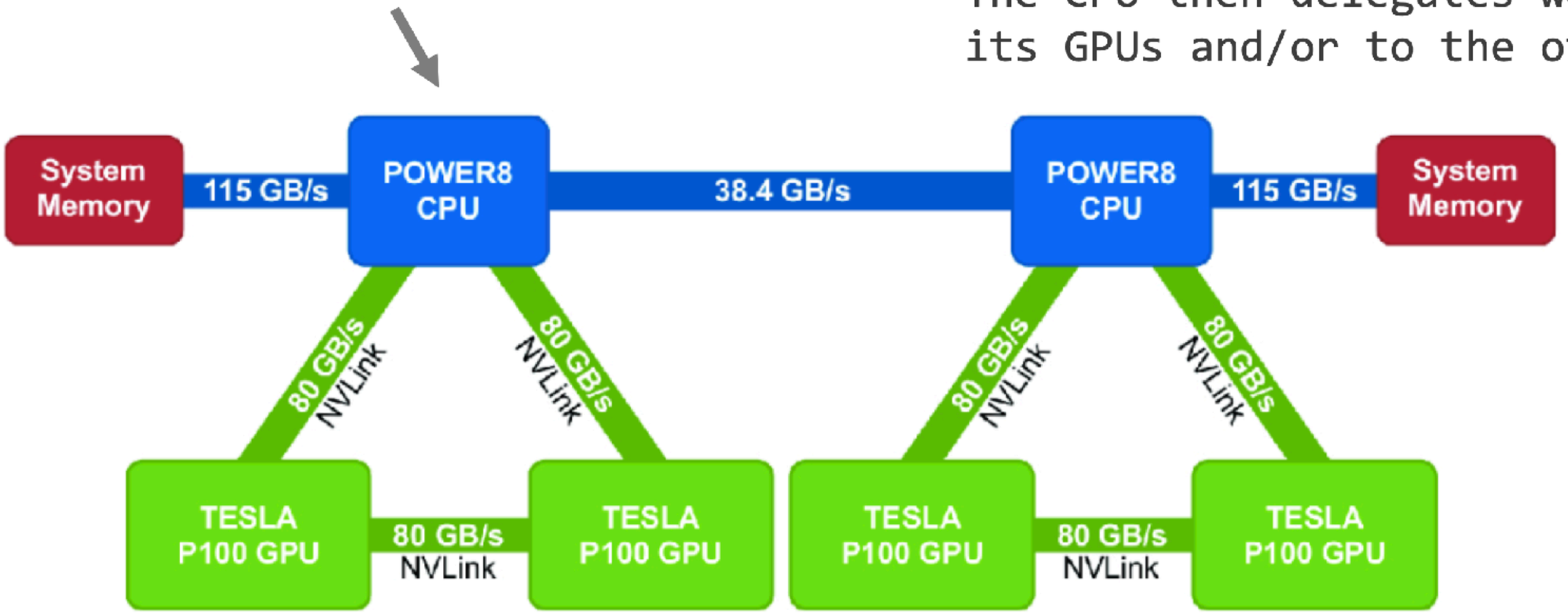
- CPUs may not have a GPU, or may have multiple GPUs. However, **a GPU (device) always has a CPU (host)** which "manages" the device.
- Device/host are like distributed workers with separate memory spaces.
  - Memory transfer from CPU to GPU is very slow (through the PCIE bus).



An example host-multiple-device setup

When you access the cluster, you are talking to one of the Power8 CPUs

The CPU then delegates work to its GPUs and/or to the other CPU



Programming GPUs: CUDA

A little history about CUDA

- Around early 2000's, researchers realized GPUs were highly efficient for scientific computing (dense linear algebra), but had to translate problems to graphics-specific languages (e.g., OpenGL)
- In 2003, Ian Buck introduced the precursor to the 2007 CUDA (Compute Unified Device Architecture)
- Opinion: CUDA is by far the dominant GPU language.
  - Everyone else has been playing catch-up to NVIDIA since ~2010 (e.g., OpenCL/SYCL, OpenACC, OpenMP, ROCm/HIP, etc).

Launching a CUDA kernel

- Host (CPU) code sets up and launches device (GPU) kernels.
  - Memory is typically allocated on the CPU and copied to the GPU.
- CUDA is a subset of C/C++, but not all functionality is efficient in CUDA
  - Example: if you make a call to “new” in your main function, it will allocate on the CPU. However, if you allocate using “new” in a kernel, every thread will try to allocate and it will be be very slow.

A simple CUDA kernel: add_vectors.cu

- Can request interactive NOTSx node with a GPU. Note: there are a limited number of GPUs, so you may wait to get an interactive node!
  - srun --pty --partition=commons --reservation=cmor421 --ntasks=1 --cpus-per-task=1 **--gres=gpu** --mem=5G --time=00:30:00 $SHELL
- To develop on NOTSx, you can:
  - Open a terminal and “ssh” into NOTSx (use “scp” to copy files back and forth between your laptop and NOTSx)
  - Connect VSCode using View -> Command palette ->

- Compiling and running the code:
  - To compile, “nvcc add_vectors.cu -o add_vectors” (arguments are same as gcc/g++). To run the executable, run “./add_vectors”.

Debugging add_vectors.cu

- I needed to load *specifically* GCC/12.3.0 and CUDA/12.1.1 to run the kernel!
- I suggest using CUDA error checking to figure out what went wrong!

```
cudaError_t code = cudaGetLastError();
if (code != cudaSuccess){
  printf("GPUassert: %s\n", cudaGetErrorString(code));
}
```

- Demo…

CUDA memory allocation

- CUDA kernels are functions (written in a subset of C + CUDA keywords) to be run on the GPU; the rest of the program is run on the CPU.
- Data is moved to the GPU using CUDA versions of C commands
  - cudaMalloc
  - cudaMemcpy

```
int N = 1e6;
float * x = new float[N];
float * y = new float[N];
```

```
// allocate memory and copy to the GPU
float * d_x;
float * d_y;
cudaMalloc((void **) &d_x, size);
cudaMalloc((void **) &d_y, size);

// copy memory over to the GPU
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
```

Running a CUDA kernel

- **Thread block** size and **grid dimensions** are specified when invoking a CUDA kernel.
  - A thread block is the group of threads that the kernel should be run on, can be organized into 1D, 2D, or 3D arrays.
  - Grid dimensions specify how many blocks are run.
- Each thread block is assumed to be independent from other blocks!

```
// call the add function
int blockSize = 128;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, d_x, d_y);

// copy memory back to the CPU
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
```

The most efficient GPU codes “offload” the computation to the GPU, transferring data between CPU/GPU only once at the beginning and once at the end of the computation.

Structure of a CUDA kernel

- Key idea: code inside of a CUDA kernel is assumed to be run on **every core/thread** of every block.
- If there are more cores than elements in x, y, this means we have to manually “mask” threads for which i >= N.
- There are “blockDim” threads per block, and the number of blocks is set during the kernel call.

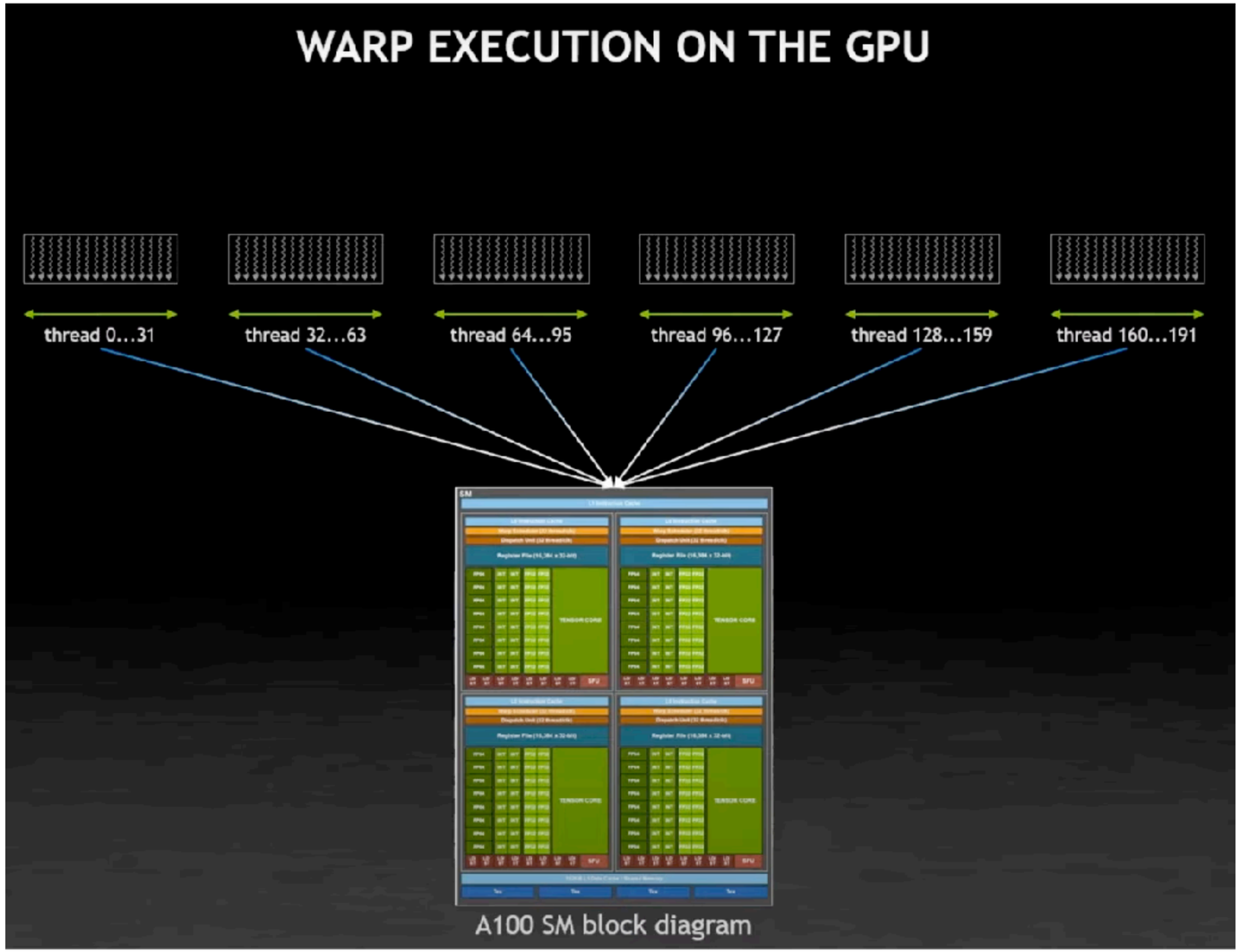
```
__global__ void add(int N, const float *x, float *y){

  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N){
    y[i] = y[i] + x[i];
  }
}
```

```
// call the add function
int blockSize = 128;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, d_x, d_y);
```

How are thread blocks executed?

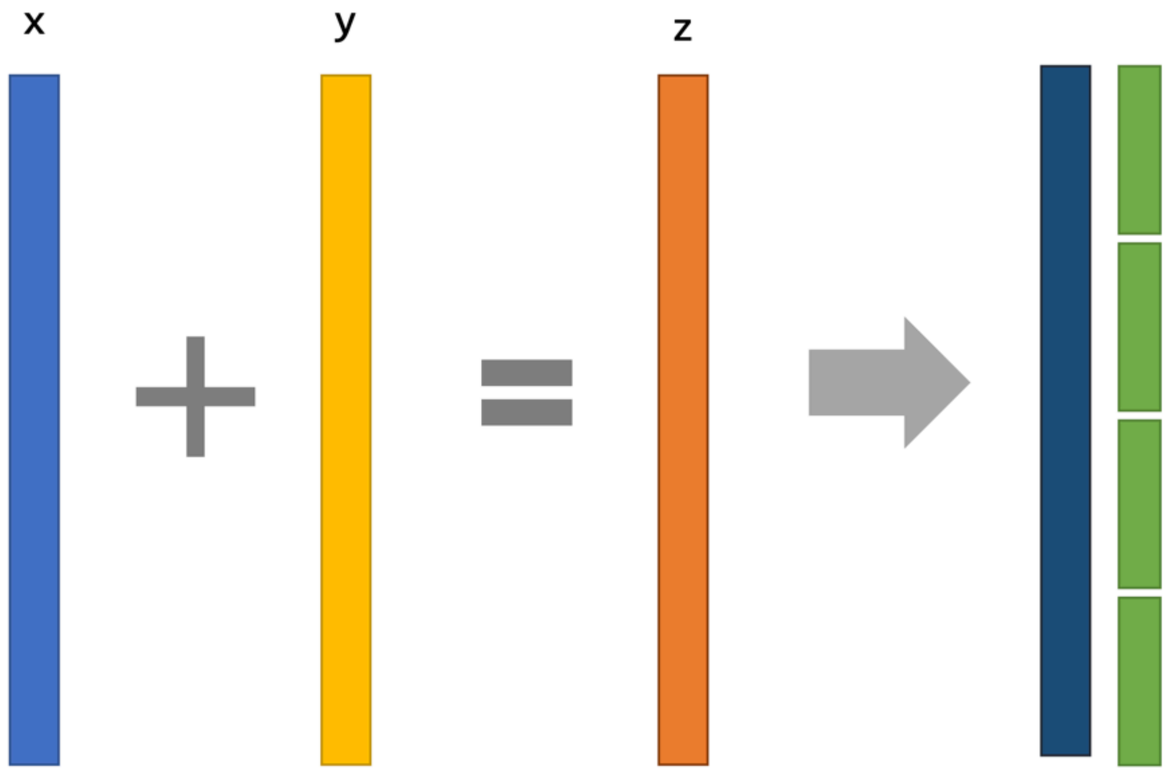
How do they map to streaming multiprocessor and warps?

- Thread blocks and grids are logical **software** concepts, streaming multiprocessors (SM) are **hardware** concepts.
- Thread blocks are scheduled to be run on available SMs.
  - Blocks are scheduled randomly until all blocks are executed.
- The warp is the smallest unit of execution on a GPU.
- More on this later…



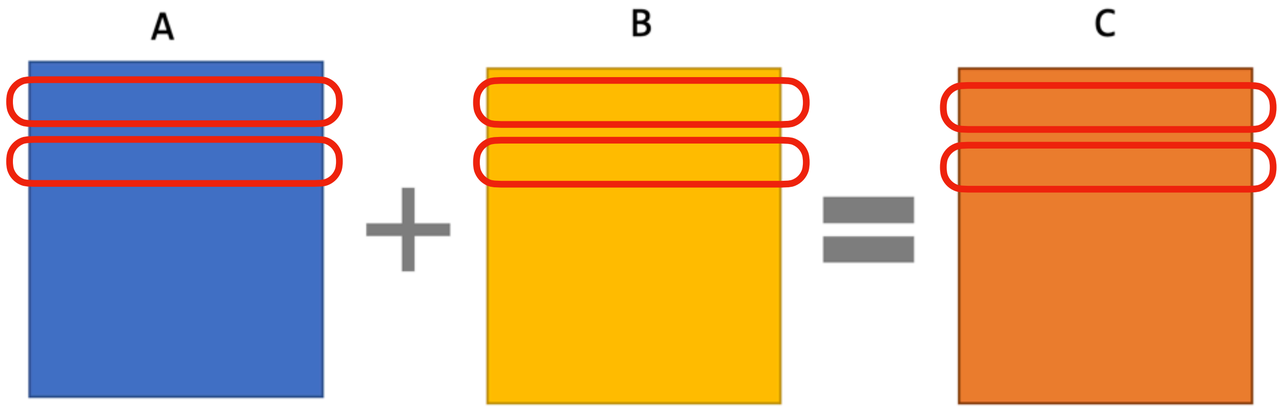
Problem decomposition on GPUs

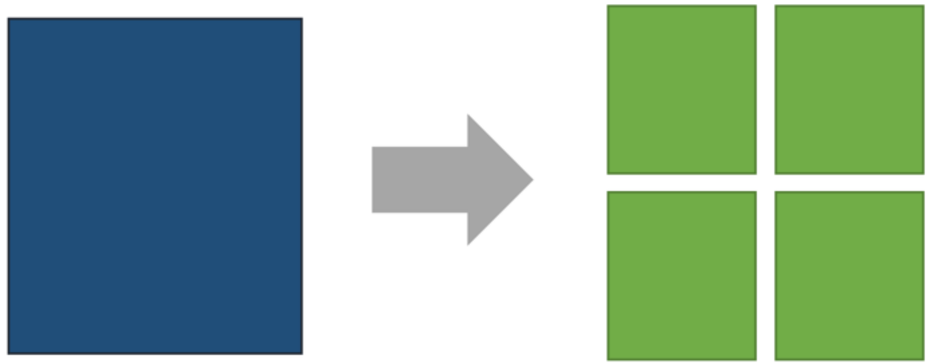
- For add_vectors.cu, we decompose vectors into chunks
- Each chunk is to be processed by one thread block.
- For example,
  - int threadsPerBlock = 128
  - int num_blocks = ceil(n/ threadsPerBlock)



Problem decomposition on GPUs: 2D arrays

- What about adding two matrices together instead?
  - Option 1: assign a thread block to each row or column
  - Option 2: assign a thread block to each “block” of a matrix.
- How do we implement each option?
- Demo: add_matrices.cu
  - What happens if the matrix size becomes too large?



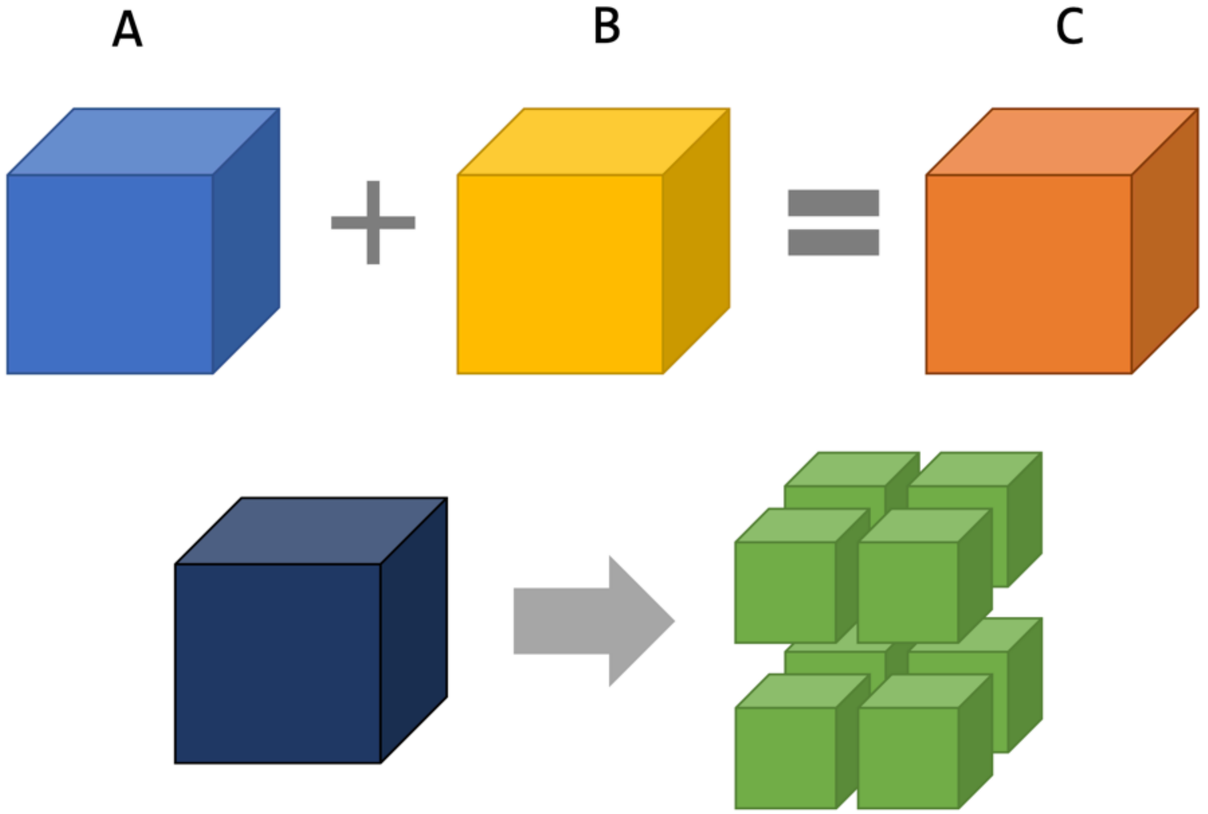


Thread blocks and grids of blocks

- Notice “blockIdx.x”, “blockDim.x”, “threadIdx.x”; we only use “.x” because we only have one dimensional blocks and grids in this case.
  - In general, you can have up to three-dimensional blocks and grids.
- There are CUDA software limits to the block size and grid dimensions:
  - Block sizes: at most 1024 threads per block, with limits of 1024, 1024, and 64 threads in the x, y, and z dimensions.
  - Grid dimensions: at most 2^31-1 = 2,147,483,647, 2^16-1 = 65,535, 2^16-1=65,535 blocks in the x, y, and z dimensions.
    - There is no limit to the total number of blocks requested.

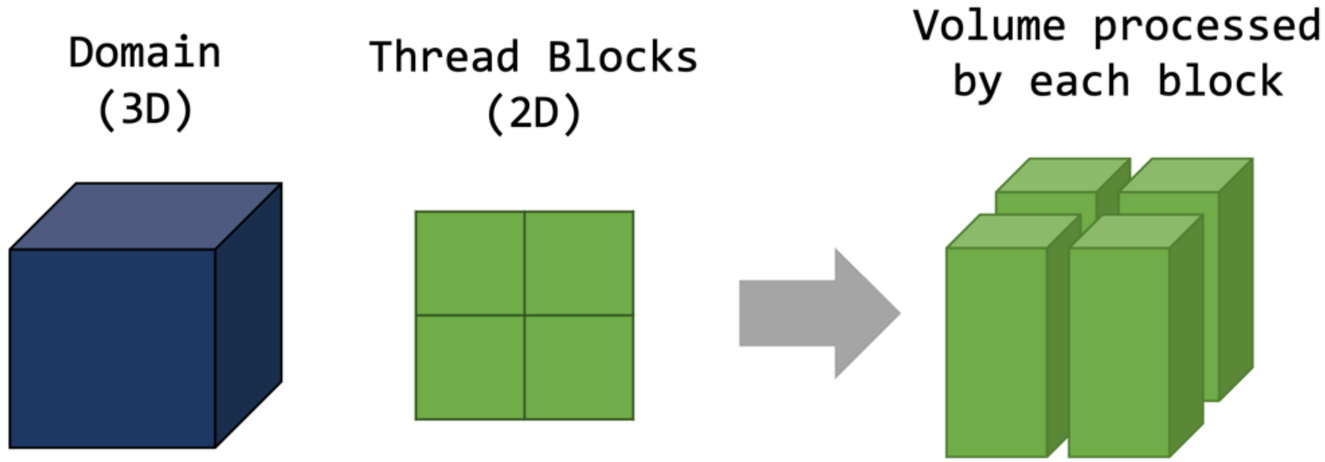
3D thread blocks and grids

- CUDA supports 1D, 2D, and 3D blocks and grids of blocks.
  - blockIdx.x, blockIdx.y, blockIdx.z
  - blockDim.x, blockDim.y, blockDim.z
  - threadIdx.x, threadIdx.y, threadIdx.z
- Cannot do 4D or beyond though.



Lower-dimensional thread blocks

- You can use thread blocks that are in a lower dimension than your data, for example, 2D thread blocks on 3D data
  - Without this, higher dimensional problems would be a problem.
- Can also be better than using dim(thread block) = dim(data) for some applications due to (shared) memory usage (more on this later)



- Example: for a 3D domain, each block can process an entire “column” in the z direction.

Efficiently utilizing memory and the memory hierarchy on GPUs

Timing a GPU kernel

- A CUDA kernel is typically non-blocking (returns as soon as it's called).
  - Can time a kernel like a standard CPU function by adding "cudaDeviceSynchronize();" to ensure the kernel has finished running.
- Another approach: CUDA events
  - Allows for detailed timing and synchronization of multiple CUDA "streams", which are sequences of kernels which can be run independently.
  - CUDA can interleave or concurrently run operations in different streams
- Later: more detailed performance metrics using the Nvidia profiler "nvprof".

Timing with CUDA events

- Create “start” and “stop” CUDA event objects
- “cudaEventRecord(event, stream_id)” records start/stop times
- “cudaEventElapsedTime” computes the wall clock time.
- Can run over multiple trials for minimum/average timings.

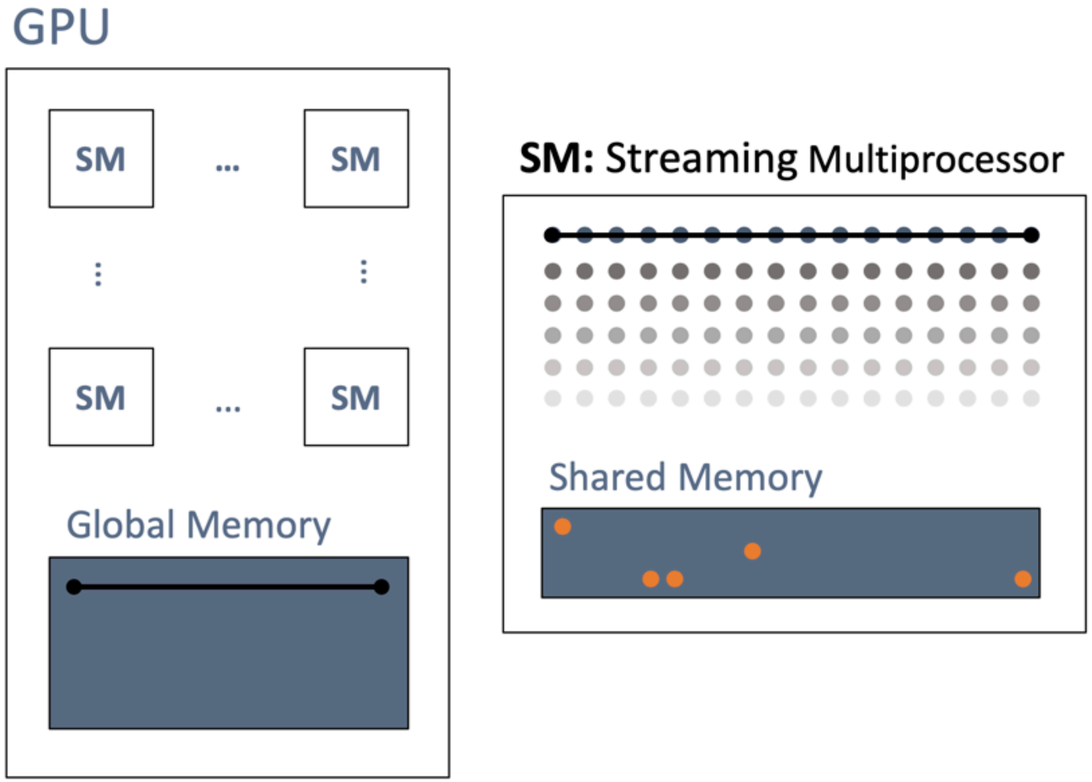
```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

matread <<< gridDims, blockDims >>> (N, d_A);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);

printf("Time to run kernel: %6.2f ms.\n", time);
```

Now that we can time a GPU kernel…

- We’ll try to determine some best practices for efficient GPU programming, especially related to memory accesses and the memory hierarchy
  - Global memory: “coalesced” memory reads
  - Shared memory management, occupancy
  - Thread-local memory: local memory, register memory

Coalesced memory reads/writes

- GPUs have phenomenal bandwidth due to parallelism in memory reads.
  - Requires “memory coalescing”: that adjacent threads read/write to/from adjacent locations in memory.
- Memory is loaded via caches from global memory (RAM on the GPU)
  - Cache lines are typically 128 bytes = (4 bytes per float) x 32.
  - A single cache hit supplies memory to an entire warp (32) of threads.



Coalesced memory write example

Consider writing to a matrix in global memory (RAM)

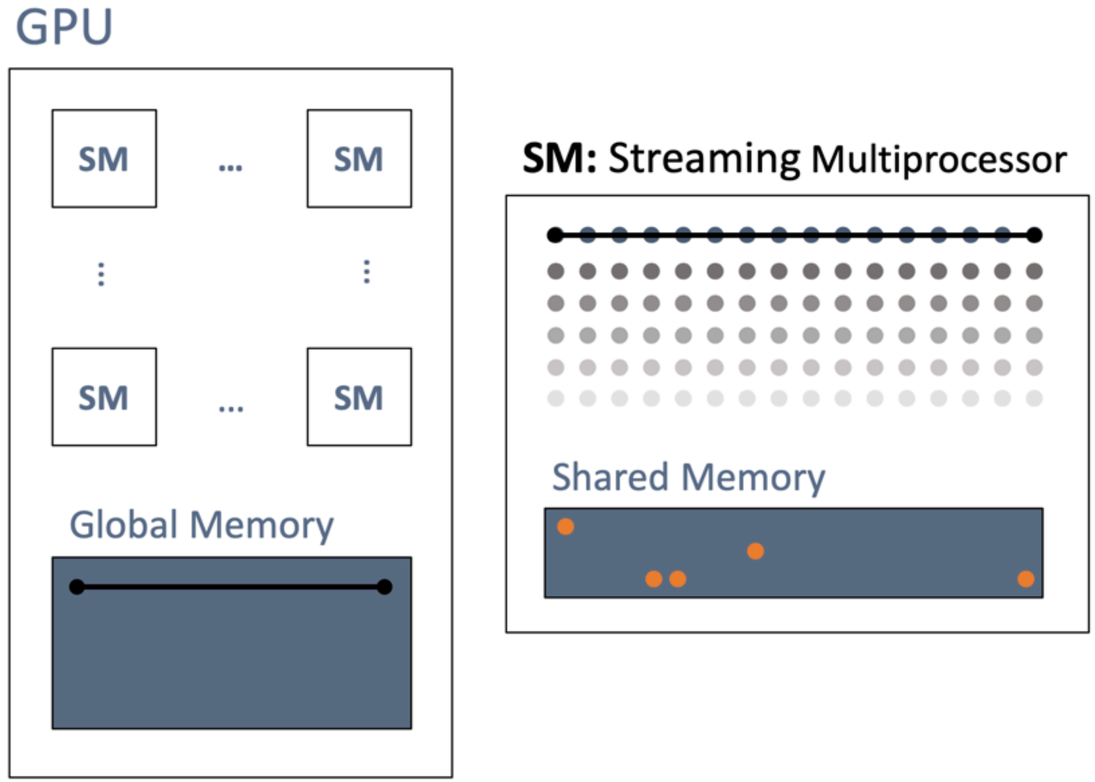
- Threads are executed “x” index first, then “y”, then “z”.
- For example, a (32, 32) block of threads executes:
  - First: “threadIdx.x = 0, …, 31” and “threadIdx.y = 0”.
  - Second: “threadIdx.x = 0, …, 31” and “threadIdx.y = 1”.
- Demo: matwrite.cu. What’s the difference in runtime?

```
__global__ void matwrite(int N, float *A){
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  int j = blockDim.y * blockIdx.y + threadIdx.y;

  if (i < N && j < N){
    A[i + j * N] = 1.f; // column major
    A[j + i * N] = 1.f; // row major
  }
}
```

Which matrix format results in coalesced memory reads?

Shared memory overview

- Shared memory is similar to L1 cache on a CPU core, shared memory access is **~100x faster** than global memory.
  - Shared memory is *local to each block*
  - Access can be random without incurring non-coalesced read penalty
- Example: can avoid non-coalesced global memory accesses using shared memory
  - Write into shared memory using coalesced global memory accesses, then access shared memory.



Communication/synchronization via shared memory

- Shared memory is also used to communicate between a block of threads
  - “__syncthreads()” is a barrier for threads within a block
  - Often used to ensure that threads are done writing to shared memory (i.e., that the shared memory is ready to use).
- Since blocks of threads must be independent, communication between blocks must be done by sharing via global memory.
  - If communication between different blocks is necessary, computations are typically broken up into two or more kernels with synchronization (e.g., “cudaDeviceSynchronize();”) in between kernel calls as necessary.

Options for allocating shared memory

- Shared memory can be allocated either in a static or dynamic fashion (similarly to stack vs heap memory).
  - Statically allocated/declared shared memory: assumes the size of the allocated memory is known at compile time.
  - Dynamically allocated shared memory: we specify the amount of memory needed at the time of kernel execution, can depend on kernel parameters.
- We will mostly use statically allocated shared memory in this course.

Shared memory example: matrix-vector product

- Computing a tall matrix-vector product “b = A*x”
  - Implement using global memory accesses only
  - Implement using shared memory accesses for “x”
  - Compare runtimes
- Demo: matvec_smem.cu

Matrix multiply with shared memory

- Shared memory is statically allocated using the “__shared__” keyword.
- Main steps:
  - Use available threads read “x” into shared memory “s_x” (using a “grid-stride loop”).
  - Each thread processes a single row of A
  - Each thread loops over the number of columns, accumulating the dot product of each row into “y[i]”.
- For more general matrix sizes, we could loop over chunks of columns of the matrix

```
#define NCOLS 64

__global__ void matvec(int N, const float *A,
                       const float *x, float *y){

  __shared__ float s_x[NCOLS];

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

  int ii = threadIdx.x;
  while (ii < NCOLS){
    s_x[ii] = x[ii];
    ii += blockDim.x;
  }

  __syncthreads();

  if (i < N){
    float val = y[i];
    for (int j = 0; j < NCOLS; ++j){
      val += A[i + j * N] * s_x[j];
    }
    y[i] = val;
  }
}
```

Dynamic shared memory on GPUs

- Specify the amount of dynamic shared memory to allocate at kernel invocation
- You can only dynamically allocate a single block of shared memory. For multiple arrays, you need to:
  - Declare pointers for each dynamic array you want
  - Assign each pointer to the appropriate memory address in the shared memory block

```
__global__ void foo(int N, float * x){
  // Note no size is specified!
  extern __shared__ int array[];
```

```
// Invoking (calling) the kernel
int size_sharedMem = n * sizeof(float); // Size in bytes
foo <<< num_blocks, threadsPerBlock, size_sharedMem >>>(n, x);
```

Register memory

- Thread-local memory, not explicitly managed.
- Generally responsible for all scalars, e.g., “int localVariable = 0”, statically allocated thread-local arrays, sometimes input variables.
- Because they have lots of threads/cores which share register memory, GPUs have fairly large amounts of register memory.
  - Can result in significant slowdowns if overused (“register spilling”).
- Can automatically count register usage using “nvcc –ptxas-options=“-v”
  - Demo…

Local memory

- Local memory: thread-local memory, but not statically sized
  - Large pool of local memory available, but it's as slow as global memory.
- If you use too many registers (e.g., "register spilling") then the GPU switches to local memory.
- Less commonly used IMO.

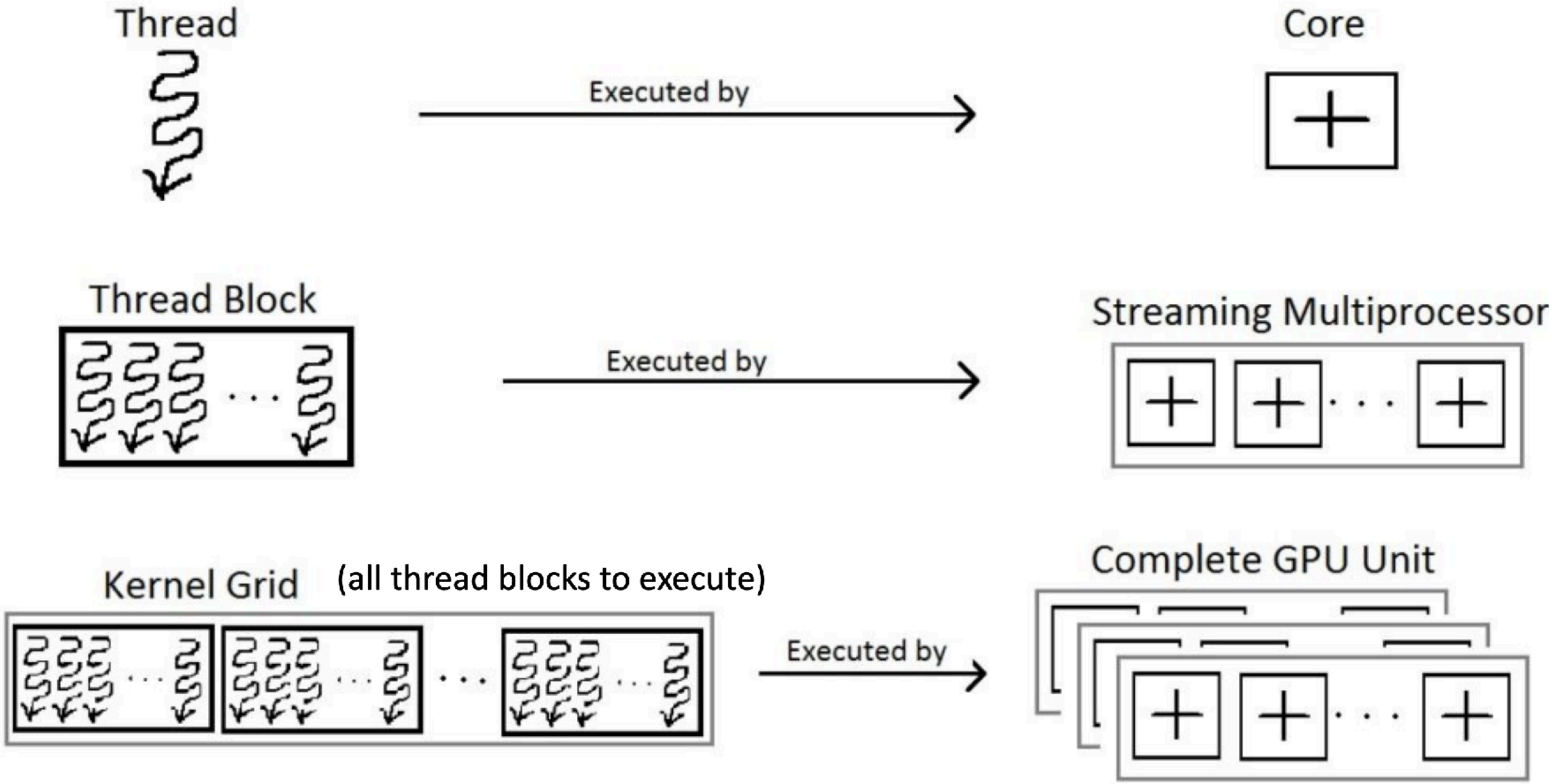
```
#define CHUNK 8
__global__ void foo(const int size)
{
  // this gets put in local memory
  float temp_local[size];

  // this gets put in register memory
  float temp_register[CHUNK];
}
```

Review of GPU memory

- Host memory: on the CPU, no size limit, PCIE transfer speed O(1-10) GB/s
- Global memory: on the GPU, size O(10) GB
  - Transfer speed: 50-100 GB/s bandwidth, 10-100x faster than host memory, need “coalesced” reads to achieve full bandwidth
- Shared memory: on GPU, local to each “streaming multiprocessor” (SM)
  - Roughly **10x** faster than coalesced global memory access
  - Overuse leads to slowdown due to *occupancy* issues (will cover later).
- Register memory: on GPU, local to each thread, up to 15x faster than shared
  - Overuse leads to slowdown due to “register spilling”

How GPU hardware executes CUDA kernels

Summary: software vs hardware terms



How do GPUs divide up work?

Recall how thread blocks/grids are mapped to SMs and warps

- “Blocks” of threads are run by streaming multiprocessors (SMs)
  - Each thread block is local to a single SM, but one SM may execute multiple thread blocks.
- Each SM can simultaneously execute some number of warps (chunks of 32 threads), the smallest unit of execution on a GPU.
  - Example: a thread block with 33 threads still requires two warps.
  - A single warp is executed by 32 CUDA “cores” (GPU hardware which are used to execute thread instructions).

A diagram of software and hardware terms

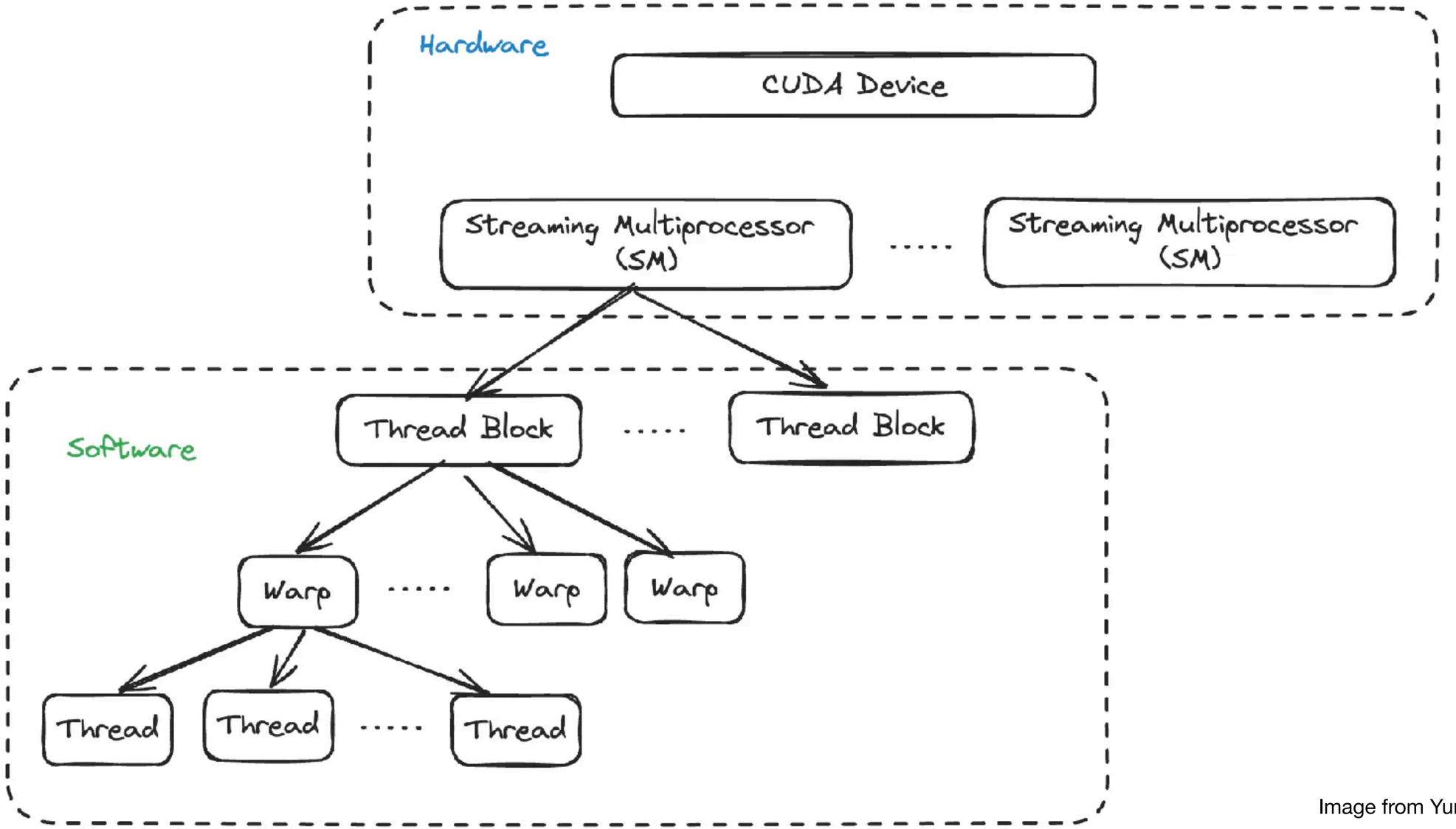
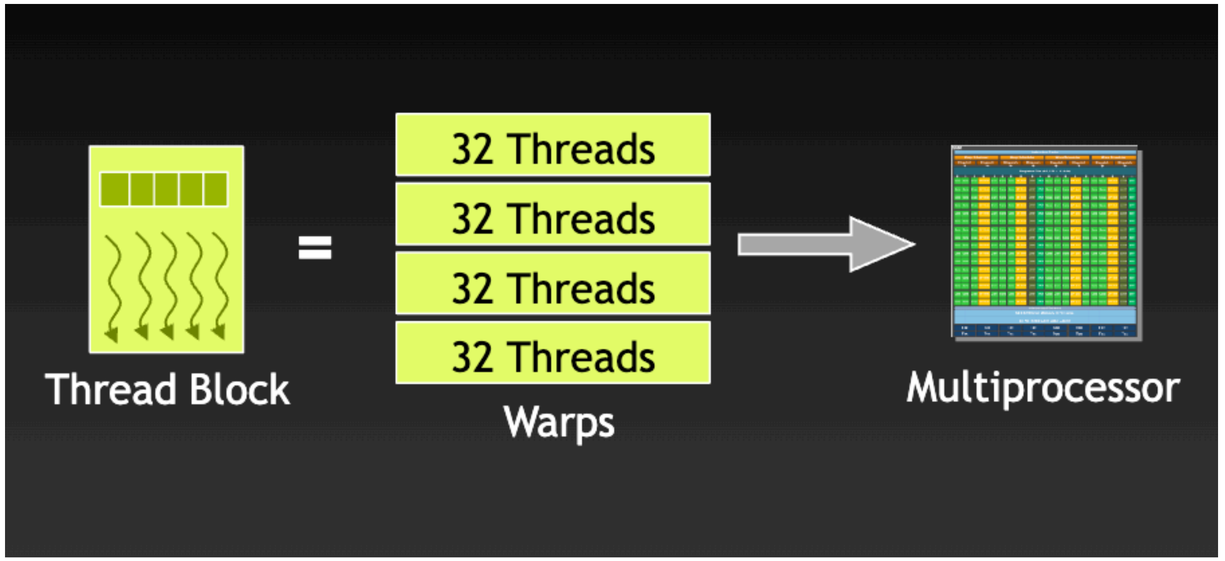


Image from Yunjiang Jiang

GPU performance depends on in-warp behavior

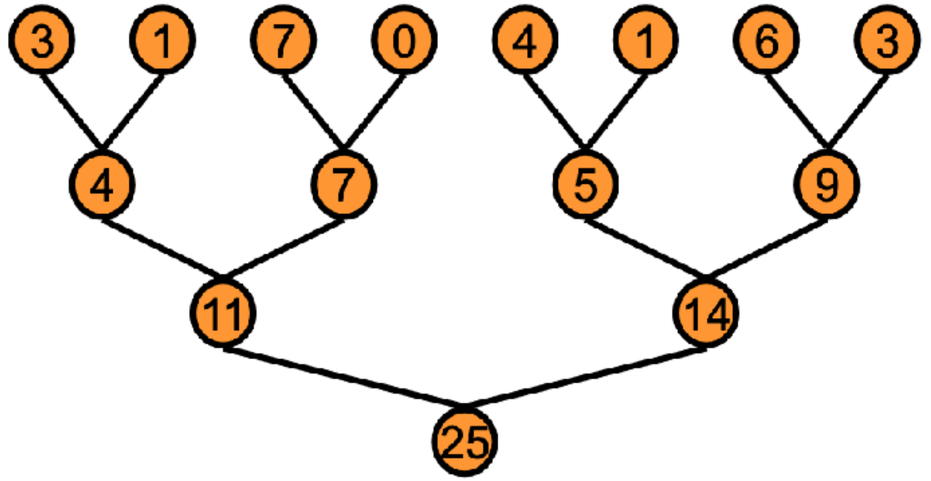
- **A single warp** is the **smallest unit of computation** a GPU, even though one warp consists of 32 threads.
  - A warp on a GPU is similar to a single thread on a CPU with 32-way vectorization.



- Synchronization/serialization may not reduce performance if they occur between two threads in different warps.
- Example: suppose if/else branching occurs for two threads. If the two threads are in the same warp, *warp divergence* occurs (the branching code is run twice).
  - Warp divergence doesn't occur **if the threads lie in different warps**.

Example: tree-based reduction

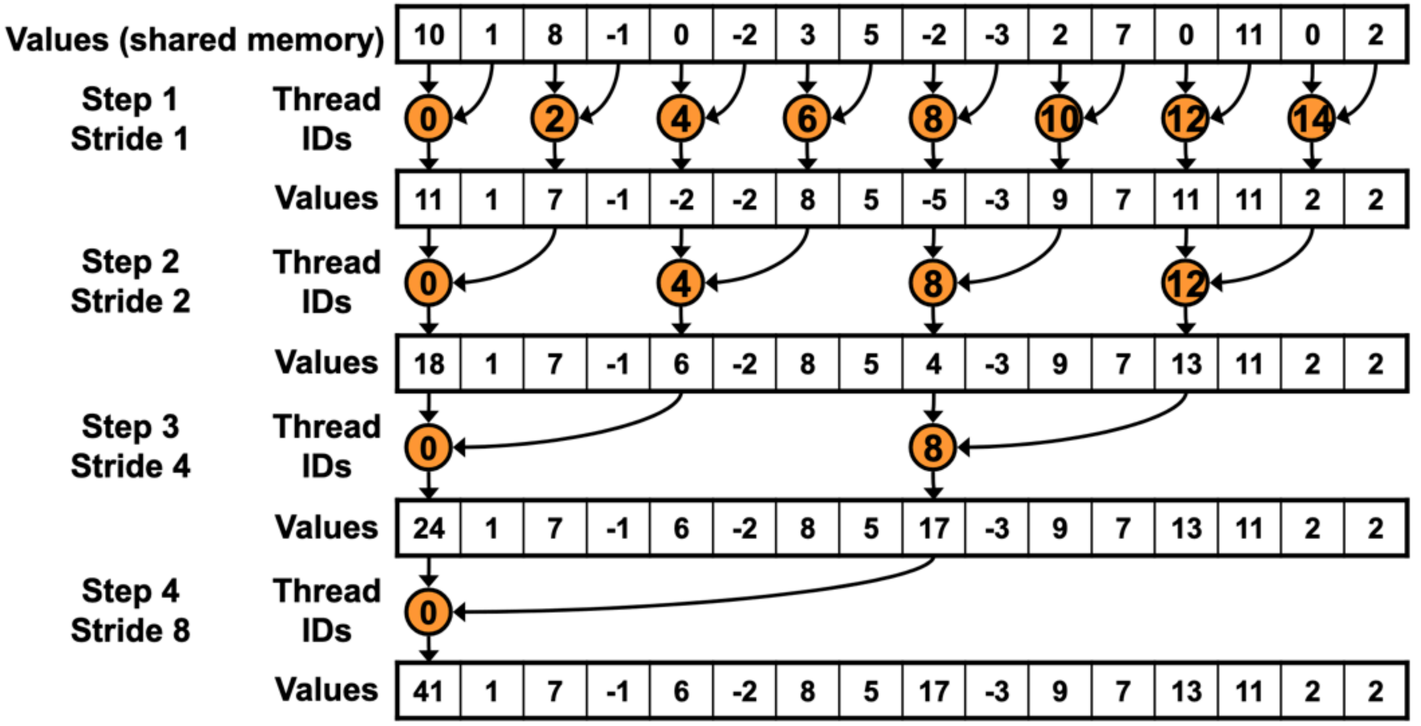
- Idea: use a tree-based reduction within each thread block.
  - Each thread block writes out the local reduction result to an intermediate array
- To compute a global reduction, can re-run reduction on the intermediate array,
  - Alternative: transfer reduced values to CPU and sum up (not too expensive if thread blocks are sufficiently large).



“Optimizing Parallel Reduction in CUDA” by Mark Harris

Tree-based reduction: version 1

- Each block reduces (sums) to a single value on thread 0
  - Step 1: thread 1 -> 0, thread 3 -> 2, thread 5 -> 4, …
  - Step 2: thread 2 -> 0, thread 6 -> 4, thread 10 -> 8
  - Step 3: thread 4 -> 0, thread 12 -> 8, …
- This is not the most efficient implementation, but will be useful in illustrating optimization steps.



- Read local block of “x” into shared memory
- Loop over strides “s” = 1, 2, 4, 8, 16, …
  - (s = 1): thread 1 -> **0**, thread 3 -> **2**, …
  - (s = 2): thread 2 -> **0**, thread 6 -> **4**, …
- If the receiving thread is a multiple of (2 * s), it receives information from thread number “thread_id + s”.

```
__global__ void partial_reduction_v1(const int N, float *x_reduced,
                                     const float *x){

  __shared__ float s_x[BLOCKSIZE];

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

  s_x[tid] = 0.f;
  if (i < N){
    s_x[tid] = x[i];
  }

  for (int s = 1; s < blockDim.x; s *= 2){
    if (tid % (2 * s) == 0){
        s_x[tid] += s_x[tid + s];
    }
    __syncthreads();
  }

  // write out once we're done reducing each block
  if (tid==0){
    x_reduced[blockIdx.x] = s_x[0];
  }
```

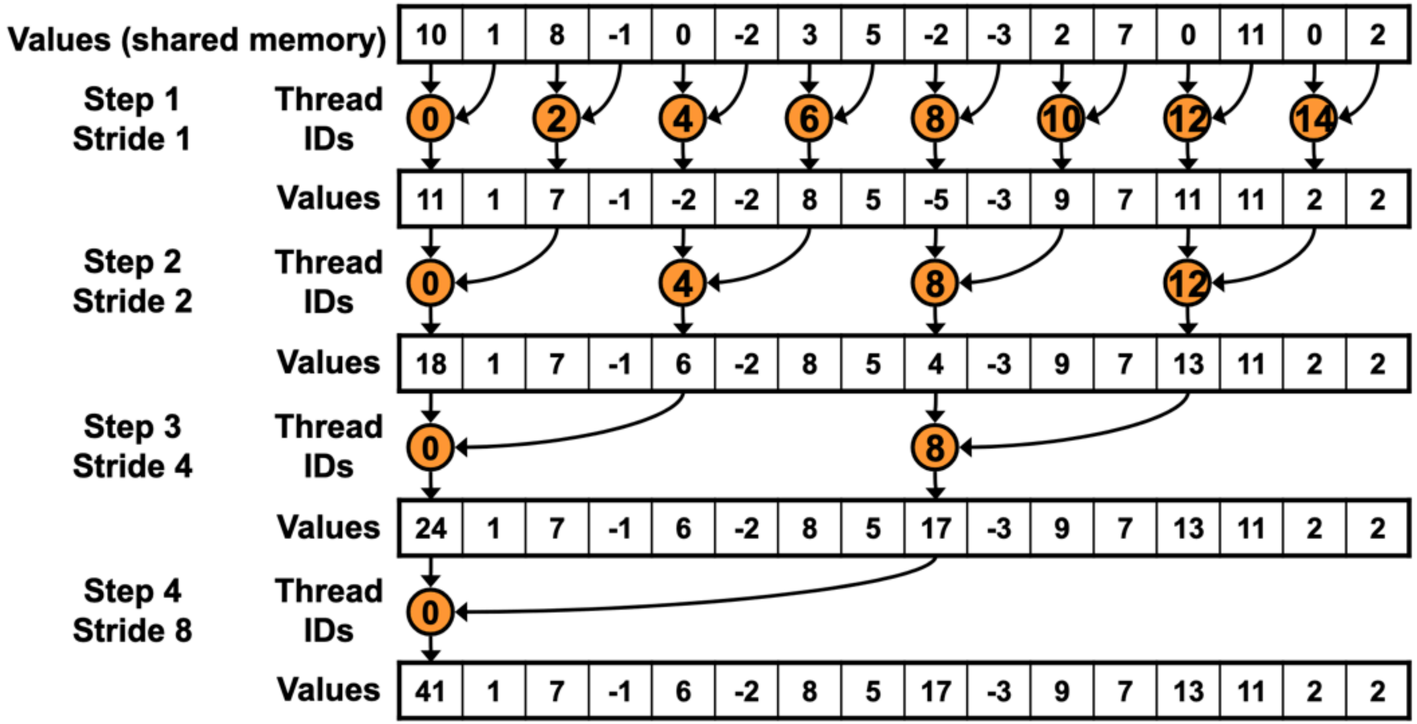
Tree-based reduction: optimizations

- We will try to optimize tree-based reduction iteratively
- Each iteration will highlight one aspect of why a GPU code might be slow
  - Code branching
  - Shared memory bank conflicts
  - GPU underutilization (e.g., idle threads)
- These three examples will also explain how the number of threads per block (i.e., block size) impacts efficiency.

Tree-based reduction: version 1 inefficiencies

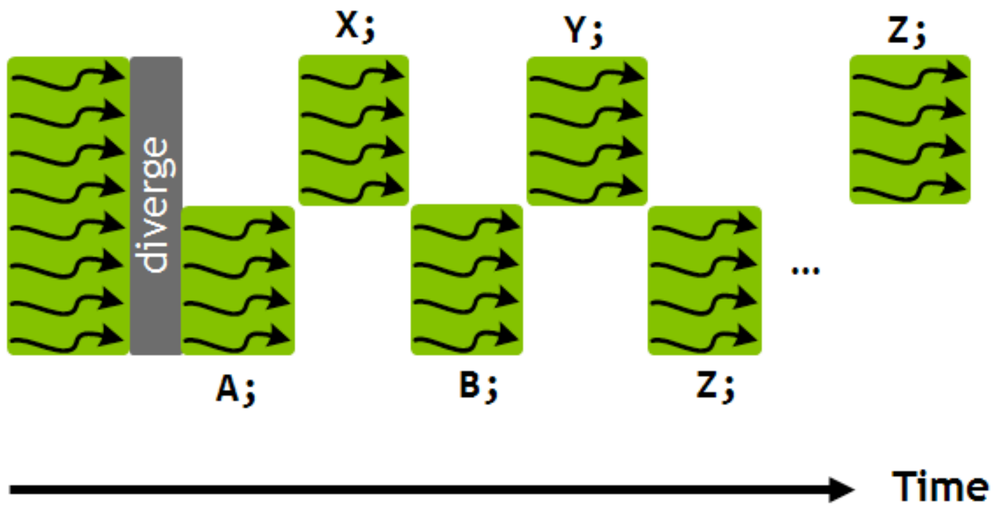
```
for (int s = 1; s < blockDim.x; s *= 2)
  if (tid % (2 * s) == 0){
```

- Issue: thread-dependent branching condition.
- Recall that GPUs treat conditional statements (e.g., if/else branches) by masking…



Recall branching and warp divergence

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



An illustration of *warp divergence*, where threads in a warp execute different branches of a CUDA kernel.

- If code branches for threads within a warp branch, it gets serialized and is no longer parallel between threads.
- **No penalty if branching happens across threads in *different warps*!**

Reduction v2

- Replace branch condition (which is highly divergent within a warp)

```
for (int s = 1; s < blockDim.x; s *= 2)
  if (tid % (2 * s) == 0){
```

with the following branching condition

```
for (int s = 1; s < blockDim.x; s *= 2){
  int index = 2 * s * tid;
  if (index < blockDim.x){
```

- Still have branching, but less divergent **within a warp**.

```
__global__ void partial_reduction_v2(const int N, float *x_reduced,
                                     const float *x){

  __shared__ float s_x[BLOCKSIZE];

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

  s_x[tid] = 0.f;
  if (i < N){
    s_x[tid] = x[i];
  }

  for (int s = 1; s < blockDim.x; s *= 2){
    int index = 2 * s * tid;
    if (index < blockDim.x){
      s_x[index] += s_x[index + s];
    }
    __syncthreads();
  }

  // write out once we're done reducing each block
  if (tid==0){
    x_reduced[blockIdx.x] = s_x[0];
  }
}
```

Why and when does this help?

```
for (int s = 1; s < blockDim.x; s *= 2)
  if (tid % (2 * s) == 0){
```

Version 1

- For reduction version 1, there is **always** warp divergence.
- Branching occurs on even and odd threads regardless of the number of threads per block.

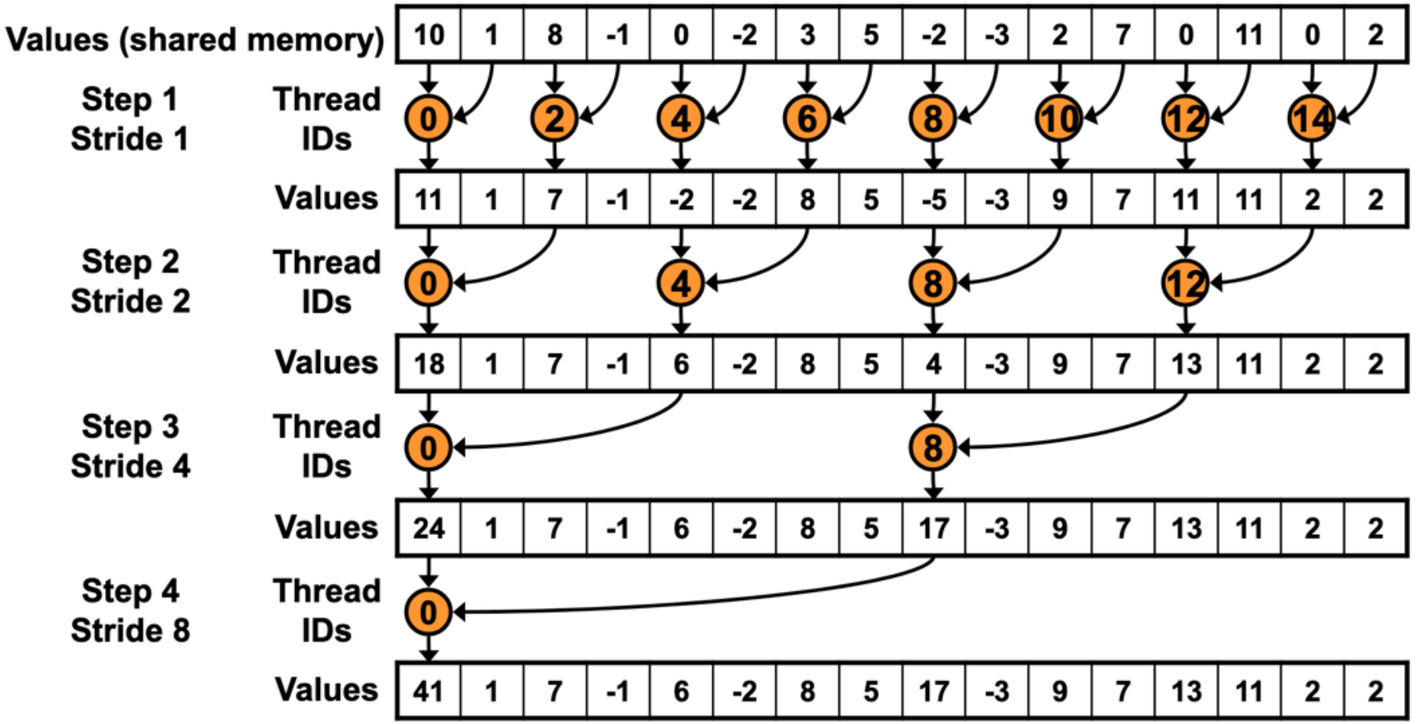
```
for (int s = 1; s < blockDim.x; s *= 2){
  int index = 2 * s * tid;
  if (index < blockDim.x){
```

Version 2

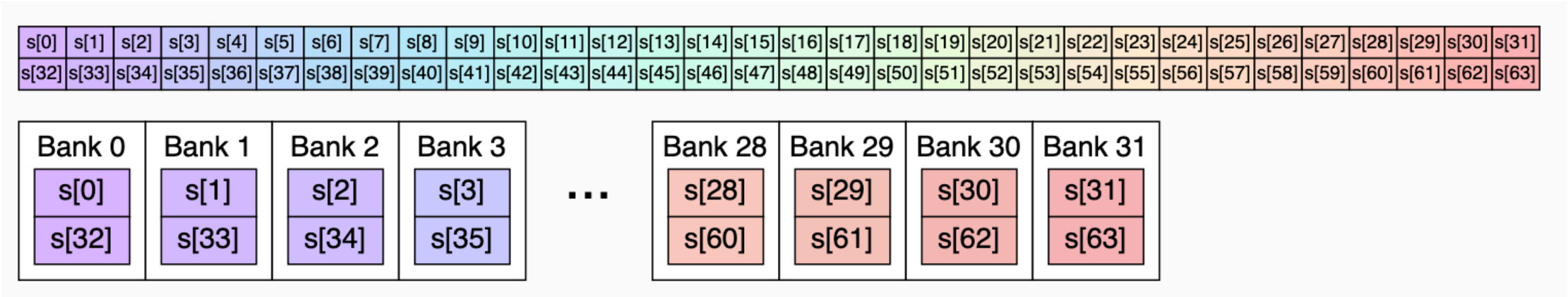
- Consider s = 1. Branching happens if “2 * tid < blockDim.x”, or if “threadIdx.x > blockDim.x / 2”
- There are “blockDim.x” threads per block, so if blockDim.x / 2 > 32, then branching happens **across different warps.**
  - **Will start to be faster for block sizes greater than or equal to 128.**

Tree-based reduction: version 2 inefficiencies

- Version 2 vs Version 1: eliminated some warp divergent-branching
- Bottleneck: bank conflicts
  - Recall false sharing in OpenMP: poor scalability when multiple threads try to access memory addresses in the same cache line
  - Similar issue (w/shared memory banks instead of cache line): GPU bank conflicts result in serialized memory accesses.



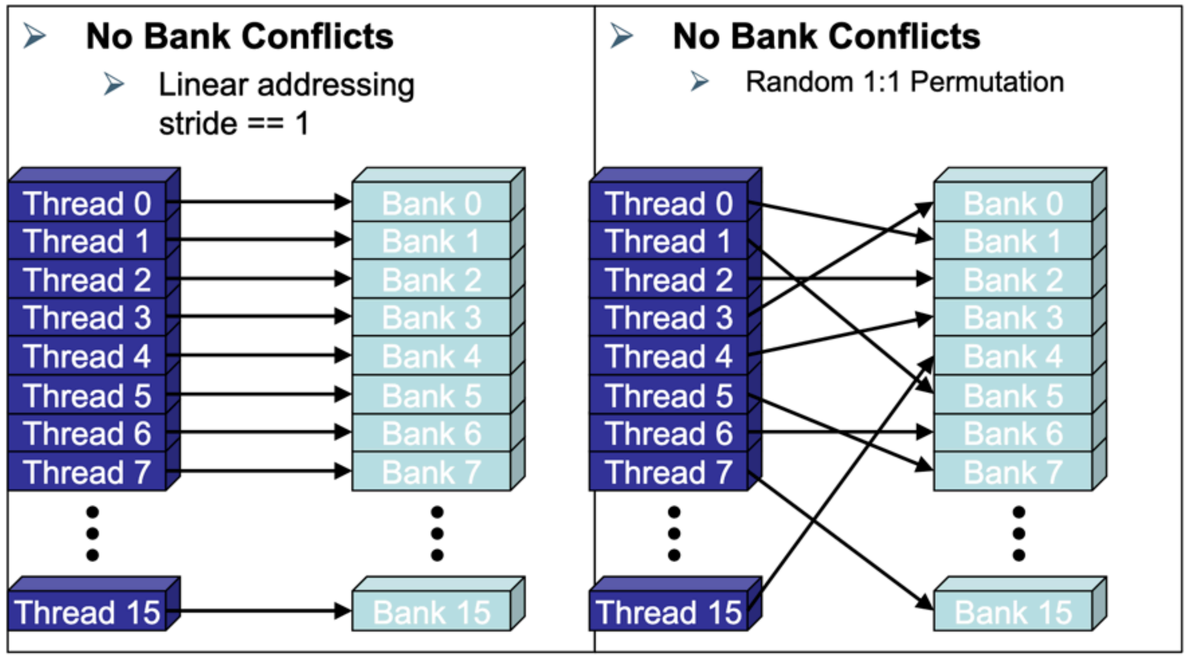
Shared memory bank conflicts

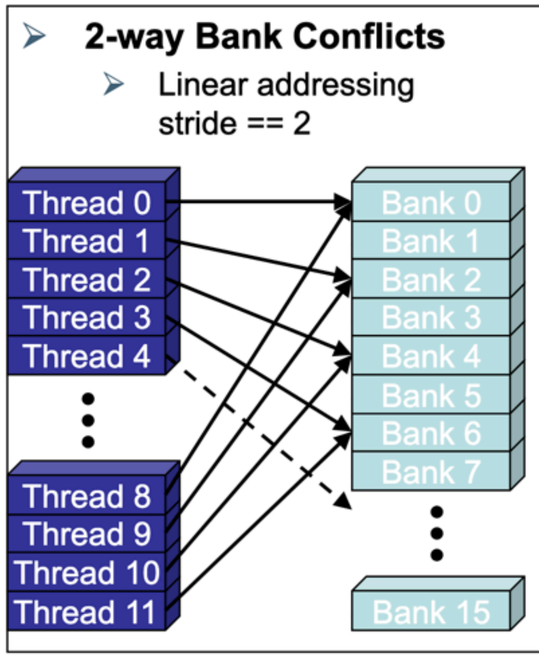


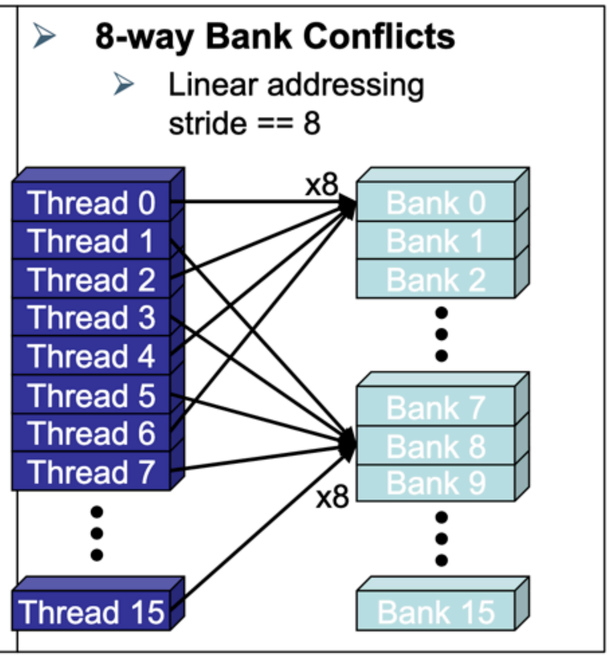
- Suppose you have “__shared__ float s[64]”. These are stored in “banks” of 4 bytes (e.g., one int or single precision float) each.
- If **two or more** threads **from the same warp** try to access the same memory bank, those memory accesses become serialized.
  - There are some exceptions: if all threads in a warp access one bank, a specialized broadcast is used instead.

https://feldmann.nyc/blog/smem-microbenchmarks

Shared memory bank conflicts, cont.





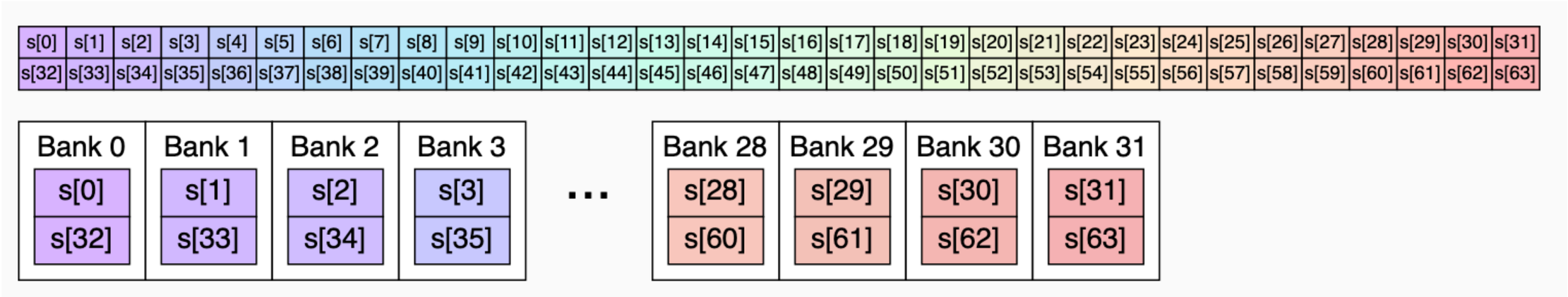


No bank conflict if each thread accesses a different location in shared memory.

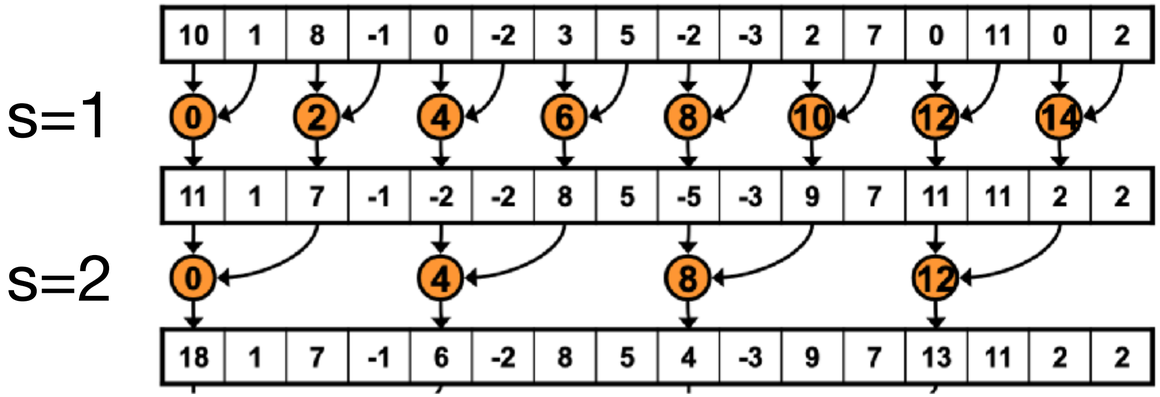
Bank conflicts occur if multiple threads in a warp try to access the same bank.

Figures from Yong Cao

Bank conflicts for reduction version 2

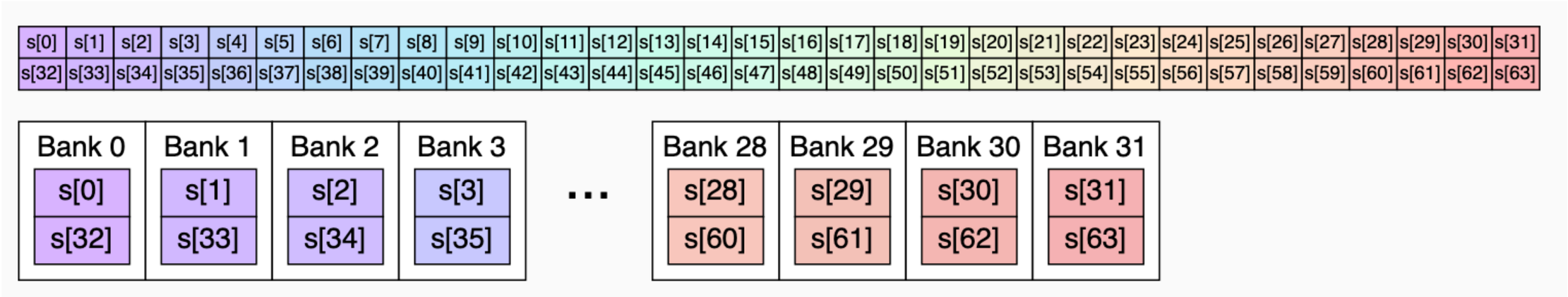


- Consider s = 1. Bank conflicts occur when:
  - thread 0 accesses s[0] in **Bank 0** and s[1] in Bank 1, thread 1 accesses s[1] and s[2], etc…
  - thread 31 accesses s[31] in Bank 31 and s[32] **in Bank 0.** Note this is within the **same warp.**
- Can fix bank conflicts via padding or by changing the memory access pattern.

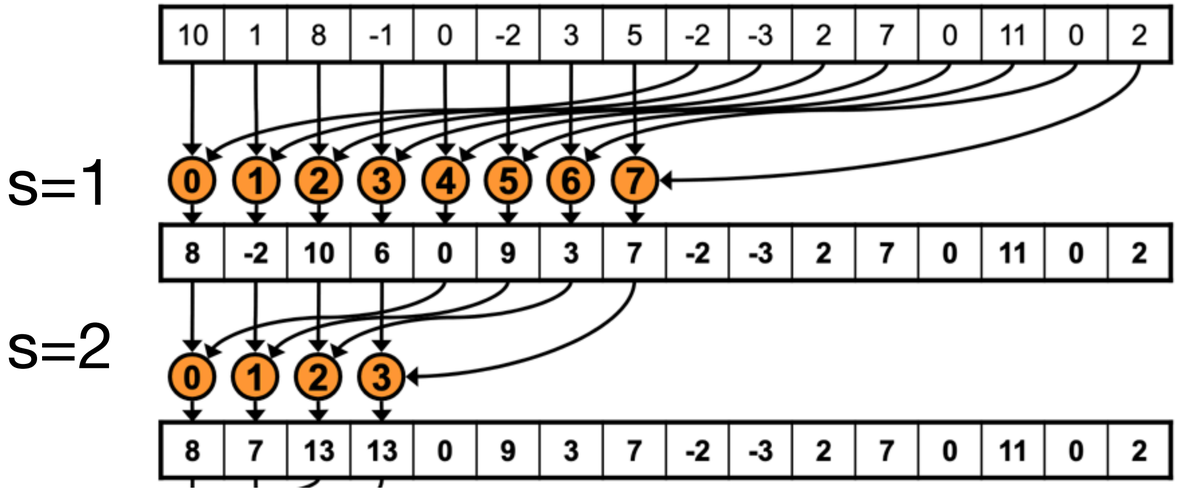


Avoiding bank conflicts for reduction

Change reduction memory access pattern



- Suppose we have **blockSize** threads per block.
- For s=1, **thread 0** and **thread (blockSize / 2)** both access Bank 0.
  - No bank conflict if **thread** (**blockSize / 2)** is in **another warp** from thread 0!
  - This holds if **thread blockSize / 2 > 32; requires** blockSize = 128 threads or larger!



New tree-based reduction access pattern which avoids bank conflicts if the block size is sufficiently large

Reduction v3

- Removes the strided for loop

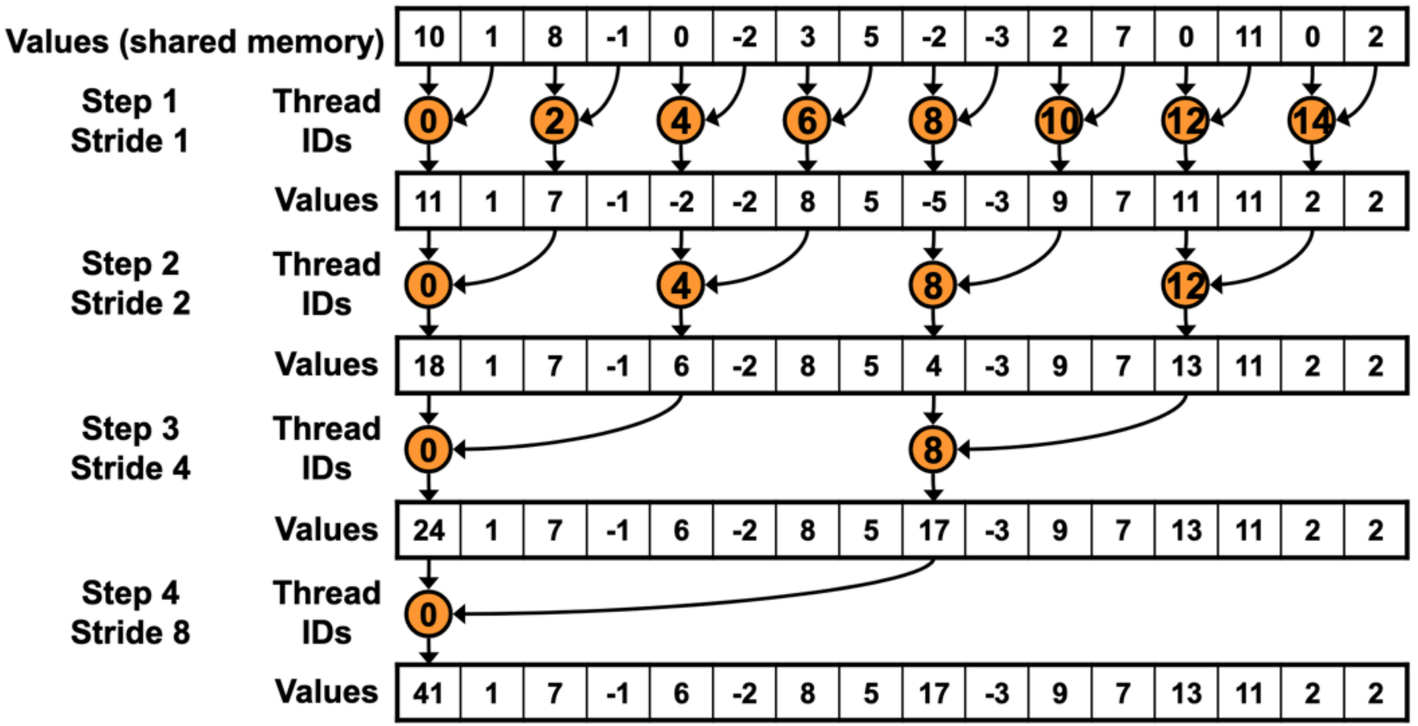
```
for (int s = 1; s < blockDim.x; s *= 2){
  int index = 2 * s * tid;
  if (index < blockDim.x){
```

- Replace with “while (number of live threads > 1)” loop
  - Initialize number of “alive” threads as the block size.
  - Accumulate upper half of threads into lower half, mark upper half of threads as dead.
- Actually a little simpler than Version 2 in my opinion.

```
__global__ void partial_reduction_v3(const int N, float *x_reduced,
                                     const float *x){

__shared__ float s_x[BLOCKSIZE];

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

// coalesced reads in
s_x[tid] = 0.f;
if (i < N){
  s_x[tid] = x[i];
}

// number of "live" threads per block
int alive = blockDim.x;

while (alive > 1){
  __syncthreads();
  alive /= 2; // update the number of live threads
  if (tid < alive){
    s_x[tid] += s_x[tid + alive];
  }
}

// write out once we're done reducing each block
if (tid==0){
  x_reduced[blockIdx.x] = s_x[0];
}
```

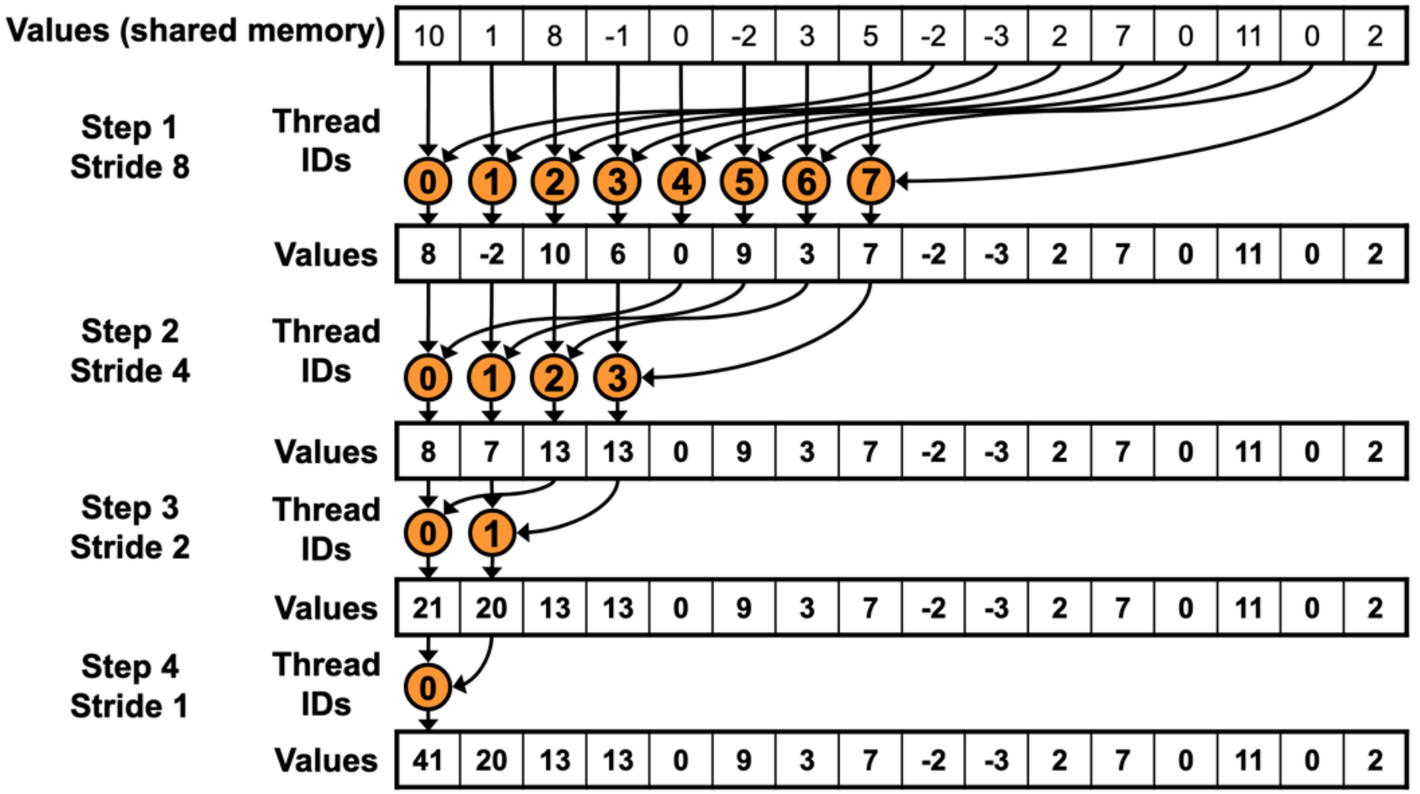
Tree-based reduction: version 3 inefficiencies

- Version 3 vs Version 2: avoid bank conflicts by changing the shared memory access pattern.
- Version 3 inefficiency: half of the threads are idle in the reduction step!
- Has to do with the way we implement the loop over levels of the reduction tree…



Tree-based reduction: version 3 inefficiencies

- Version 3 vs Version 2: avoid bank conflicts by changing the shared memory access pattern.
- Version 3 inefficiency: half of the threads are idle in the reduction step!
- Has to do with the way we implement the loop over levels of the reduction tree…



Tree-based reduction: version 3 inefficiencies

Details of the implementation

- Notice that this kernel is only active if “tid < alive”
  - “alive /= 2” halves the number of active threads from the start!
- Can perform the same amount of work using only half the number of threads (or thread blocks).
  - Each thread starts by adding two values from global memory together into shared memory.

```
// number of "live" threads per block
int alive = blockDim.x;

while (alive > 1){
  __syncthreads();
  alive /= 2; // update the number of live threads
  if (tid < alive){
    s_x[tid] += s_x[tid + alive];
  }
}
```

Reduction v4

- Launch only half the number of thread blocks

```
int numBlocksHalf = (numBlocks + 1) / 2;
partial_reduction_v4 <<< numBlocksHalf,
```

- Each block now has to cover twice the entries, so we change the way threads mapping to elements of the array
- A block of threads now maps to “2 * blockDim.x” entries of the array “x”.

```
__global__ void partial_reduction_v4(const int N, float *x_reduced,
                                     const float *x){

__shared__ float s_x[BLOCKSIZE];

const int i = (2 * blockDim.x) * blockIdx.x + threadIdx.x;
const int tid = threadIdx.x;

// coalesced reads
s_x[tid] = 0.f;
if (i < N){
  s_x[tid] = x[i] + x[i + blockDim.x];
}

// number of "live" threads per block
int alive = blockDim.x;

while (alive > 1){
  __syncthreads();
  alive /= 2;
  if (tid < alive){
    s_x[tid] += s_x[tid + alive];
  }
}
```

```
// write out once we're done reducing each block
if (tid==0){
  x_reduced[blockIdx.x] = s_x[0];
}
```

Tree-based reduction: versions 5-8

- This follows a well-known 7-step guide by Mark Harris (Nvidia) to optimizing tree-based reduction on GPUs.
- Reduction versions 1-4 are more straightforward GPU optimizations
  - Useful for understanding a GPU architecture, yields ~4-8x speedup.
- Reduction versions 5-7 involve more invasive optimizations (loop unrolling, volatile memory, warp-level synchronization via “__syncwarp()”)
  - Yields an additional ~4x speedup (for a total of 30x).

Another example of shared memory: matrix transpose

- Consider transposing a matrix (from column major to column major)
- For CPUs, resulted in non-contiguous memory accesses; for GPUs, results in non-coalesced memory writes.
- Solution: use shared memory along with blocking / tiling.
  - Coalesced writes into shared memory, then coalesced writes into global memory from shared memory

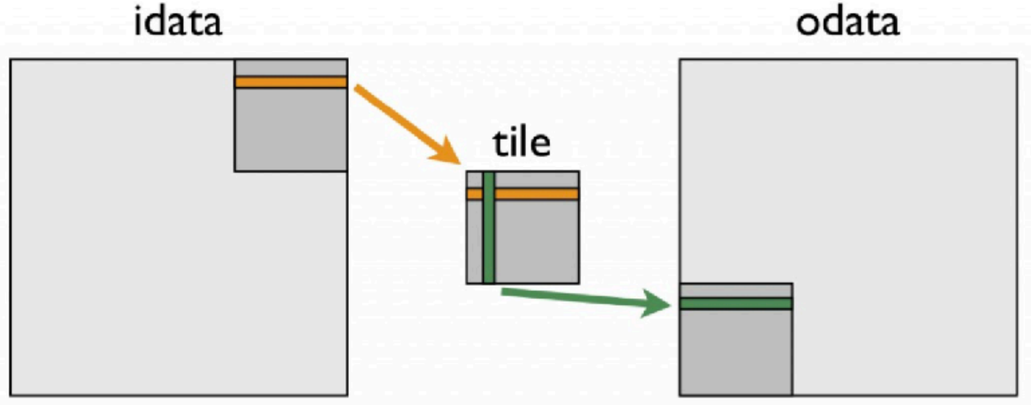


Illustration of a “tiled” matrix transpose using shared memory

GPUs and “occupancy”

GPU memory specs

Volta Streaming Multiprocessor (SM)

GV100 GPU

Each SM has:

- 64 FP32 cores
- 32 FP64 cores
- 64 KB registers
  - 16000 ints or floats
- 128 KB L1 + shared mem

Odd thing about GPUs: massive parallelism, but relatively limited memory…

GPUs cannot “max out” both parallelism and memory usage

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6	7
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	2551
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

- Max number of registers and threads per SM is very large.
  - Max 516120 registers over all threads
- However, from previous slide, we only have 64 KB (16000 ints) of register memory **in total**…?
- It’s impossible to use the maximum number of registers per thread!

GPU hardware limits

- Block size = number of threads per block
  - There are both hardware and software limits to the block size (usually 1024 threads per block)
- Number of blocks = grid dimension
  - There is a hard limit: 65535 (2^16-1) per dimension
- If you have more blocks than can be run on SMs, they are queued up and executed in any order.
- These limitations do not typically impact performance…

GPUs limit SM utilization based on shared resources

- Each SM can execute multiple thread blocks. **However**, each SM also has a limited pool of threads, as well as a limited pool of shared memory and register memory.
- An SM will end up running fewer thread blocks if each thread block uses a large amount of resources:
  - Example: many threads = fewer thread blocks (usually OK)
  - Example: each block uses too much shared/register memory (not as OK).
    - Unused warps = potentially inefficient GPU usage
- SM utilization efficiency measured by the concept of **“occupancy”**

Occupancy calculator

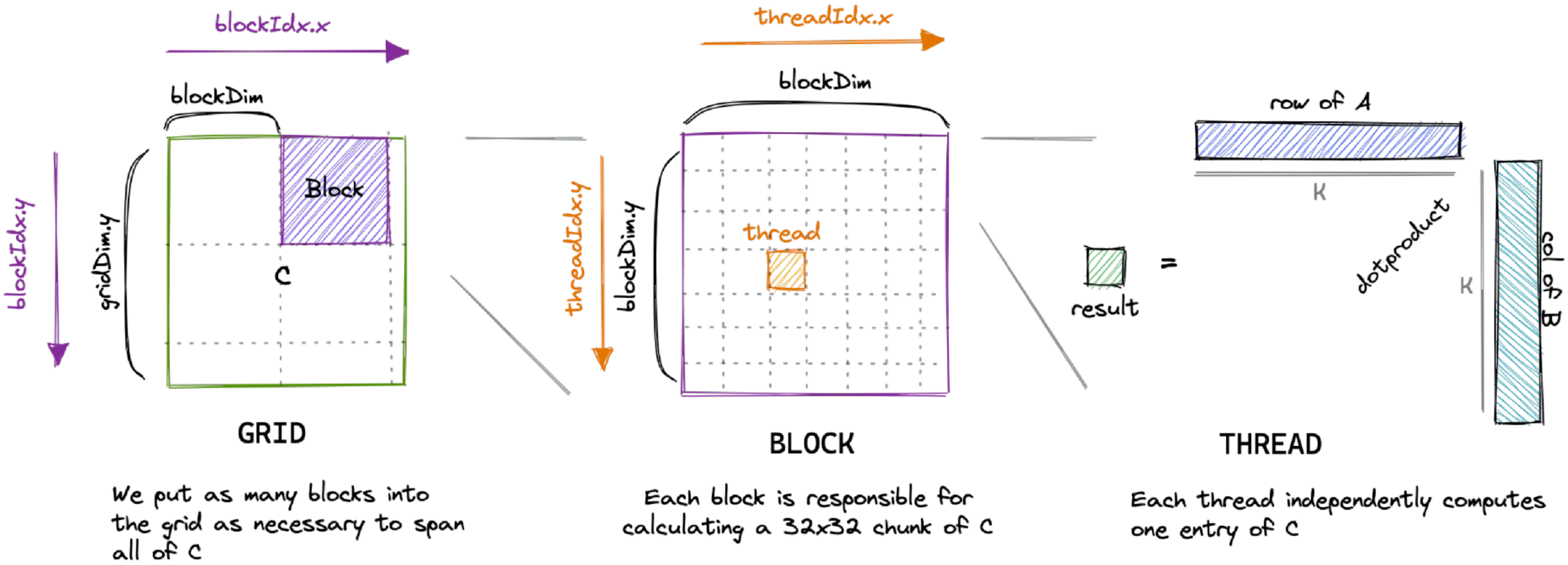
- How efficiently GPU SMs are utilized is measured by “occupancy”
- An online example: https://xmartlabs.github.io/cuda-calculator/
  - NOTSx GPUs use compute capability 7.0 (can check via “nvidia-smi --query-gpu=compute_cap —format=csv”), so these calculators are still OK.
  - CUDA suggests Nsight Compute’s built-in occupancy calculator for any GPUs with compute capability > 8.6.
- Can use "nvcc -arch=sm_37 --ptxas-options=-v code_to_compile.cu” to analyze register usage per thread and shared memory usage per block
  - These values can be plugged into an occupancy calculator (Excel, online)

Matrix multiplication on GPUs

Recap of GPU concepts

- Memory hierarchy on GPUs: host << global << shared < register
  - Must use memory coalescing for fast global memory accesses
- Occupancy: GPUs have massive parallelism but are constrained by shared resources (shared and register memory).
  - GPUs schedule thread blocks to run on streaming multiprocessors (SMs).
  - Using too much shared or register memory per block/thread leads to under-utilization (lower occupancy)

Version 1: naive matmul



This will be our reference implementation that we analyze and improve.

Version 1: naive matmul

- Uses 2D thread blocks and grids
  - Block size is a user-defined parameter, but doesn't matter as much for this version.
- Each thread computes a dot product between row of A and column of B.
- Notice the global memory accesses of A, B, and C - are these reads and writes coalesced?

```
__global__ void matmul(int N, const float *A,
                       const float *B, float *C) {

  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < N && j < N) {
    float val = 0.f;
    for (int k = 0; k < N; ++k) {
      val += A[k + i * N] * B[j + k * N];
    }
    C[j + i * N] += val;
  }
}
```

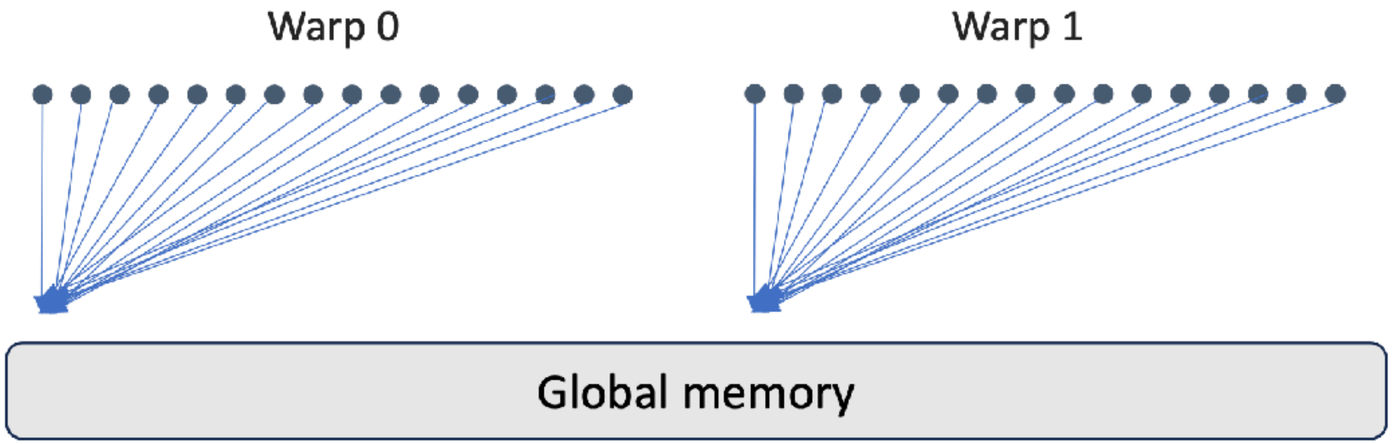
Which memory accesses are coalesced?

- Memory accesses to A, C are **non-coalesced** and will be slow.
  - Strided memory access with respect to thread index “i”.
- However, it turns out that memory accesses for B are **broadcasted** (non-coalesced, but are still fast) if the block size is large enough!
  - Must have a block size of at least (32, 32) so that each index “j” corresponds to at least one warp.

```
__global__ void matmul(int N, const float *A,
                       const float *B, float *C) {

  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  const int j = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < N && j < N) {
    float val = 0.f;
    for (int k = 0; k < N; ++k) {
      val += A[k + i * N] * B[j + k * N];
    }
    C[j + i * N] += val;
  }
}
```

Global memory broadcasts are not slow

- Non-coalesced memory accesses (contiguous threads do not access contiguous memory) are typically slow (serialized).
- An exception is if every thread in a warp accesses **the same global memory location.**



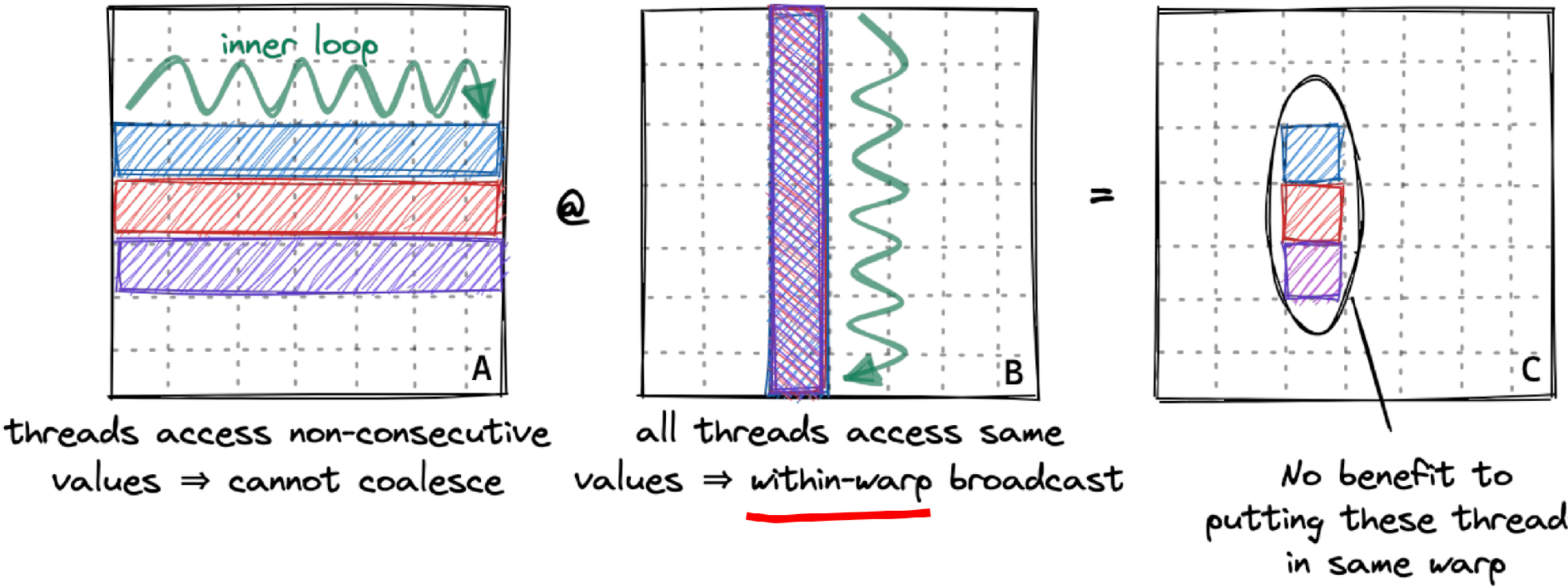
Robert_Crovella Moderator Oct 24 '18

Memory accesses from separate warps (i.e. from threads not in the same warp) are never coalesced. Memory accesses from the same warp but emanating from different instructions in the instruction stream are never coalesced. However in each case you may still get some benefit from the cache, compared to going to global memory for all accesses.

If threads in the same warp access (read) the same location in the same instruction/clock cycle, the memory subsystem will use a “broadcast” mechanism so that only one read of that location is required, and all threads using that data will receive it in the same transaction. This will not result in additional transactions to service the request from multiple threads in this scenario.

https://forums.developer.nvidia.com/t/accessing-same-global-memory-address-within-warps/66574

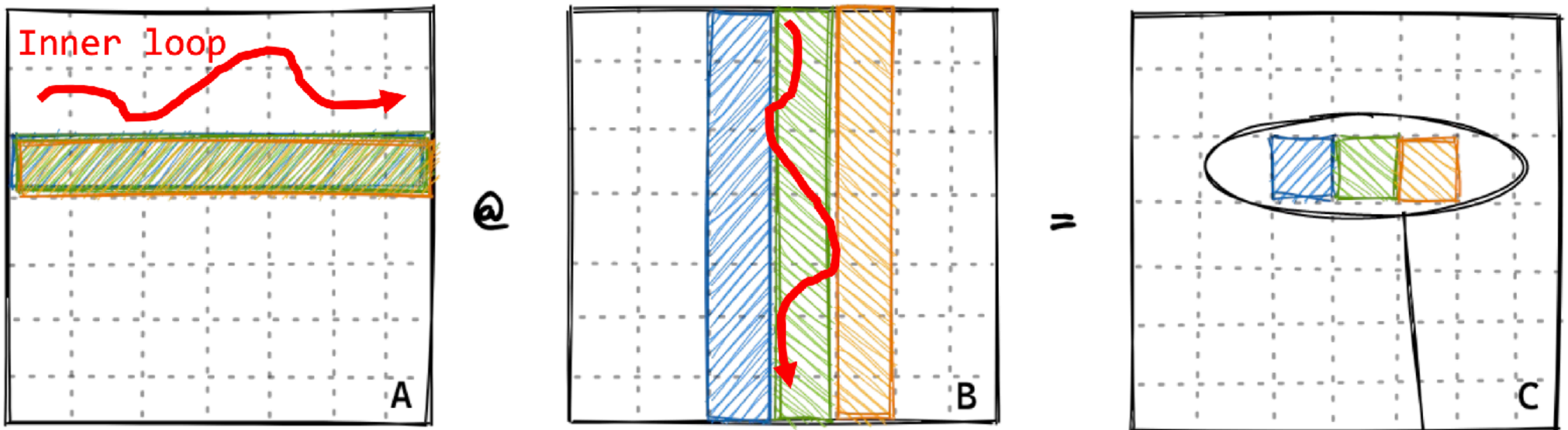
Fixing version 1 (naive implementation)



Can avoid non-coalesced memory reads by changing **thread mapping**.

Version 2 (memory coalescing)

Coalescing kernel:



all threads access same values ⇒ within-warp broadcast

threads access consecutive values ⇒ can coalesce

Make sure these threads end up in same warp to exploit coalescing

Equivalent to parallelizing over scaled column contributions

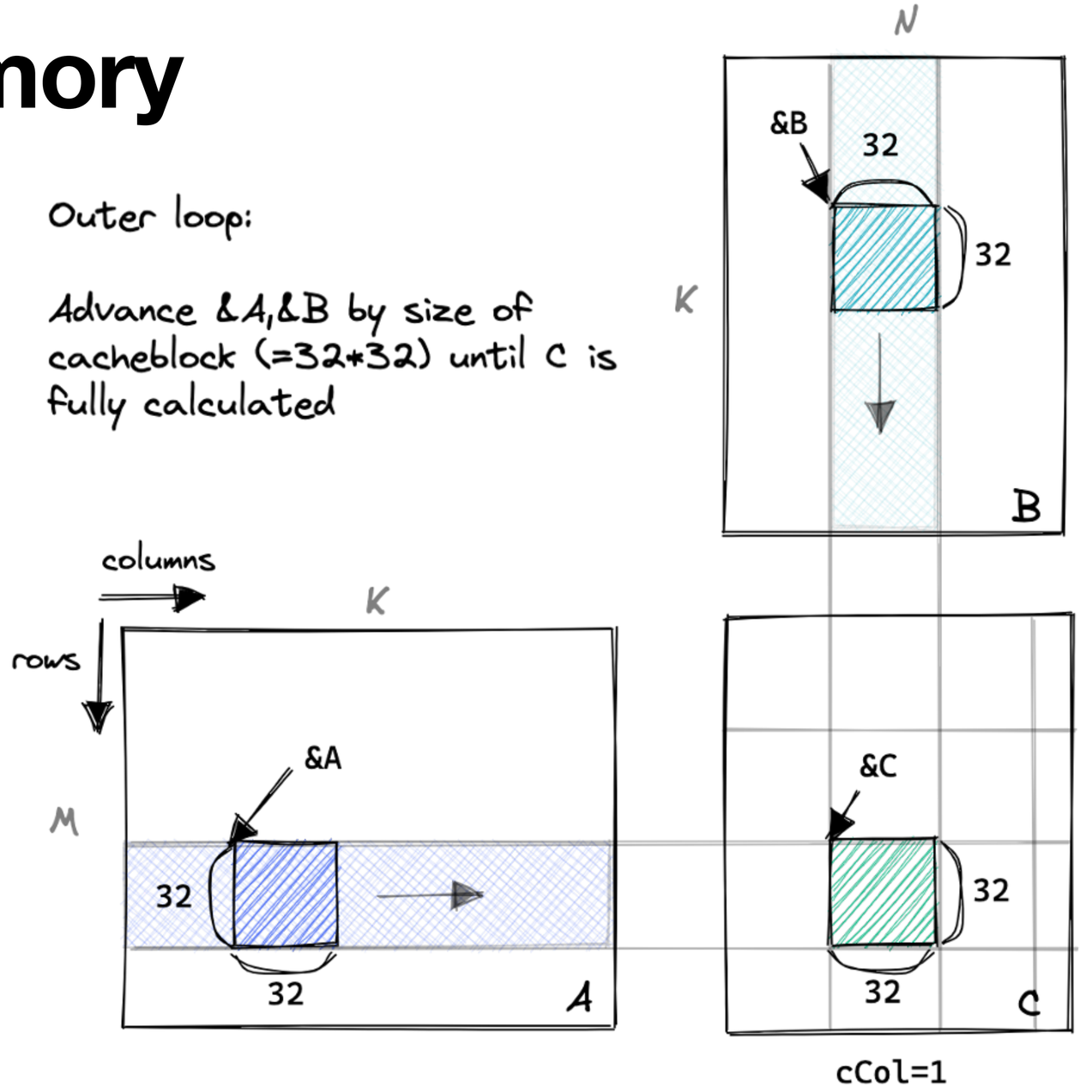
Version 2: naive matmul

- Flip the order of the row and column indices so that “i” maps to “threadIdx.y” and “j” maps to “threadIdx.x”.
- Memory accesses for B, C are now coalesced!
- Memory accesses for A are broadcasted if the block size is at least (32, 32)

```
__global__ void matmul(int N, const float *A,
                       const float *B, float *C) {

  const int j = blockIdx.x * blockDim.x + threadIdx.x;
  const int i = blockIdx.y * blockDim.y + threadIdx.y;

  if (i < N && j < N) {
    float val = 0.f;
    for (int k = 0; k < N; ++k) {
      val += A[k + i * N] * B[j + k * N];
    }
    C[j + i * N] += val;
  }
}
```

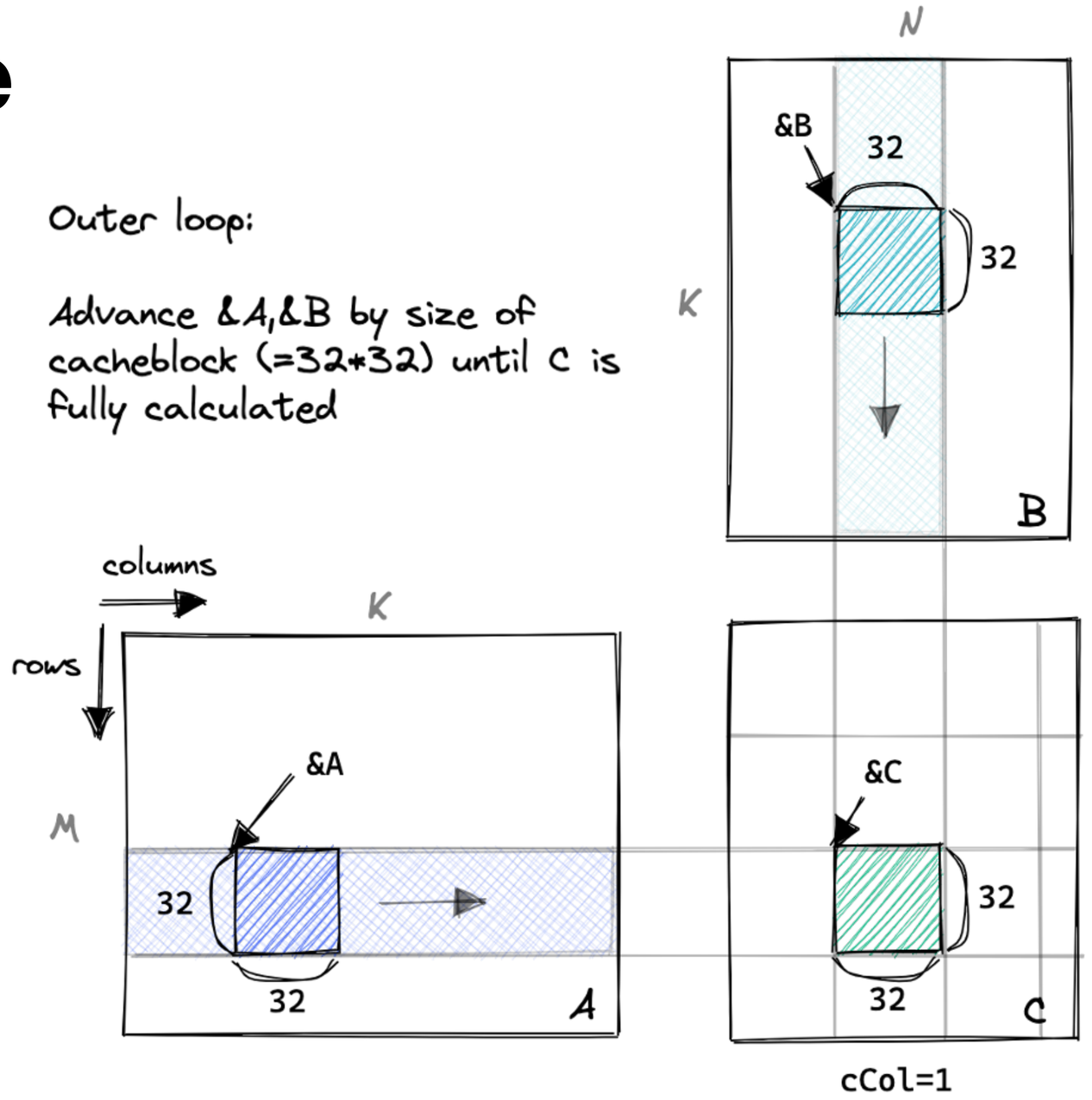
Version 3: shared memory

- Version 2 has coalesced global memory reads, but still has redundant memory reads.
  - Similar to the serial case - version 2 has low arithmetic intensity
- Version 3 is similar to cache-blocking, but using explicitly managed shared memory instead.
- Analysis of arithmetic intensity for serial blocking still holds (but now we don't have to guess cache size).



Version 3 pseudocode

- Loop through blocks in the “k” direction (instead of entries)
- For each block, read a block of A, B in global memory into shared memory arrays s_A, s_B
- Each thread computes a single entry of the product of s_A, s_B by looping over the “k” block dimension.
- Block size should be (32, 32) or larger for in-warp broadcasts and coalesced memory reads.



Version 3: implementation details

- The implementation of cache-blocking is part of Homework 4
- Some tricks that I used:
  - I used the same thread mapping from Version 2 to exploit coalescing when reading global memory into shared memory.
  - I statically “#define BLOCKSIZE …” so that s_A and s_B can be allocated as static shared memory arrays of length “BLOCKSIZE * BLOCKSIZE”.
  - I locally index into “s_A, s_B” using “threadIdx.x, threadIdx.y”
  - You have to keep track of the starting location of current block of A, B. This can be done using indexing, but I used pointer arithmetic to just increment the memory address that “float * A, B” are pointing to after each block read.

Additional optimizations

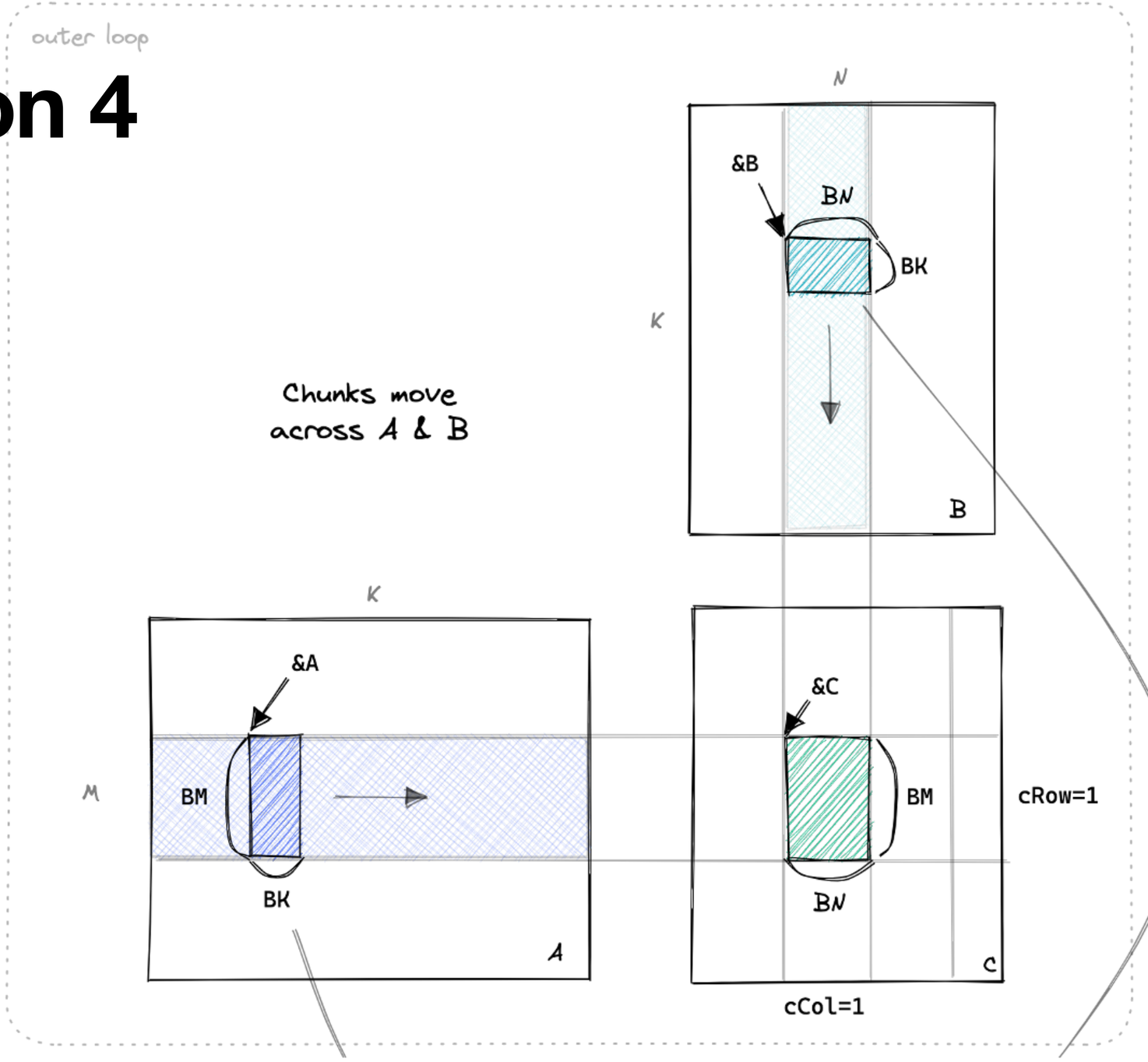
- Version 3 just uses the basics of what we've taught about GPU memory
  - Fast global memory reads/writes via coalesced memory accesses
  - Replace repeated global memory accesses with faster shared memory accesses
- S. Boehm goes through an additional 5 steps of optimization to create Versions 4-8 of matrix multiply.
  - We'll go through a more detailed overview of Version 4
  - We'll skim through some optimizations used for Versions 5-8

Overview of Version 4

- Version 3 (~3000 GFLOPS/sec) provides a speedup of about 2.5x over Version 1 (~300 GFLOPS/sec).
- In comparison, Boehm's "Version 4" achieves ~8500 GFLOPS/sec.
  - Profiling reveals that shared memory reads take up most of the time.
  - Idea: if shared memory reads are the bottleneck, reducing shared memory usage should speed things up more.
- Solution: **each thread processes multiple elements** of A*B, use **register memory** instead of shared memory.

Overview of Version 4

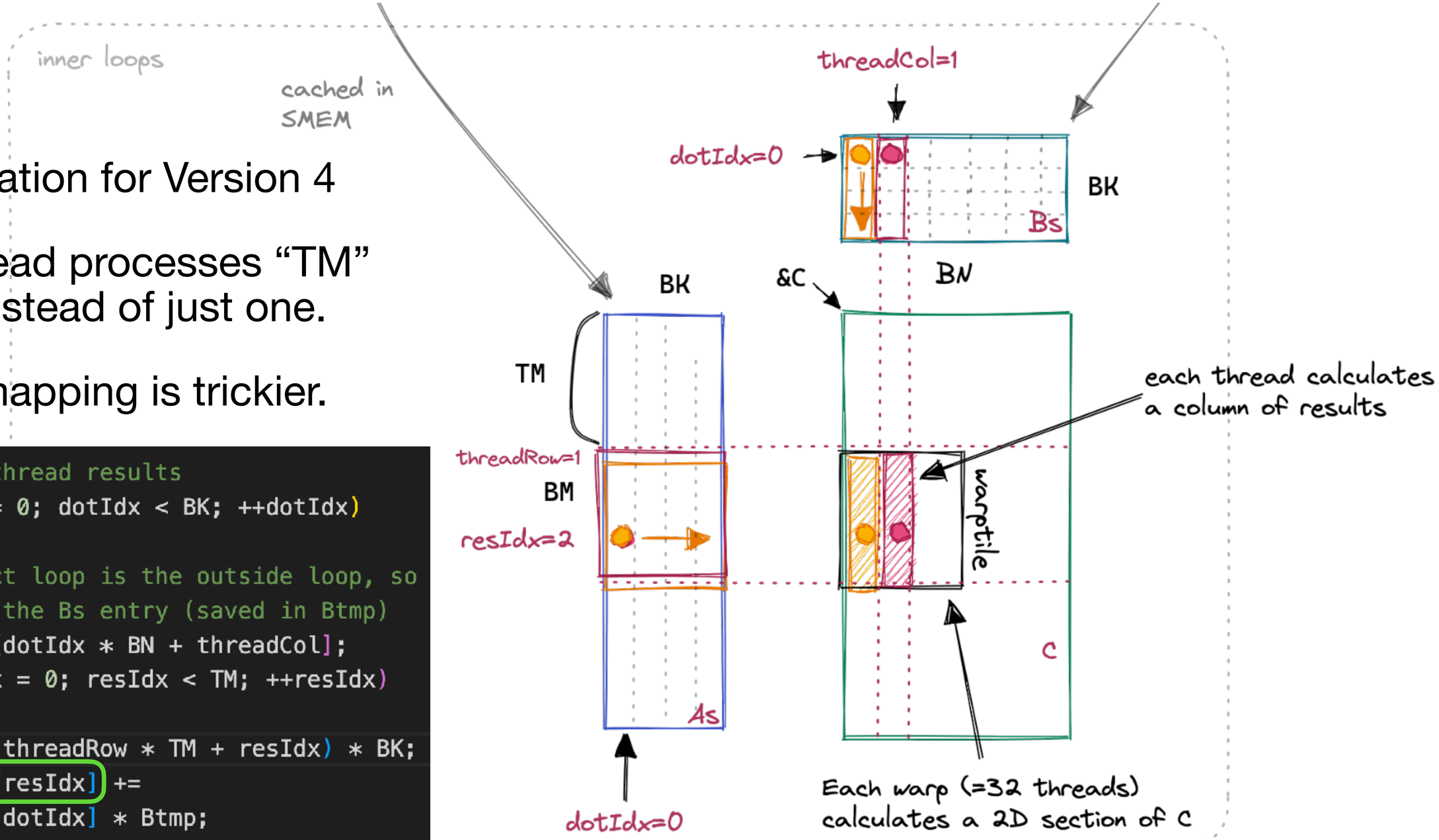
- Version 4 still does cache-blocking/tiling, but doesn't use square blocks anymore
- Each rectangular block of A, B is stored in shared memory.
  - Inner iteration over blocks uses thread-local register memory, however.
  - Each thread processes multiple elements to reduce



- Inner iteration for Version 4
- Each thread processes “TM” entries instead of just one.
- Thread mapping is trickier.

```
// calculate per-thread results
for (uint dotIdx = 0; dotIdx < BK; ++dotIdx)
{
  // the dotproduct loop is the outside loop, so
  // we can reuse the Bs entry (saved in Btmp)
  float Btmp = Bs[dotIdx * BN + threadCol];
  for (uint resIdx = 0; resIdx < TM; ++resIdx)
  {
    int offset = (threadRow * TM + resIdx) * BK;
    threadResults[resIdx] +=
      As[offset + dotIdx] * Btmp;
  }
}
```

Accumulate into thread-local register memory instead of shared memory.



Overview of Versions 5-8

- Version 4 has each thread process a column of C (“1D block tiling”)
- Version 5 has each thread process a **block** of C, exploiting some memory reuse that you get in 2D block tiling that you don’t see in 1D block tiling (about 1/2 the smem accesses).
  - Version 5 achieves ~16000 GFLOPS/sec vs ~8500 GFLOPS/sec (Version 4)
- Version 6 (**vectorizing** global, shared memory accesses): ~18000 GFLOPS/sec
- Version 7 (autotuning computational parameters): ~20000 GFLOPS/sec
- Version 8 (exploiting warp structure, hardware specific-ish): ~22000 GFLOPS/sec
- **Still the overall winner: Nvidia’s cuBLAS library: ~24000 GFLOPS/sec**

Miscellaneous topics

Miscellaneous topics

- Useful tricks: grid-stride loops, template kernel parameters
- Small optimizations: loop unrolling, vectorization using packed float4, int4
- Single vs double precision operations
- Performance analysis using Godbolt and CUDA profiler nvprof
- Alternatives to CUDA for portable GPU programming

Grid stride loops

```
__global__ void add(const int N, const float *x, float *y){

  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int ii = i; ii < N; ii += blockDim.x * gridDim.x){
    y[ii] += x[ii];
  }
}
```

- Can avoid invalid kernel arguments (e.g., too many threads) by using *grid-stride loops (*loop over available threads until all elements are processed).
- Allows “blockSize” and “numBlocks” to be chosen arbitrarily.

Loop unrolling

```
#pragma unroll
for (int i = 0; i < 4; ++i)
  y[i] = y[i] + x[i];
```

- Common compiler optimization: unrolling a statically sized for loop
- After loop unrolling, the following code becomes:

```
y[0] = y[0] + x[0];
y[1] = y[1] + x[1];
y[2] = y[2] + x[2];
y[3] = y[3] + x[3];
```

Why loop unrolling might help

```
y[0] = y[0] + x[0];
y[1] = y[1] + x[1];
y[2] = y[2] + x[2];
y[3] = y[3] + x[3];
```

- Why is loop unrolling an optimization?
  - Unrolling tends to improve Instruction Level Parallelism (ILP).
  - Unrolled code can be further optimized by the compiler
- Downside: unrolling a large loop can lead to higher register usage.

Vectorization using int4, float4

- In addition to coalesced memory reads, GPUs are capable of SIMD memory reads and vectorized operations per-thread.
  - 128-bit vector lanes = can vectorize load/stores (or operations) of 4 integer or single-precision float values.
  - Can also vectorize load/stores with 2 double values, or any struct whose size is a power of 2 (see below for an example).

```
struct Foo {int a, int b, double c}; // 16 bytes in size
Foo *x, *y;
…
x[i]=y[i];
```

add_vectors example using float4

```
const float4 x1 = x[i];
float4 y1 = y[i];

y1.x += x1.x;
y1.y += x1.y;
y1.z += x1.z;
y1.w += x1.w;

y[i] = y1;
```

```
reinterpret_cast<float4*>(y)[i] += reinterpret_cast<float4*>(x)[i];
```

Tools: nvprof for profiling CUDA kernels

- nvcc is useful for static analysis of kernels
- nvprof: useful for **runtime** analysis, also an easier way to time kernels.
  - nvprof has 100+ metrics which provide very detailed insight into performance of CUDA kernels (e.g., occupancy, bandwidth, number of memory transactions, shared memory efficiency, etc).
- Alternatives: Nvidia Visual Profiler (nvvp)
  - Will suggest areas to optimize based on nvprof data
  - Modern GPUs (compute capability 7.5 and up) have moved to “Nsight”

Tools: nvprof for profiling CUDA kernels

- Demo with add_vectors.cu on NOTSx.
- Steps: compile as usual via “nvcc add_vectors.cu”, then, run the program via “nvprof --metrics all ./a.out 10000 128”.
- Some suggestions:
  - The -arch flag specifies the minimum GPU compute capability that your program requires. For example, adding the flag “-arch=sm_70” to your nvcc command will allow nvprof to provide more precise info.
  - Note that “sm_70” is specific to the NOTSx V100 GPUs; see https://arnon.dk/matching-sm-architectures-arch-and-gencode-for-various-nvidia-cards/.

float vs double in CUDA

```
__global__ void add(const int N, const float *x, float *y){
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N){
    y[i] = y[i] + 2.0 * x[i]; // this result gets promoted to double!
  }
}
```

- GPUs are generally much more efficient at single precision operations (more single precision cores than double precision cores).
- Your performance may drop if you introduce double precision operations, so be careful with numeric constants (e.g., 1.0 (double) vs 1.f (float) or “#define PI 3.14159f”)

float vs double in CUDA

- If you intend to use single precision, use “nvprof” to detect stray double precision operations (nvprof –metrics flop_count_dp, flop_count_sp …).
- Can make switching both single and double precision tricky.
  - One option: use a macro to switch between float and double

```
#define datafloat (float or double)
datafloat a = 1.f; // will cast to a double if datafloat==double
```

Some additional CUDA functionality

- **Functions:** we've only used __global__ to define GPU *kernels*, which are executed by the CPU to be run on the GPU
  - Can also define __device__ functions, which are called from the GPU.
- **Templates:** added in CUDA 7 around 2015 - you can pass parameters into a CUDA kernel using C++ 11 style template parameters.
  - In this class, we used #define to construct computational parameters, e.g., BLOCKSIZE. These are resolved at compile-time, which is necessary to construct static shared memory arrays.
  - Because both macros and C++templates are resolved at compile-time, they should both behave similarly.

Alternatives to CUDA: OpenACC

```
#pragma acc kernels copyin(a[0:n],b[0:n]), copyout(c[0:n])
for(i=0; i<n; i++) {
    c[i] = a[i] + b[i];
}

// compile with -acc flag, similar to -fopenmp framework
```

- Some GPU support in OpenMP 4.0+. OpenACC (acc = accelerators) is more mature than OpenMP for GPUs IMO.
  - Not as efficient as OpenMP is for multi-threading.
  - Supports more features on Nvidia GPUs and CUDA.

Alternatives to CUDA: domain specific languages (DSL)

- “Portability” libraries: translates a DSL to CUDA, Metal, OpenMP, etc.
- Kokkos: Sandia National Laboratories, most popularly used, aims to work with standard C++ functionality.
- Raja: Lawrence Livermore National Labs (LLNL), allows for more detailed optimizations than Kokkos.
- OCCA: Virginia Tech + LLNL, lightweight, maps to both GPU/OpenMP/CPU. Highly optimizable.
- Others: AMD HIP, SYCL, HPX, ROCm, and probably more that I’ve missed.
- Note that with any additional library, you introduce more complexity and potential bugs, and not all CUDA features may be supported.

GPU computing in Python

- Nowadays, most scientific computing is done in higher level languages like Python and Julia. GPU kernels can be called from within these.
- Python: PyCUDA and PyOpenCL are maintained by Andreas Klockner (https://andreask.cs.illinois.edu/aboutme/)
  - Allows users to call CUDA (and OpenCL) kernels from within Python
  - Mature software: supported since 2009

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print(dest-a*b)
```

GPU computing in Julia

- I use the Julia programming language in most of my work.
- Julia uses *multiple dispatch* (conceptually similar to function overloading in C++) to specialize functions on different *types*.
  - Example: add(x::float, y::float) vs add(x::float4, y::float4)
- CUDA.jl introduces custom CuArray types which live on the GPU.
  - Array operations and linear algebra get dispatched to CUBLAS.
  - Can still write custom kernels which take CuArrays as inputs.
- KernelAbstractions.jl is a portability library in Julia (mostly CPU and CUDA).

Course recap

- Four main components
  - Serial single CPU programming
  - Shared memory parallelism: multi-core CPUs, OpenMP
  - Distributed parallelism: MPI
  - GPU computing: CUDA
- Overarching themes: Moore’s law has made arithmetic operations fast, but the cost of accessing memory has not yet caught up.
  - HPC is about making efficient use of memory on each architecture

HPC is about reducing memory access costs

- Serial single CPU programming
  - Make memory reads as fast as possible with contiguous memory accesses
  - Design implementations to efficiently use L1, L2, …, etc cache.
- Shared memory parallelism: multi-core CPUs, OpenMP
  - Same concerns as CPU programming.
  - False sharing can reduce scalability due to serialized (i.e., non-parallel) memory accesses.
  - More focused on how to use OpenMP clauses + tasks.

HPC is about reducing memory access costs

- Distributed parallelism: MPI
  - Message passing paradigm - no shared memory pool, all communications must be explicitly requested.
  - Reducing memory access costs is still important, but is more commonly analyzed in terms of *communication* costs.
- GPU computing: CUDA
  - Oddly similar to the CPU IMO: RAM vs global memory, contiguous vs coalesced memory reads, L1 cache vs shared memory.

Thank you!