CS 250B: Modern Computer Systems

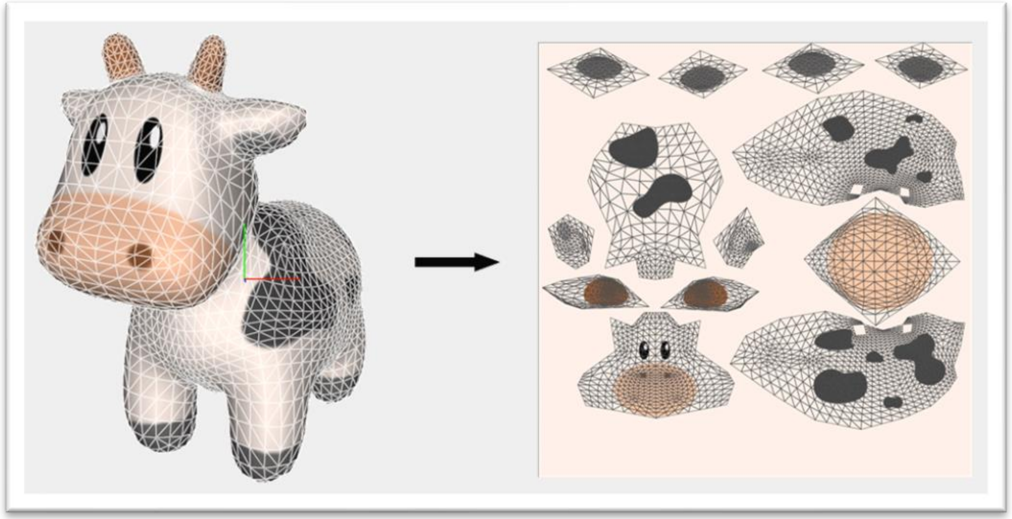
GPU Computing Introduction

Sang-Woo Jun

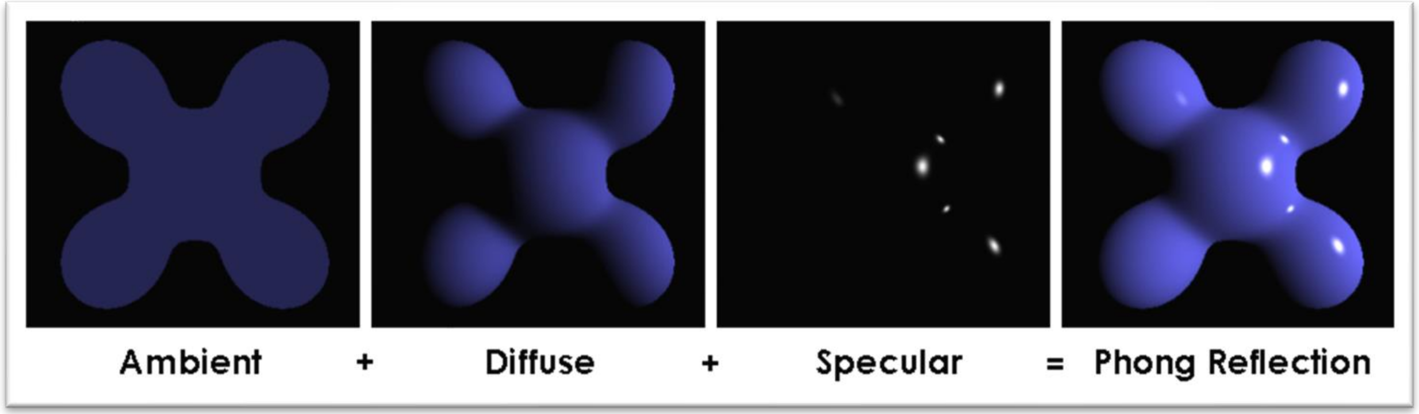
UCI

Graphic Processing – Some History

- 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- 3D graphics processing is immensely computation-intensive



Texture mapping



Shading

Warren Moore, "Textures and Samplers in Metal," Metal by Example, 2014
Gray Olsen, "CSE 470 Assignment 3 Part 2 - Gourad/Phong Shading," grayolsen.com, 2018

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

Doom (1993) : “Affine texture mapping”

- Linearly maps textures to screen location, disregarding depth
- Doom levels did not have slanted walls or ramps, to hide this

Graphic Processing – Some History

- Before 3D accelerators (GPUs) were common
- CPUs had to do all graphics computation, while maintaining framerate!
  - Many tricks were played

Quake III arena (1999) : “Fast inverse square root” magic!

```c
float Q_rsqrt( float number )
{
    const float x2 = number * 0.5F;
    const float threehalfs = 1.5F;

    union {
        float f;
        uint32_t i;
    } conv = {number}; // member 'f' set to value of 'number'.
    conv.i  = 0x5f3759df - ( conv.i >> 1 );
    conv.f  *= ( threehalfs - ( x2 * conv.f * conv.f ) );
    return conv.f;
}
```

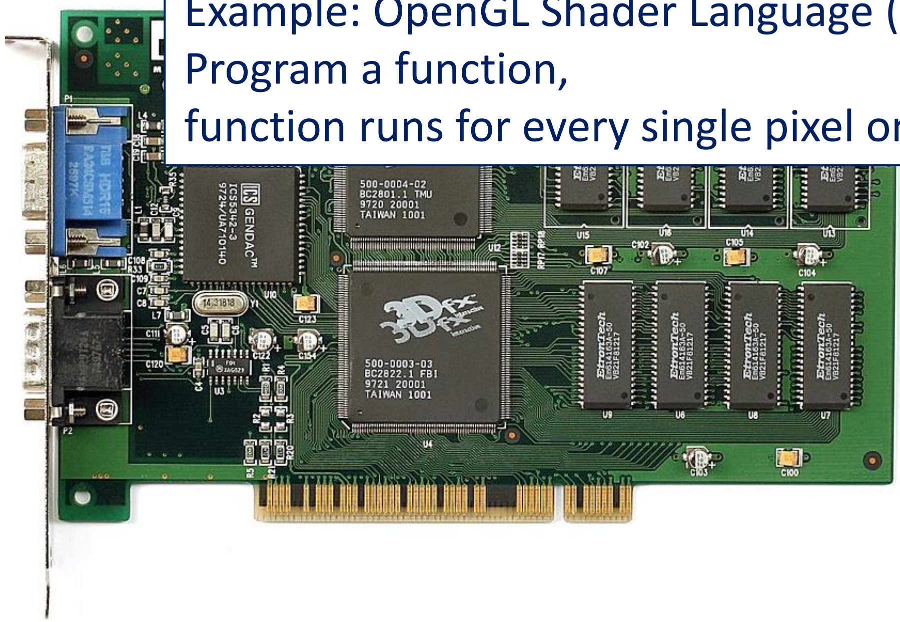
Introduction of 3D Accelerator Cards

- Much of 3D processing is short algorithms repeated on a lot of data
  - pixels, polygons, textures, …
- Dedicated accelerators with simple, massively parallel computation

Motoracer: The Monster 3D offers a perfect 3D image.

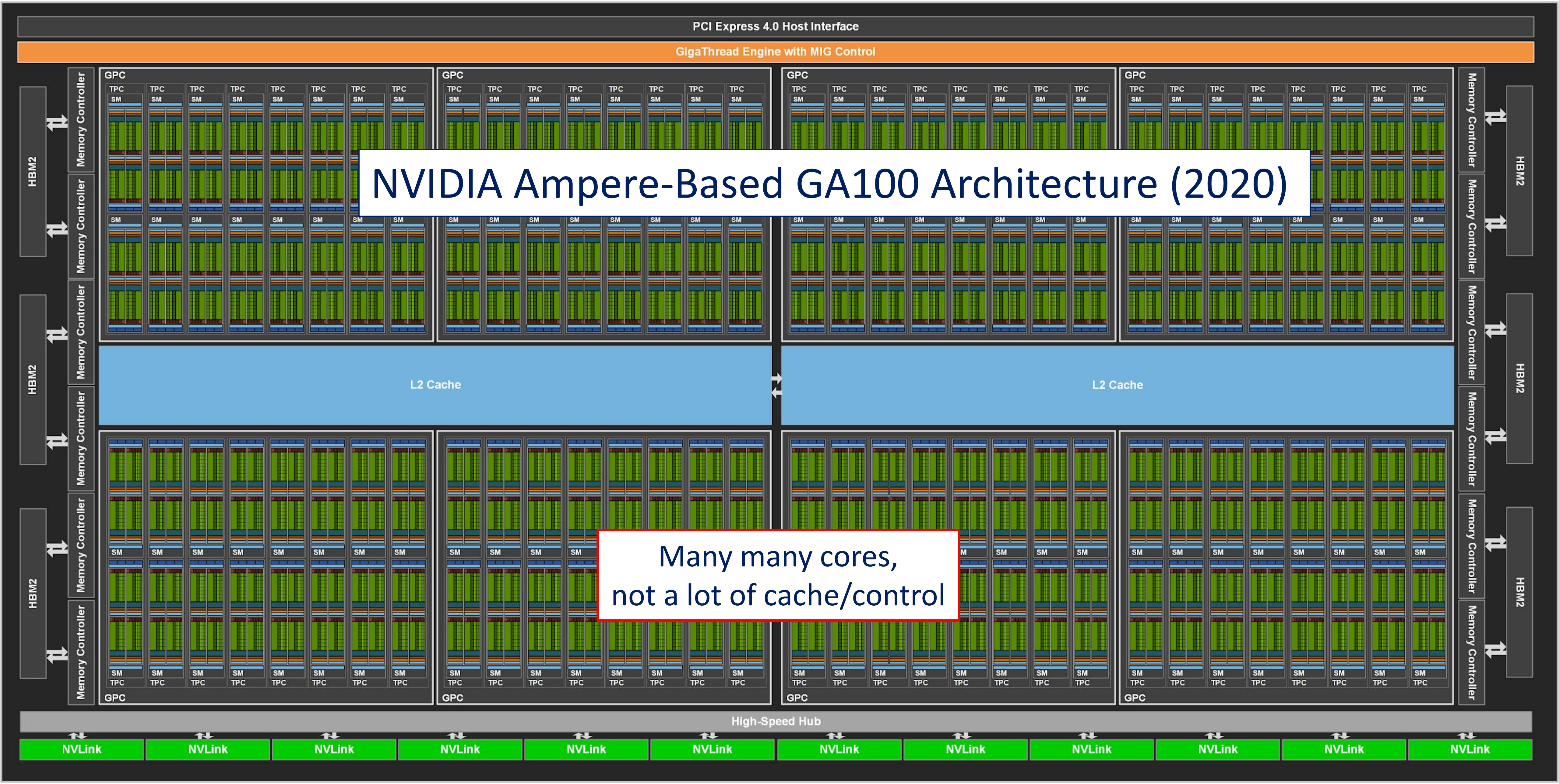
Source: Tom's Hardware, 1997

Example: OpenGL Shader Language ("GLSL")
Program a function,
function runs for every single pixel on screen



A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)

NVIDIA Ampere-Based GA100 Architecture (2020)

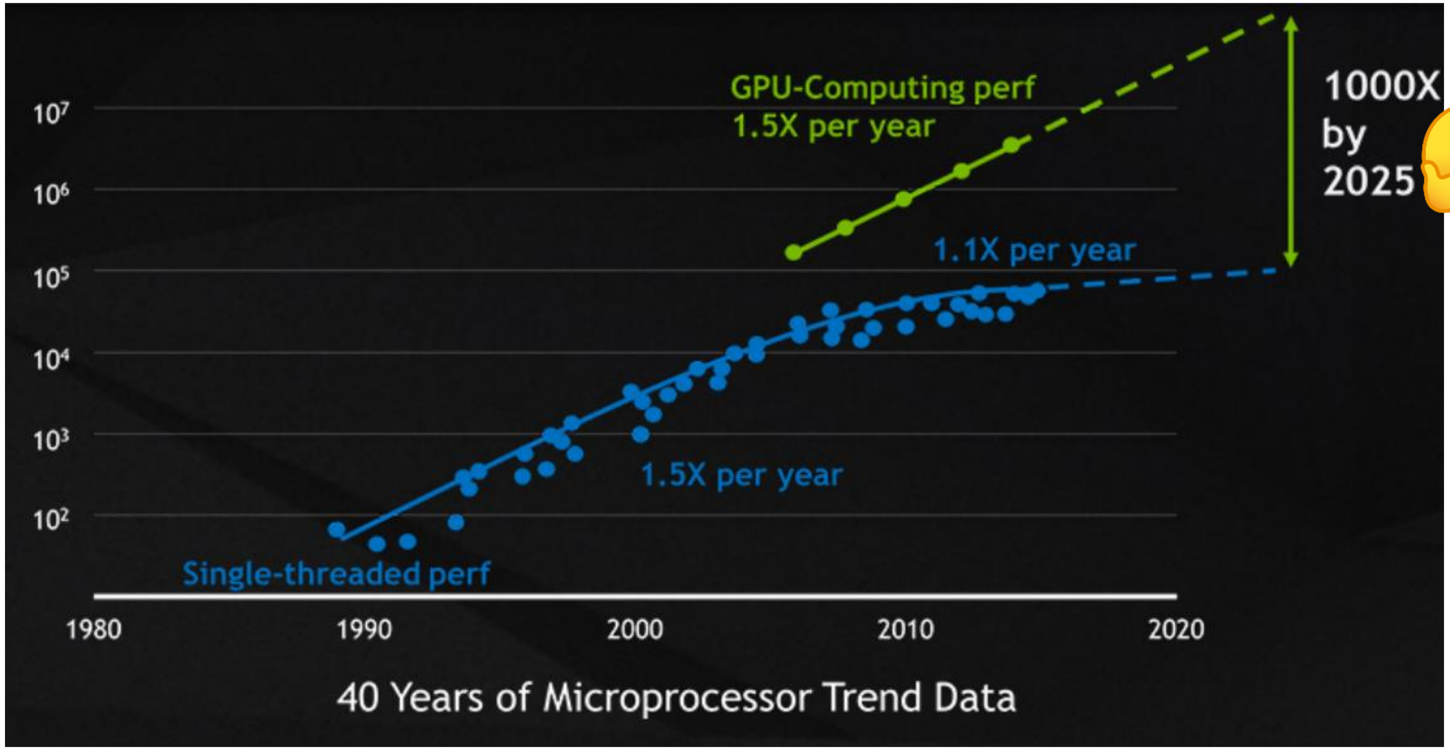


Many many cores,
not a lot of cache/control

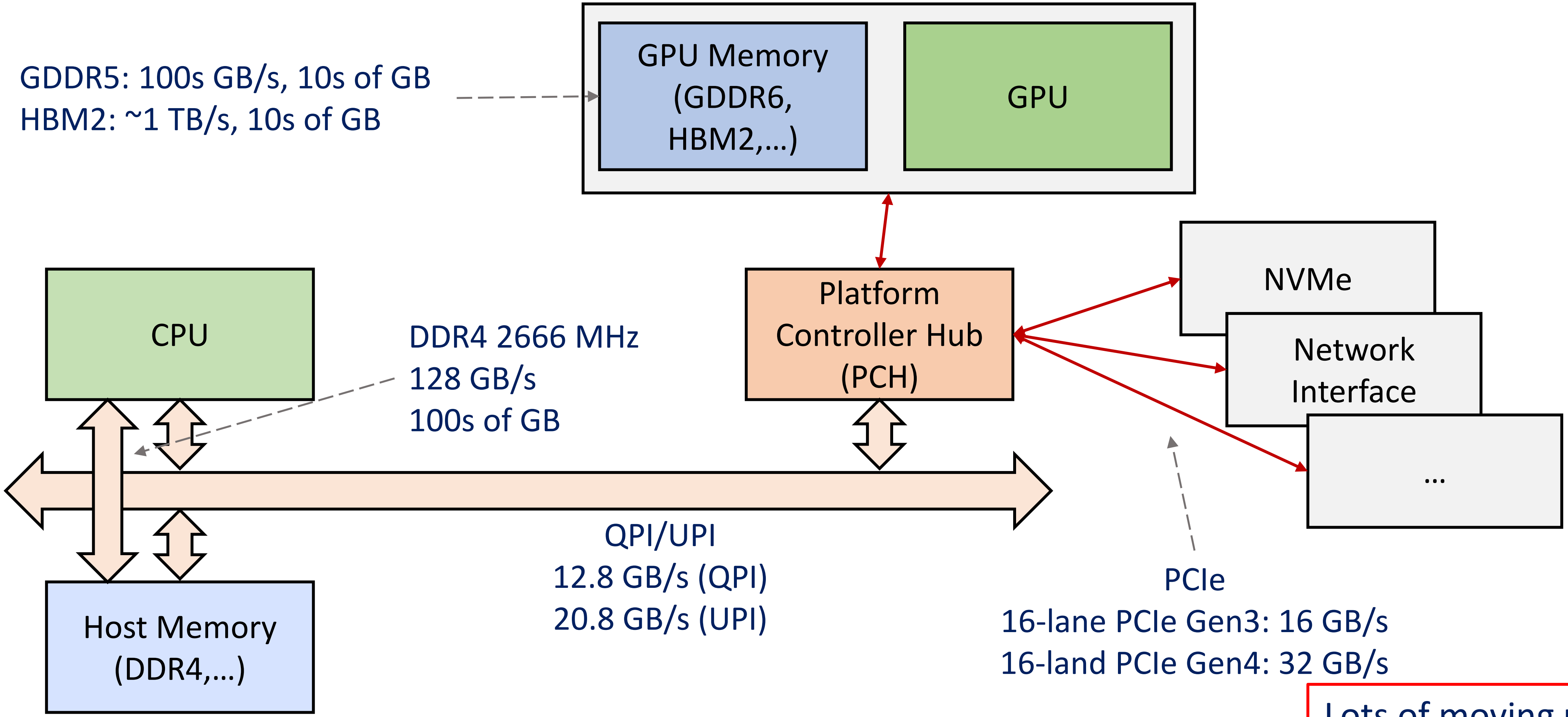
Peak Performance vs. CPU

	Throughput	Power	Throughput/Power
Intel Skylake	128 SP GFLOPS/4 Cores	100+ Watts	~1 GFLOPS/Watt
NVIDIA V100	15 TFLOPS	200+ Watts	~75 GFLOPS/Watt

Also,



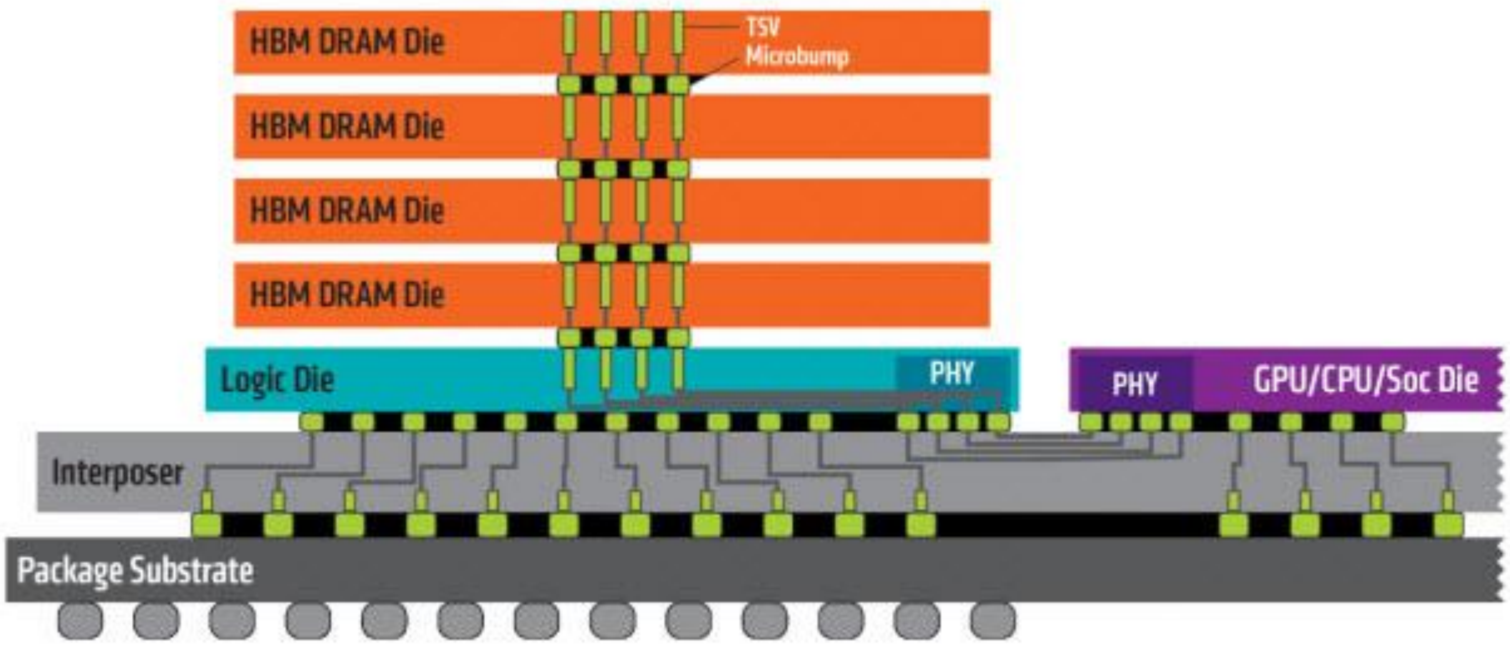
System Architecture Snapshot With a GPU



Lots of moving parts!

High-Performance Graphics Memory

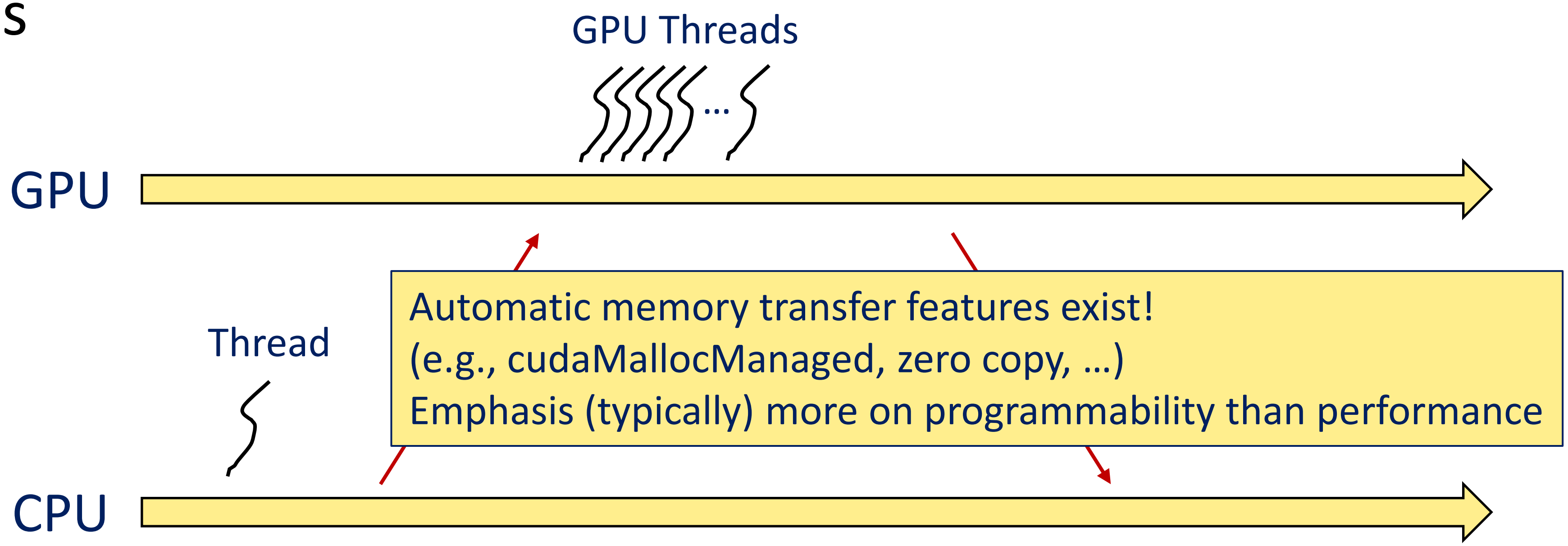
- Modern GPUs even employing 3D-stacked memory via silicon interposer
  - Very wide bus, very high bandwidth
  - e.g., HBM2 in Volta, Ampere



Graphics Card Hub, “GDDR5 vs GDDR5X vs HBM vs HBM2 vs GDDR6 Memory Comparison,” 2019

Massively Parallel Architecture For Massively Parallel Workloads!

- NVIDIA CUDA (Compute Uniform Device Architecture) – 2007
  - A way to run custom programs on the massively parallel architecture!
- OpenCL specification released – 2008
- Both platforms expose synchronous execution of a massive number of threads



The Hardware Lottery

Sarah Hooker

Communications of The ACM, 2021

UCI

Hardware Lottery Winners: General-Purpose CPU Threads

- Moore’s Law + Dennard Scaling = Dependable performance scaling
- Faster general-purpose hardware available next year
  - Why risk uncertain reward with specialized designs?!
- Resources focused on general purpose CPUs faster

Hardware Lottery Winners: General-Purpose CPU Threads

- Von-Neumann general-purpose CPUs
  - Not very good with parallel execution
  - Not much emphasis on memory bandwidth

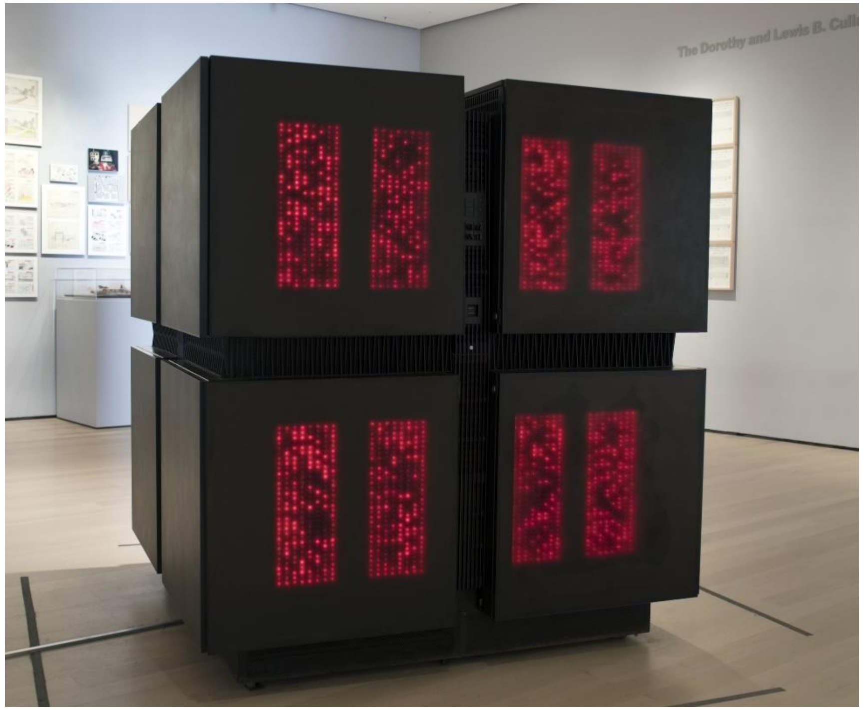
- Efficient with branch-heavy expert systems
  - Favors symbolic approaches to AI (LISP, Prolog)
- Inefficient with massively parallel matrix multiplication
  - Disfavors neural networks

Hardware Lottery Losers: Neural Nets and the AI Winter

- “The lost decades”, or the “AI Winter”
  - Research predominantly focused on symbolic approaches
  - Insufficient hardware capacity to train realistic neural nets
- NN theory was already available
  - Backpropagation (1963, reinvented in 1976, and again in 1988)
  - Deep convolutional neural networks (1979, paired with backpropagation in 1989)
  - Need for parallel architectures and memory already noticed in 1980
- But… already lost the hardware lottery

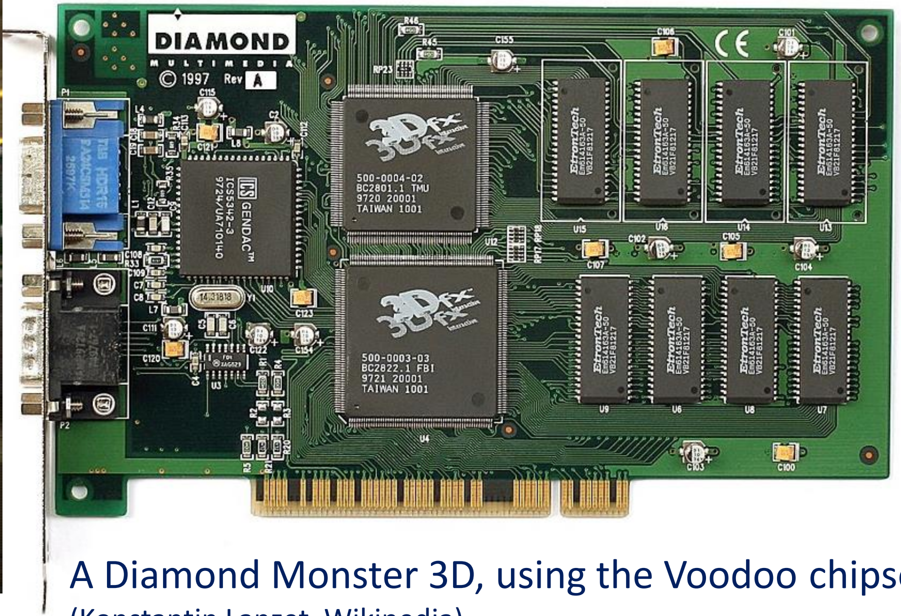
Hardware Lottery Losers: Neural Nets and the AI Winter

- Ventures into specialized hardware for NN existed
  - e.g., “Connection Machine” (pictured), 1985
- But none reached critical mass
  - Fractured ISA, programming model
  - No application -> No customers -> No research -> No application…



New Hardware Lottery Winners: GPUs

- A “fluke” in the 2000s enabled neural networks
  - GPUs originally designed for gaming
  - Massively parallel, a program for each pixel (for example)
  - Re-purposed for training!



A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)

CNNs and GPUs – Perfect Match

- Two papers using CNNs to identifying cats

- “Building High-Level Features Using Large Scale Unsupervised Learning”
  - 16,000 CPU cores
  - 2012
- “Deep learning with COTS HPC systems”
  - Two CPU cores and two GPUs
  - 2013

What other ideas are we missing due to the hardware lottery?

Yet Another Lottery Winners: Specialized Hardware

- CNNs have reached critical mass, won the hardware lottery (finally)
  - Hardware is optimizing for CNNs
  - Tensor cores in GPUs, bfloat units in CPUs, TPUs, …
  - Quantized arithmetic, unstructured pruning, etc making way into hardware
- Specialized hardware enables ever-larger models
  - The baseline models are becoming very deep, very large

Yet Another Lottery Losers: Non-CNN Models

- But, other ideas have lost the lottery
  - If an alternative algorithm is as complex as CNNs but not trainable with TPUs
  - Not feasible to train!
  - Imagine training a modern NN without GPUs
- Example: “Capsule Networks” (2019)
  - “include novel components like squashing operations and routing by agreement.”
  - “aimed to solve for key deficiencies in convolutional neural networks (lack of rotational invariance and spatial hierarchy understanding)”
  - “but strayed from the typical architecture of neural networks as a sequence of matrix multiplies.”

Yet Another Lottery Losers: Non-CNN Models

- Are capsule nets the future? Maybe, maybe not!
- But, researchers will gravitate towards models/algorithms well-suited for GPU/TPU/Matrix multiply.
  - And away from those unsupported
- What great ideas are we missing because they lost the hardware lottery?

Back to CUDA…

CUDA Execution Abstraction

- Block: Multi-dimensional array of threads
  - 1D, 2D, or 3D
  - Threads in a block can synchronize among themselves
  - Threads in a block can access shared memory
  - CUDA (Thread, Block) ~= OpenCL (Work item, Work group)
- Grid: Multi-dimensional array of blocks
  - 1D or 2D
  - Blocks in a grid can run in parallel, or sequentially
- Kernel execution issued in grid units
- Limited recursion (depth limit of 24 as of now)

GPU programming abstraction

- "SIMT" (Single Instruction Multiple Threads), introduced by NVIDIA
  - Simply put: Identical program ("Kernel") executed on multiple threads
  - Thread ID is given as a parameter to the program,
    so each thread can perform different work despite identical code
  - Another kernel parameter is "block size", the number of threads to use

CPU Code example

```
for (ii = 0; ii < cnt; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

GPU Code example

```
__global__ void KernelFunction(...) {
   int tid = threadIdx.x;
   int blocksize = ceiling(cnt/blockDim.x);
   for (i = 0; i < blocksize; ++i ) {
      int ii = blocksize*tid+i;
      if ( ii < cnt ) C[ii] = A[ii] + B[ii];
   }
}
```

Thread dimensions given as part of request from host software

Simple CUDA Example

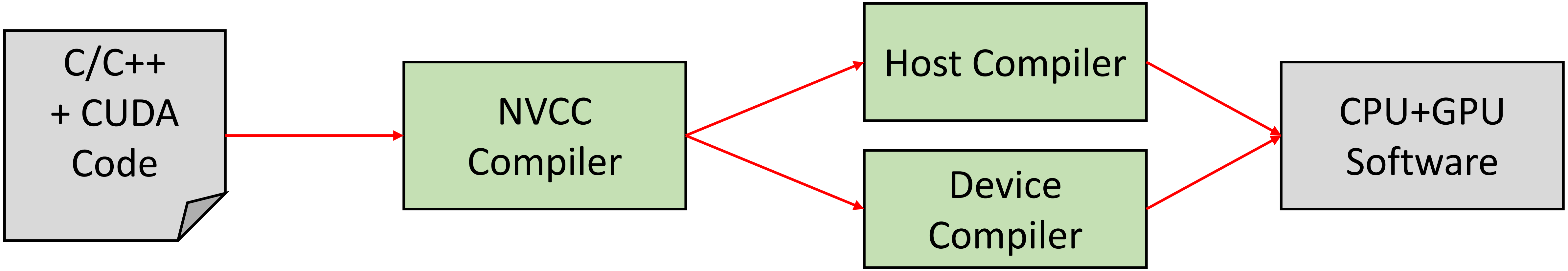
Asynchronous call

CPU side

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

GPU side

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```



Simple CUDA Example

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);

}
```

1 block

N threads per block

Should wait for kernel to finish

__global__:
In GPU, called from host/GPU

__device__:
In GPU, called from GPU

__host__:
In host, called from host

N instances of VecAdd spawned in GPU

One function can be both

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

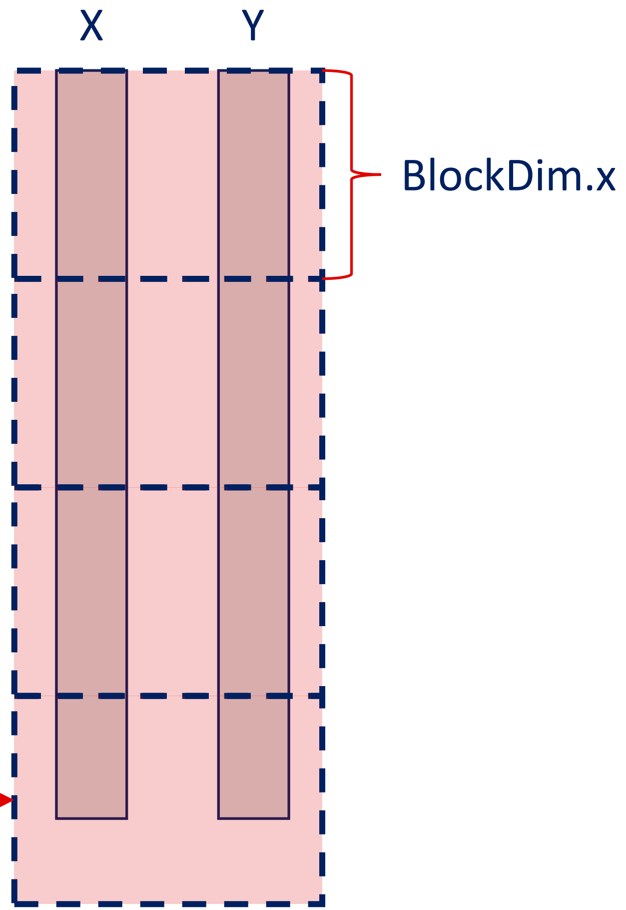
Only void allowed

Which of N threads am I?
See also: blockIdx

End-to-End Example: SAXPY

- “Single-precision A*X Plus Y”

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```



End-to-End Example: SAXPY

```
int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
...
  cudaM                              tToDevice);
  cuda                               tToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Host Memory

Device Memory

Copy to Device

Call Kernel

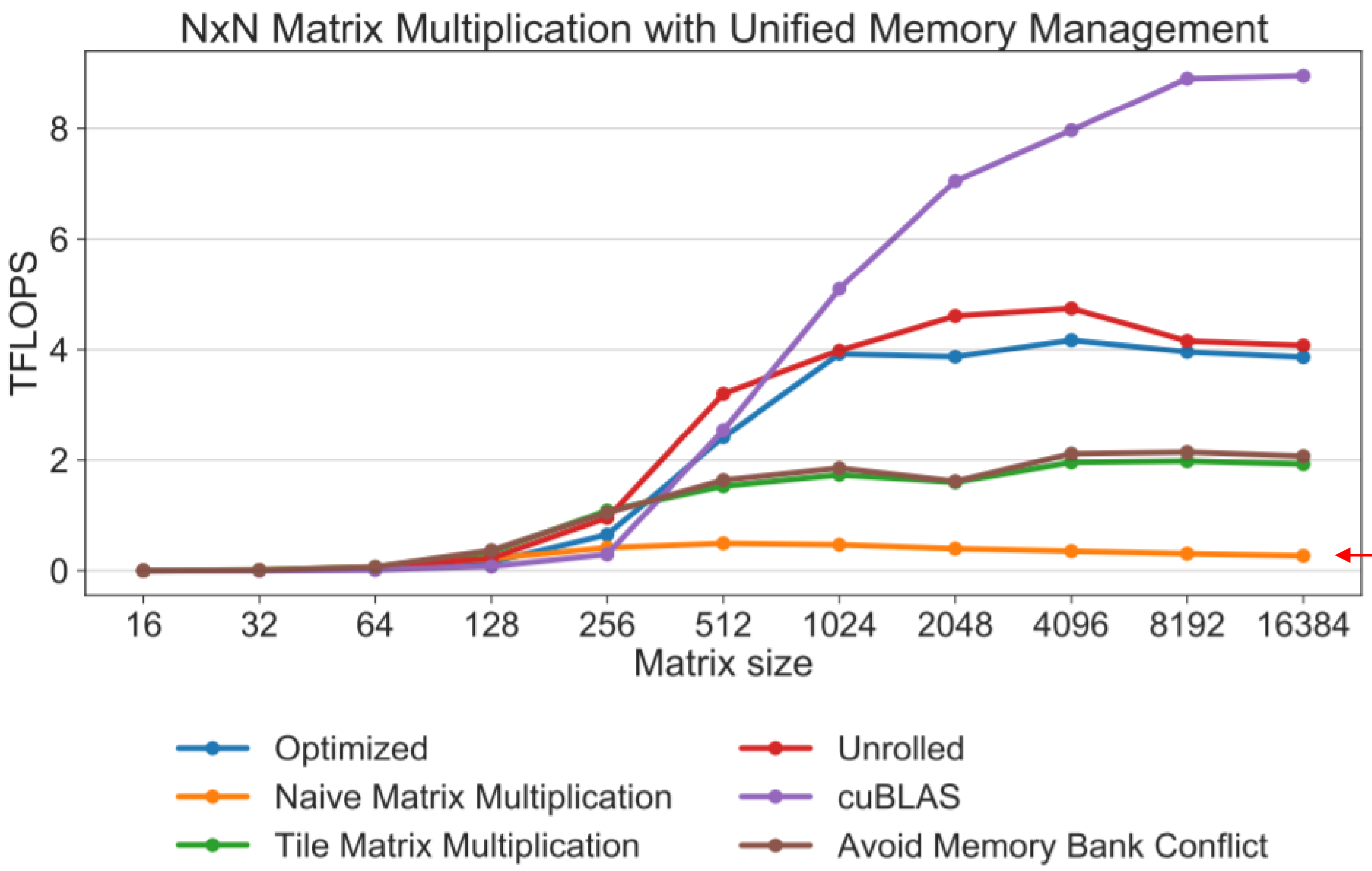
Copy Result

```
% nvcc -o saxpy saxpy.cu
% ./saxpy
```

Great! Now we know how to use GPUs

Bye?

Matrix Multiplication
Performance Engineering

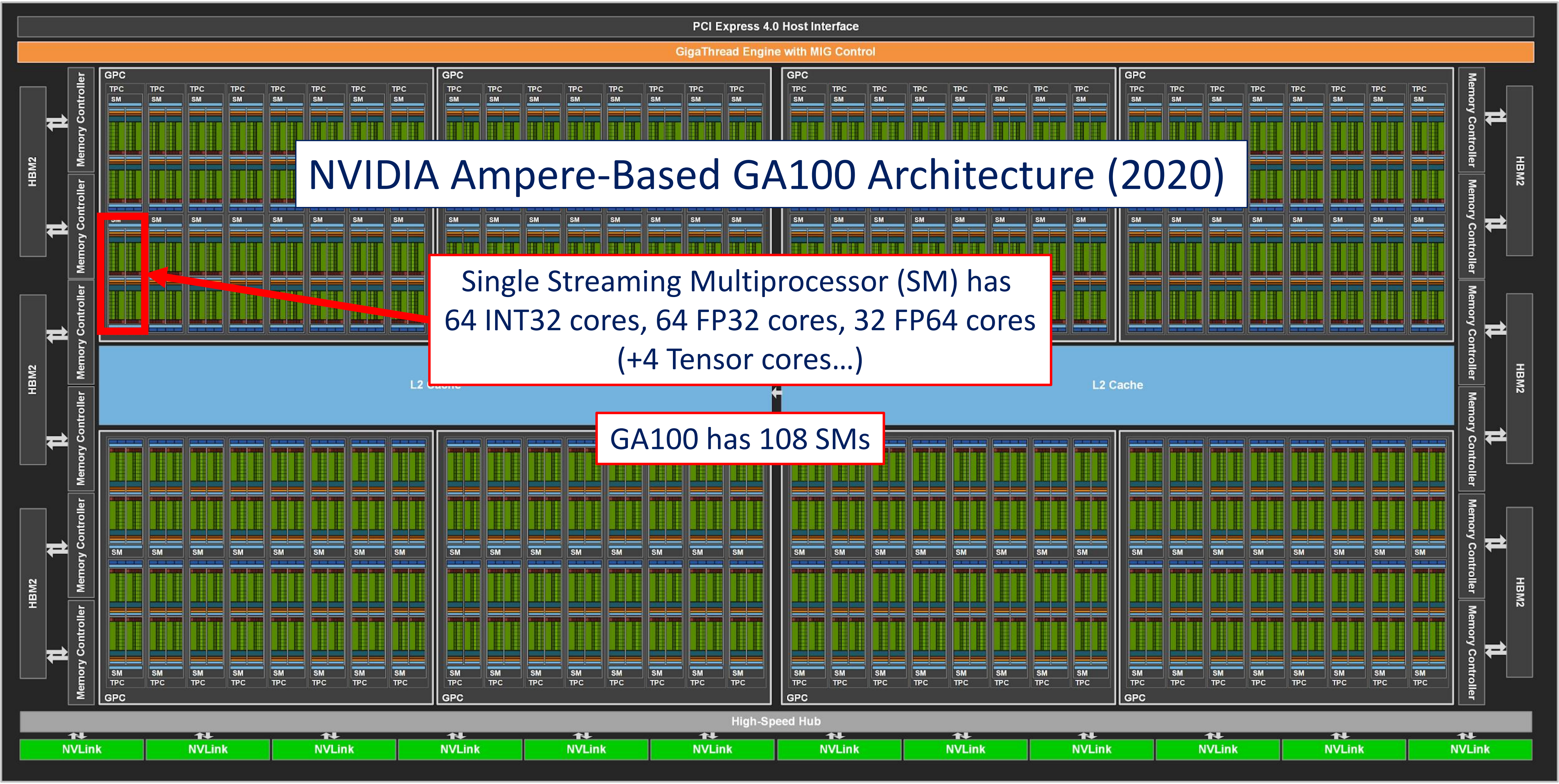


No faster than CPU

Results from NVIDIA P100

Coleman et. al., “Efficient CUDA,” 2017

Architecture knowledge is needed (again)



NVIDIA Ampere-Based GA100 Architecture (2020)

Single Streaming Multiprocessor (SM) has 64 INT32 cores, 64 FP32 cores, 32 FP64 cores (+4 Tensor cores…)

GA100 has 108 SMs

Ampere Execution Architecture

- 64 INT32, 64 FP32, 32 FP64, 4 Tensor Cores
  - Specialization to make use of chip space…?
- Not much on-chip memory per thread
  - 164 KB Shared memory
  - 256 Registers
- Hard limit on compute management
  - 32 blocks AND 2048 threads AND 1024 threads/block
  - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
  - Enough registers/shared memory for all threads must be available (all context is resident during execution)

More threads than cores – Threads interleaved to hide memory latency

Resource Balancing Details

- How many threads in a block?
- Too small: 4x4 window == 16 threads
  - 128 blocks to fill 2048 thread/SM
  - SM only supports 32 blocks -> only 512 threads used
    - SM has only 64 cores… does it matter? Sometimes!
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
  - Runtime error: “invalid configuration argument”
- Too large: 1024 threads using more than 256 Byte registers
- Limitations vary across platforms (Fermi, Pascal, Volta, Ampere, …)

CS 250B: Modern Computer Systems

GPU Architecture And Performance

Sang-Woo Jun

UCI

GPU Processor Architecture

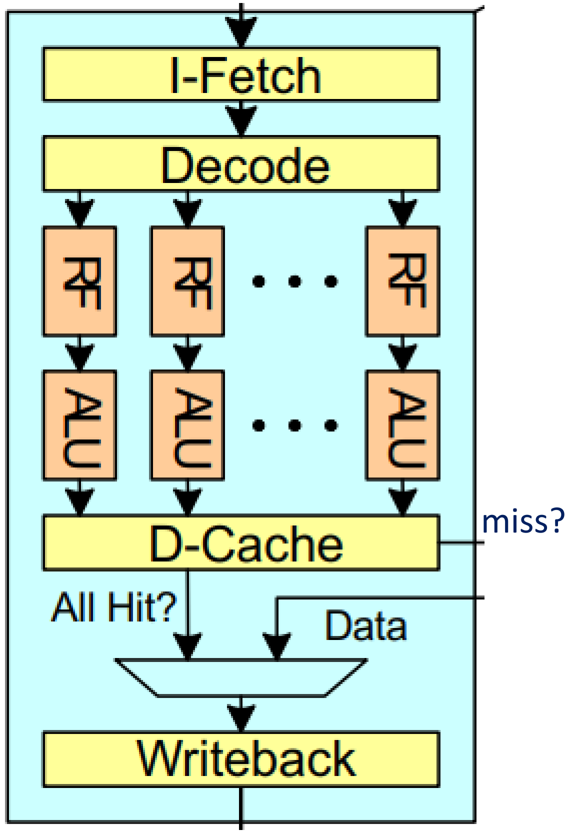
- GPUs have thousands of threads running concurrently at GHzs

- Much simpler processor architecture
  - Dozens of threads scheduled together in a SIMD fashion
  - Much simpler microarchitecture (doesn’t need to boot Linux)
- Much higher power budget
  - CPUs try to maintain 100 W power budget (Pentium 4 till now)
  - Thermal design power (TDP) for modern GPUs around 300 W
    - TDP: Safe level of power consumption for effective cooling

CPU (i7) adding 1 Billion floats: 2.14s, NVIDIA Turing with only one thread: 29.16s

GPU Processor Architecture

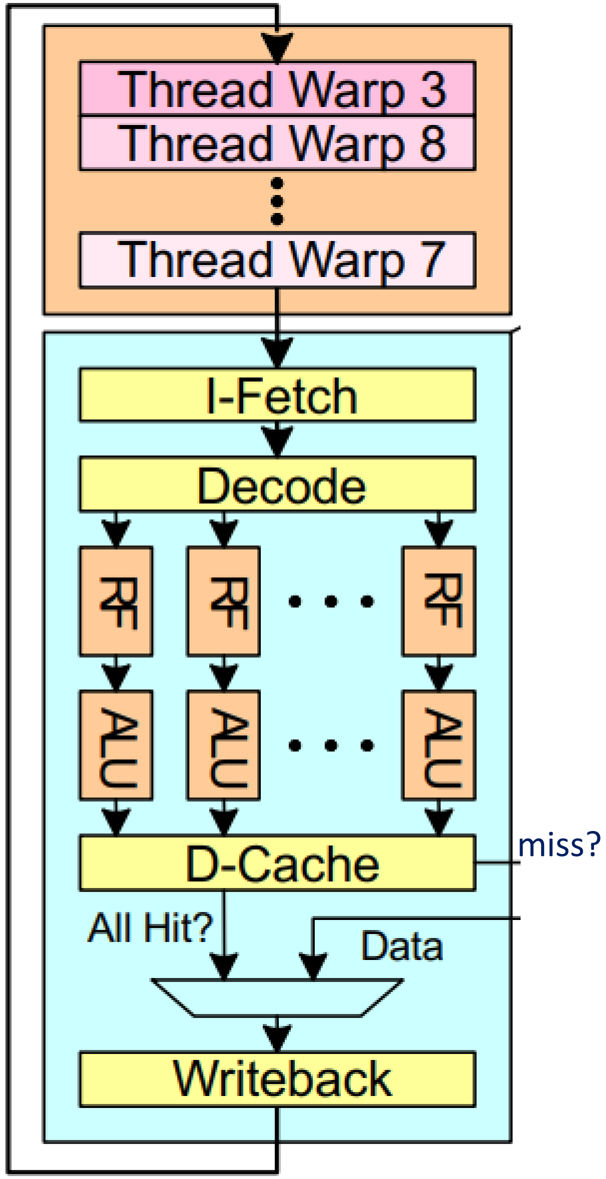
- Cores are organized into units of “warps”
  - Threads in a warp share the same Fetch and decode units
  - Drastically reduces chip resource usage
    - One reason why GPUs can fit so many cores
  - Basically a warp is one thread with SIMD operations
    - But exposes multithread abstraction to the programmer
  - Typically 32 threads per warp for NVIDIA, but may change
    - Warp size information is not part of programming abstraction



Source: Tor Aamodt

GPU Processor Architecture

- Each warp hardware can handle many sets of threads
  - Context switch in case of memory access request, to hide memory access latency
- A large block of threads can map across many streaming multiprocessors
  - Thread 0 to 31 map to warp 0,
    Thread 32 to 63 map to warp 1, …



Thread Synchronization in CUDA

- Synchronization is possible within a block
  - __syncthreads() is a barrier operation
- Synchronization is unnecessary within a warp
  - SIMD anyways
- Synchronization is not (easily) available between blocks
  - __syncthreads() does nothing
  - No shared memory
  - We can implement synchronization using slow global memory…

So far, typical parallel, multithreaded programming

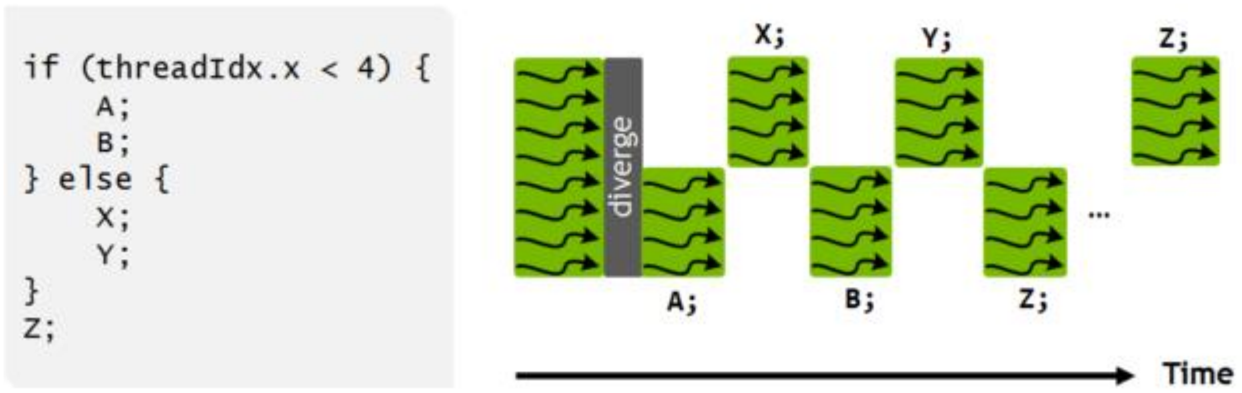
But, caveats for performance engineering starts here!

Warp Scheduling Caveats

- Remember: Threads within a block share the same fetch, decode units
  - All threads in a warp are always executing the same instruction
  - What if their execution diverges?
    - e.g., if (tid%2) func1(), else func2()
    - e.g., if (A[tid] < 100) X++, else Y++
- Divergence across warps don't matter
  - Different warps, different fetch+decode
- What about intra-warp divergence?

Warp Scheduling Caveats

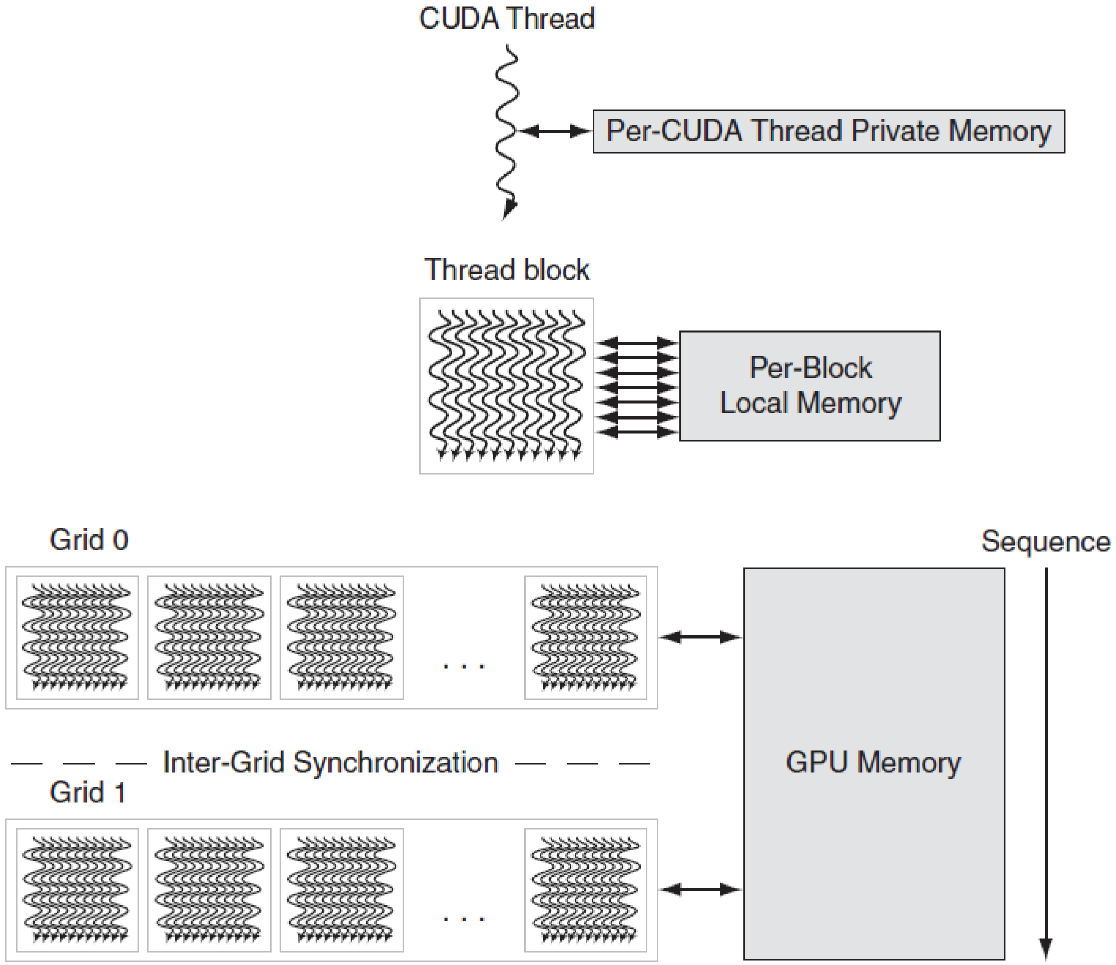
- Intra-warp execution divergence incurs “control divergence”
  - The warp processor must execute both paths, one after another
    - Whole warp will execute one direction first with some threads suspended, and the other direction with the other threads suspended
  - If 32 threads go down 32 different branches, no performance gain with SIMD!



2018, “Using CUDA Warp-Level Primitives,” NVIDIA

GPU Memory Architecture

- ❑ Not much on-chip memory per thread
  - 256 Registers per FP32 core
  - 164 KB Shared memory
- ❑ Relatively fast off-chip “global” memory
  - But not fast enough!
  - GDDR6 or HBM2 can deliver up to +1TB/s
  - Shared across 2048+ threads…
- ❑ Pretty much no memory consistency between blocks
  - Once data goes to off-chip main memory, explicit synchronization critical!
    - Expensive!

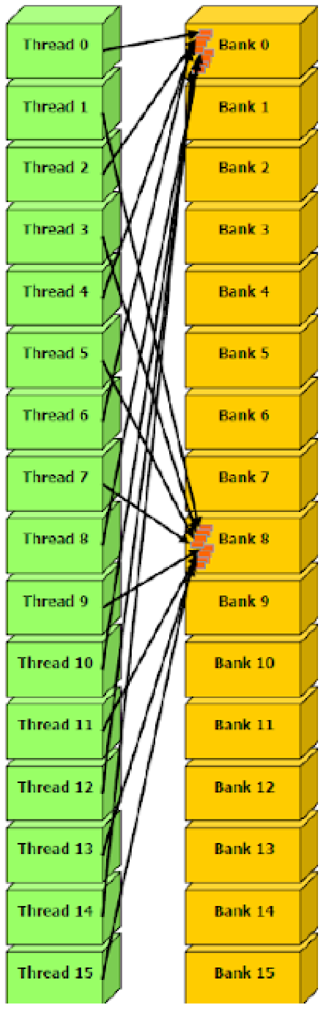


Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”

GPU Memory Architecture

- Remember: A block can have thousands of threads
  - They can all be accessing shared memory at once
  - Shared memory hardware can’t have a port for each thread
  - Serializing memory access will kill performance
    - Performance will be limited by one shared memory access per thread per cycle

- Organized into banks to distribute access
  - Best performance if all threads in warp access different banks
  - Best performance if all threads access the same address (broadcast)
  - Otherwise, bank conflicts drastically reduce performance



8-way bank conflict
1/8 memory bandwidth

Prominent Performance Engineering Topics

- Warp level execution
  - Avoid branch divergence within nearby threads
  - Algorithmic solutions for warp-size oblivious computations often possible
- Shared memory bank conflict
  - Map data access per thread to interleaved addresses
- Synchronization overhead
  - Avoid __syncthreads whenever possible (e.g., Within warp)
  - Avoid inter-block synchronization
- Memory reuse
  - Cache-optimized algorithms

CS 250B: Modern Computer Systems

GPU Application Examples

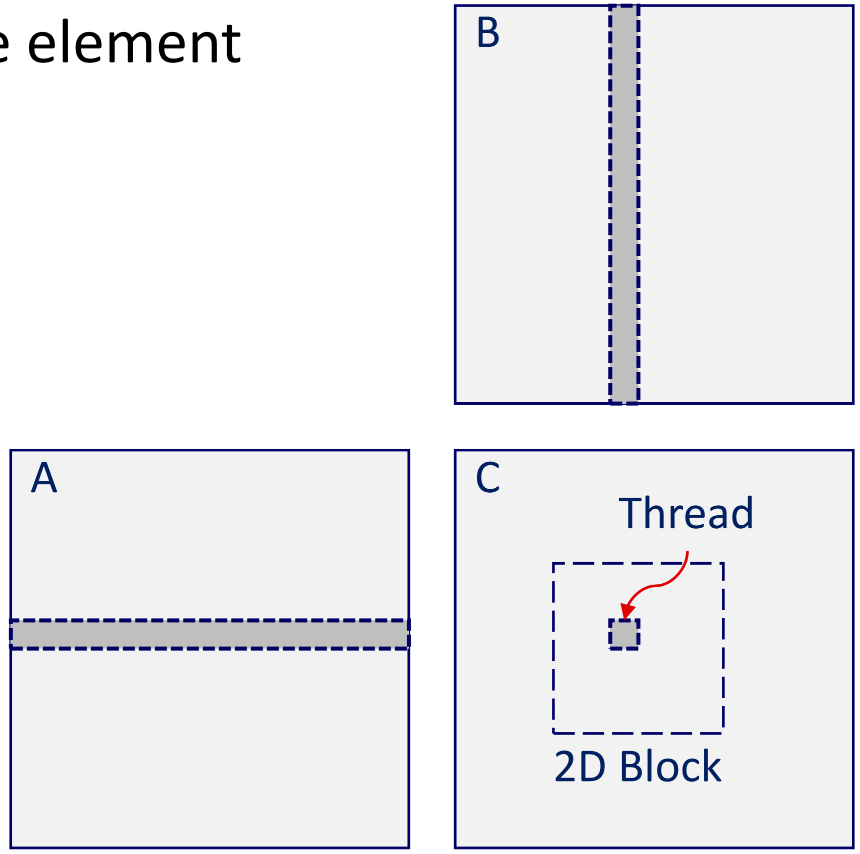
Sang-Woo Jun

UCI

Application 1: Matrix Multiplication

- Dividing Matrix Multiplication into blocks of threads
  - Simple solution: each thread responsible for one element
  - Remember: 1024 threads maximum per block
  - Spawn as many blocks as needed to cover C

- Shared memory is used to do some caching
  - Good enough?



A Naïve Matrix Multiplication Kernel

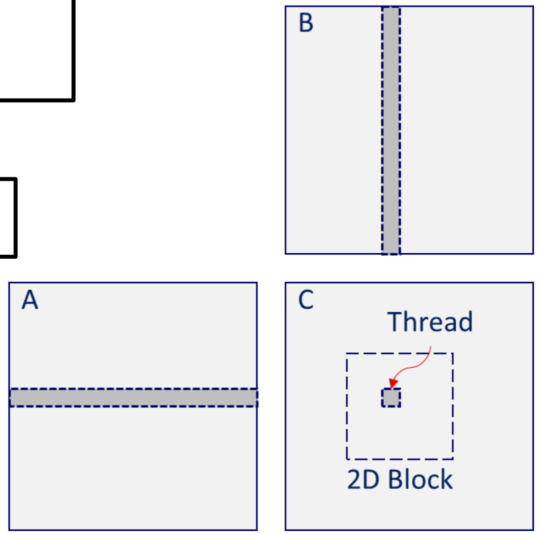
```
__global__ void MatrixMult0(float* a, float* b, float* c, int N) {
        int Row = blockIdx.y*blockDim.y+threadIdx.y;
        int Col = blockIdx.x*blockDim.x+threadIdx.x;

        if ((Row < N) && (Col < N)) {
                for (int k = 0; k < N; ++k) {
                        c[Row*N+Col] += a[Row*N+k]*b[k*N+Col];
                }
        }
}
```

```
MatrixMult0<<<dim3(N/BW,N/BW,1),dim3(BW,BW,1)>>>(d_a,d_b,d_c,N);
```

Width of a 2D square block of threads

Max threads per block: 1024
Max BW: 32



Performance So Far

- 16,384 x 16,384 Matrix
- NVIDIA RTX 2080 ti (Peak GFLOPS: 13500)
- Naïve implementation
  - Elapsed: 16.865s
  - GFLOPS: 521

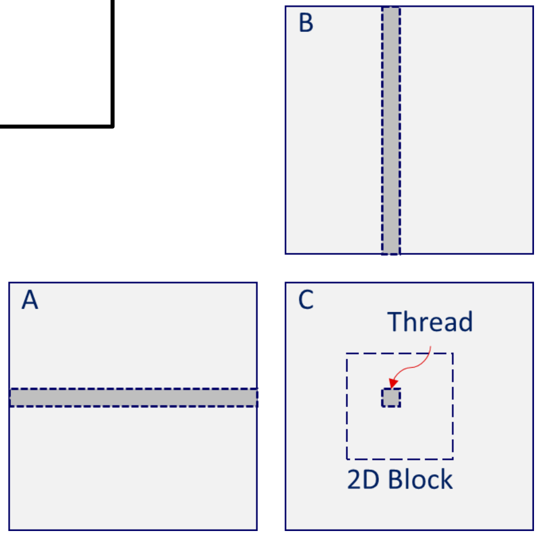
16K * 16K *16K * 4B / 616GB/s ~= 26s
… Some caching!

A Naïve Matrix Multiplication Kernel

```
__global__ void MatrixMult0(float* a, float* b, float* c, int N) {
        int Row = blockIdx.y*blockDim.y+threadIdx.y;
        int Col = blockIdx.x*blockDim.x+threadIdx.x;

        if ((Row < N) && (Col < N)) {
                for (int k = 0; k < N; ++k) {
                        c[Row*N+Col] += a[Row*N+k]*b[k*N+Col];
                }
        }
}
```

Is this reused?



Attempt 2: Local Variable For Reuse

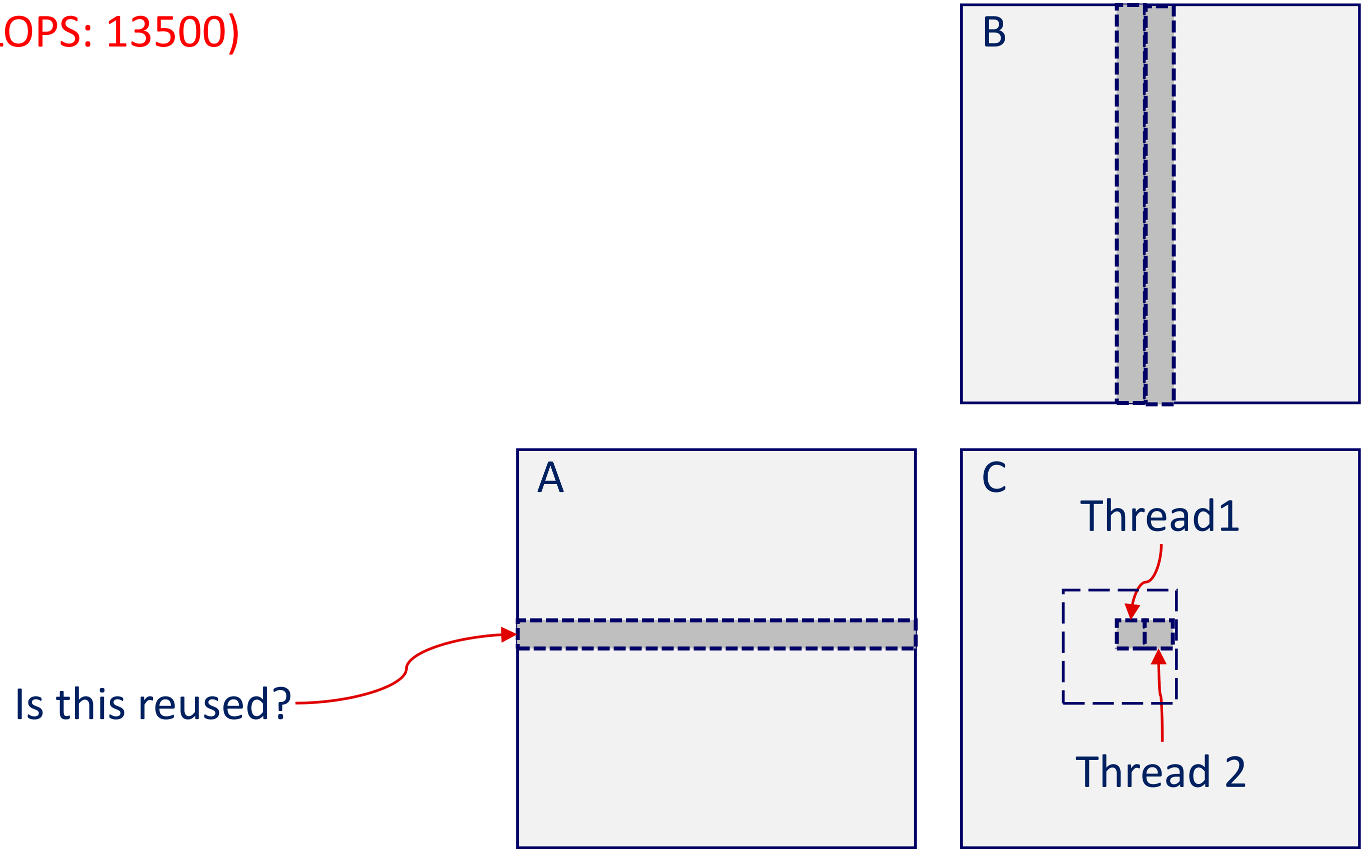
```
__global__ void MatrixMult1(float* a, float* b, float* c, int N) {
        int Row = blockIdx.y*blockDim.y+threadIdx.y;
        int Col = blockIdx.x*blockDim.x+threadIdx.x;

        if ((Row < N) && (Col < N)) {
                float Pvalue = 0;
                for (int k = 0; k < N; ++k) {
                        Pvalue += a[Row*N+k]*b[k*N+Col];
                }
                c[Row*N+Col] = Pvalue;
        }
}
```

Local variable for reuse

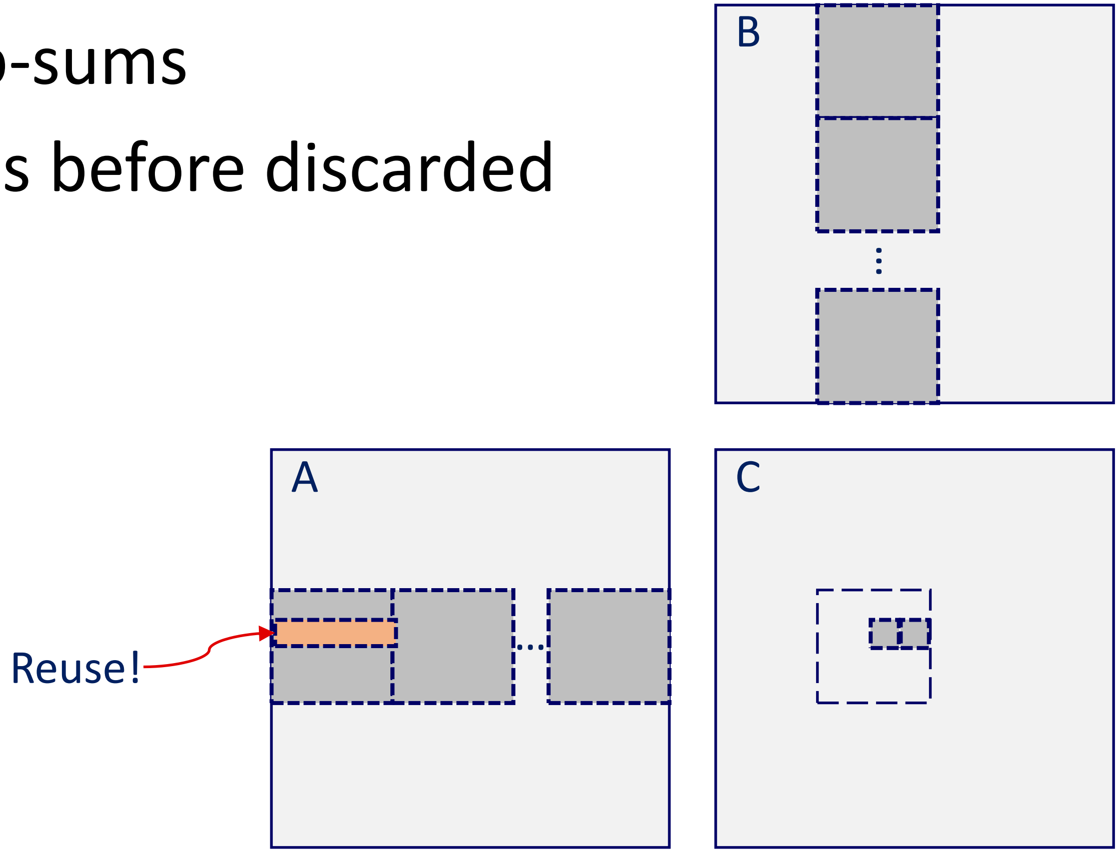
Performance So Far

- 16,384 x 16,384 Matrix
- NVIDIA RTX 2080 ti (Peak GFLOPS: 13500)
- Naïve implementation
  - Elapsed: 16.865s
  - GFLOPS: 521
- Local reuse 1
  - Elapsed: 5.08
  - GFLOPS: 1728

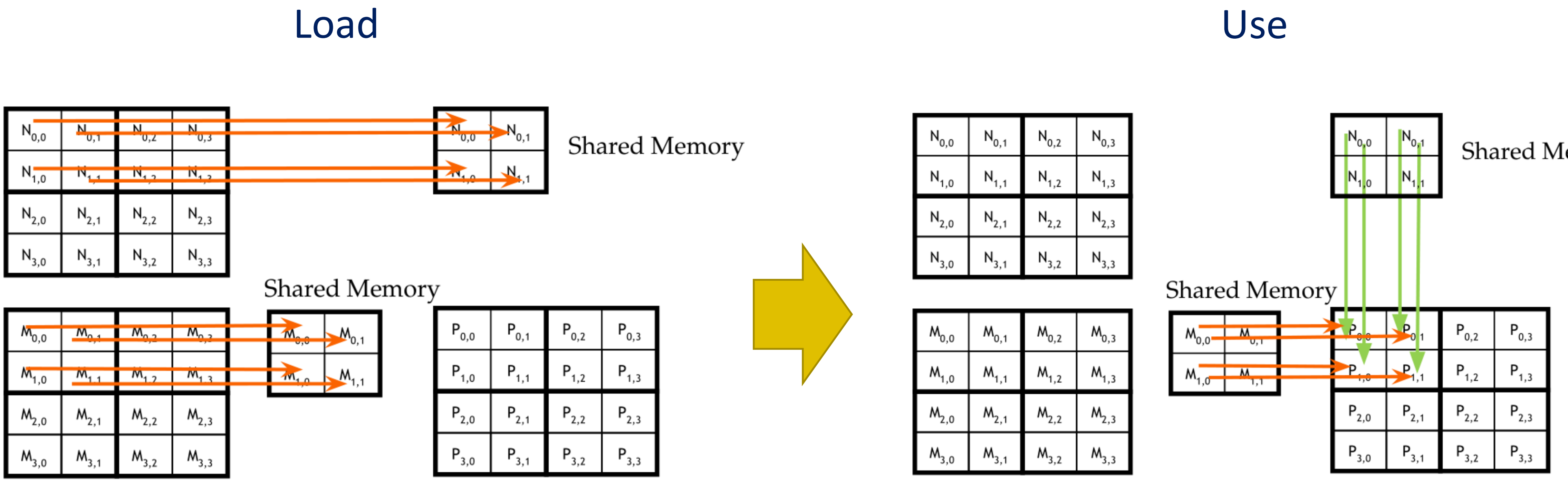


Attempt 3: Shared Memory

- Explicitly manage caches using __shared__
- Calculate result by adding N/BW sub-sums
- Sub-blocks will be used by all threads before discarded



Multithreaded Load To Shared Memory



Source: NVIDIA, UIUC GPU Teaching Kit

Attempt 3: Shared Memory

```
__global__ void MatrixMult2(float* a, float* b, float* c, int N) {
        __shared__ float ds_a[BLOCK_WIDTH][BLOCK_WIDTH];
        __shared__ float ds_b[BLOCK_WIDTH][BLOCK_WIDTH];

        int bx = blockIdx.x; int by = blockIdx.y;
        int tx = threadIdx.x; int ty = threadIdx.y;
        int Row = by * blockDim.y + ty;
        int Col = bx * blockDim.x + tx;

        float Pvalue = 0;
        for (int p = 0; p < N/BLOCK_WIDTH; ++p) {
                ds_a[ty][tx] = a[Row*N + p*BLOCK_WIDTH+tx];
                ds_b[ty][tx] = b[(p*BLOCK_WIDTH+ty)*N + Col];
                __syncthreads();
                for (int i = 0; i < BLOCK_WIDTH; ++i)Pvalue += ds_a[ty][i] * ds_b[i][tx];
                __syncthreads();
        }
        c[Row*N+Col] = Pvalue;
}
```

Wait until load is done for all threads

Wait until computation is done for all threads

Performance So Far

- 16,384 x 16,384 Matrix
- NVIDIA RTX 2080 ti (Peak GFLOPS: 13500)
- Naïve implementation
  - Elapsed: 16.865s, GFLOPS: 521
- Local reuse 1
  - Elapsed: 5.08s, GFLOPS: 1728
- Shared memory
  - Elapsed: 3.94s, GFLOPS: 2229

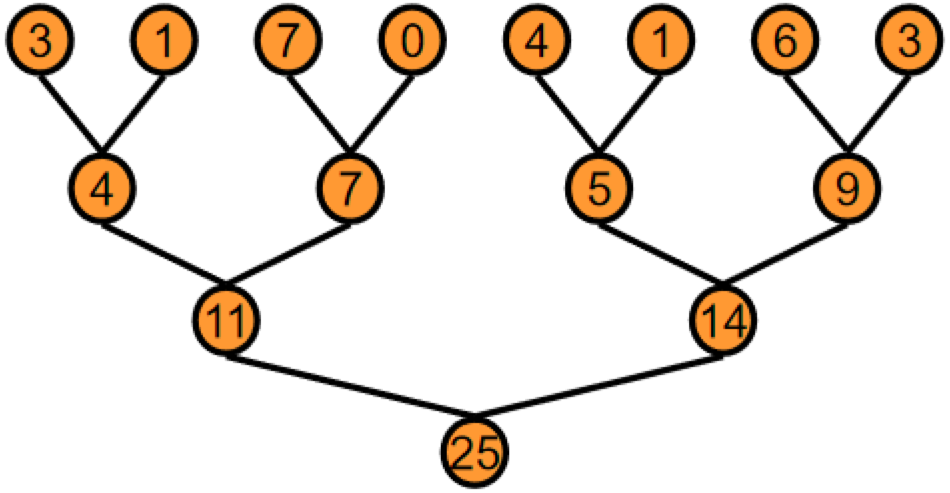
Block Size Considerations

- More re-use with larger blocks!
- With 16x16 blocks (256 threads)
  - 512 word loads from memory
  - 256 * (2*16) = 8,192 FLOPs
  - 16 FLOP per load
- With 32x32 blocks (1024 threads)
  - 1024 word loads from memory
  - 1024 * (2*32) = 65,536 FLOPs
  - 32 FLOP per load

Unfortunately, threads per block limited to 1024

Application 2: Parallel Reduction

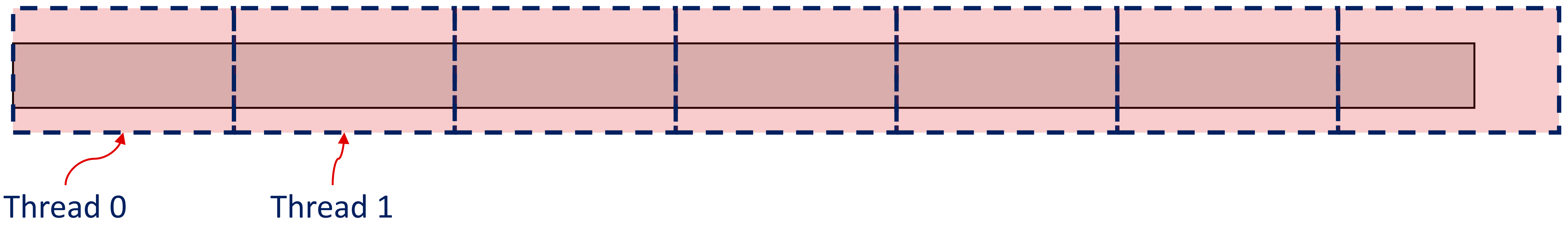
- Combines an array of elements and produces a single result
  - E.g., adding all values in an array, finding maximum, calculating average, …
- If the operation is associative, i.e., (A+B)+C == A+(B+C), calculation can be parallelized



Source: Mark Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA Developer Technology, 2007

How To Best Allocate Work To Threads?

- Straightforward method: divide blocks of work across threads
- Will this be efficient?
  - Warp affinity of algorithm
  - Good data access patterns, etc?
- How many threads should we spawn?
  - As many threads as cores: Too little threads… Main memory latency not hidden! ☹
  - Too many threads: Is there any downsides to this?



Method 0: Consecutive work blocks

- Each kernel run will reduce data size to blocks*threads
  - Must run iteratively until reduced to 1
  - How many threads, how many blocks? Too small: too many iterations!
  - Let’s fix threads per block to 1024 (max for this architecture)
- Peak performance when ~64 elements per thread
  - ~40ms for 2^{30} elements
  - Is this good?

```
__global__
void reduce_consecutive(int *g_idata, int *g_odata, unsigned int N) {
    extern __shared__ int sdata[];

    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int threadcnt = gridDim.x*blockDim.x;
    unsigned int workcnt = (N+threadcnt-1)/threadcnt;
    unsigned int i = workcnt * idx;

    int psum = 0;
    while ( i < N && i < workcnt*(idx+1) ) {
        psum += g_idata[i];
        i++;
    }
    g_odata[idx] = psum;
}
```

Our Goal: Memory Saturation

- Reduction is an O(N) problem, ideally reading each element exactly once
- Not much computation per memory, so likely memory bound
  - RTX 2080 ti's GDDR6 memory has peak bandwidth of 616 GB/s
  - We want to reach this utilization
  - E.g., 2^{30} elements = 4 GB, ideally **6 ms**

- Let's follow the guidelines in NVIDIA's "Optimizing Parallel Reduction in CUDA," NVIDIA Developer Technology, 2007

Method 1: Interleaved Addressing

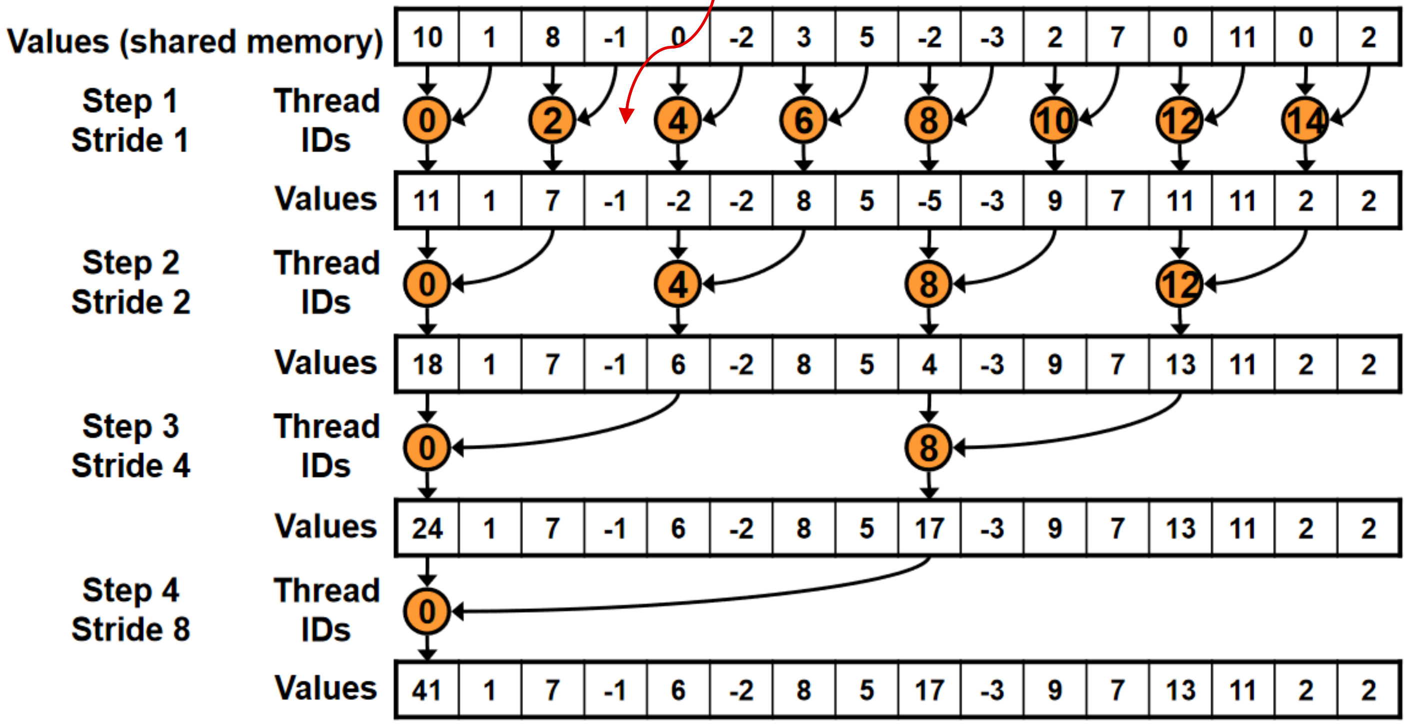
- ❑ Each block of 1024 threads reducing 1024 elements to 1
  - Use shared memory!
- ❑ 47 ms, 91 GB/s

```
__global__
void reduce0(int *g_idata, int *g_odata, int N) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

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

Where are all the odd threads?



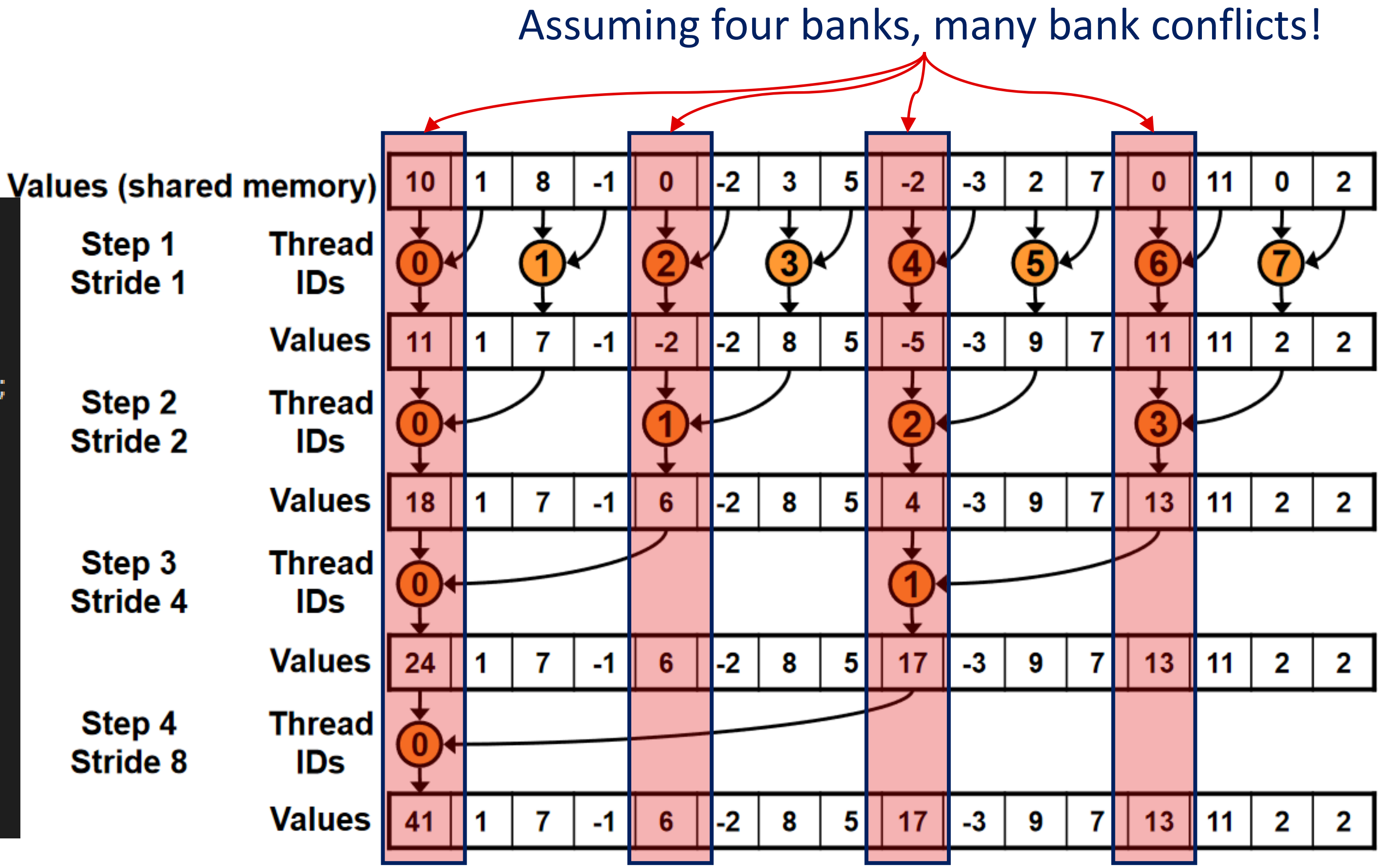
Method 1b: Better Thread Allocation

- More threads are doing work!
- 33.41 ms, 128 GB/s

```
__global__
void reduce1(int *g_idata, int *g_odata, int N) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;
        if (index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```



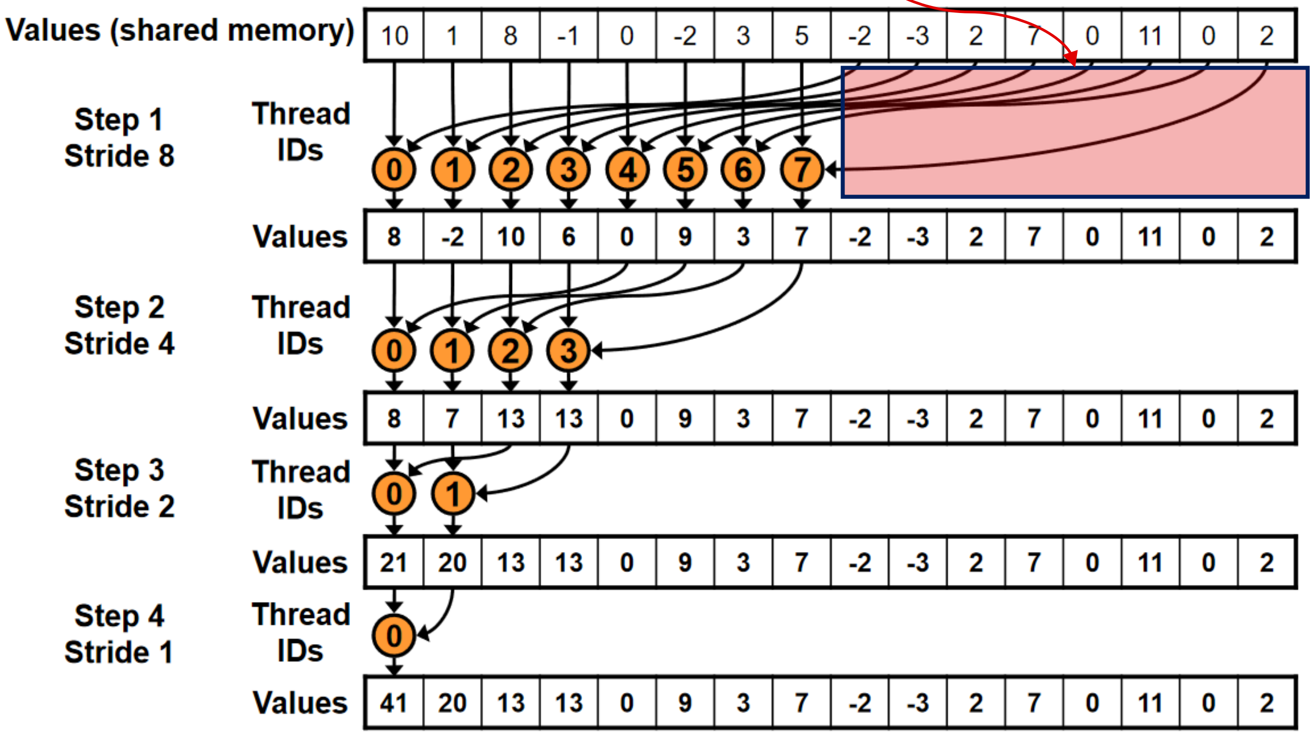
Method 2: Sequential Addressing

- Change thread mapping to group to lower elements
- Consecutive addresses have no bank conflict
- 29.76 ms, 144 GB/s

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

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

For N threads, N/2 threads are never used!



Method 3: More Work per Thread

- Instead of 1024 elements per 1024 threads, 2048 elements!
- 15.36 ms, 280 GB/s

- Q1: What if we use the same method for all previous attempts?
  - Interleaved: 47 ms -> 24.35 s, Better thread: 33.41 -> 17.35 ms
- Q2: Can we take this further? More work per thread?

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
```

```
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i]+g_idata[i+blockDim.x];
__syncthreads();

for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
```

Method 4: Back To Work Blocks Per Thread

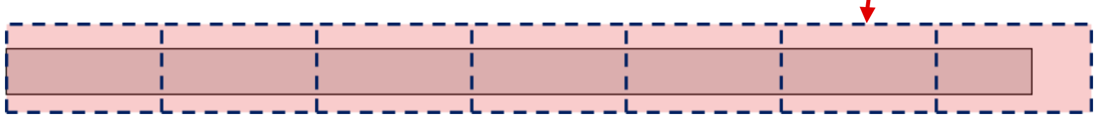
- But this time, use method 2 to reduce within a block
  - Lots of work per thread,
  - Small result set per iteration
  - Best of both worlds?
- How many blocks?
  - 8,192 blocks: 40 ms
  - 32,768 blocks: 28.50 ms
  - 131,072 blocks: **8.52 ms! 504 GB/s!**
  - 524,288 blocks: 17.96 ms…

```
__global__
void reduce7r(int *g_idata, int *g_odata, unsigned int N) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int idx = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int threadcnt = gridDim.x*blockDim.x;
    unsigned int workcnt = (N+threadcnt-1)/threadcnt;
    unsigned int i = workcnt * idx;

    sdata[tid] = 0;
    while ( i < N && i < workcnt*(idx+1) ) {
        sdata[tid] += g_idata[i];
        i++;
    }
    __syncthreads();

    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```



Method 4: Why?

- Most likely, random access issue in DRAM
  - Many threads scheduled potentially out of order causes random access
  - DRAM isn't really random access!
  - We will get into details later
  - Bottom line: Consecutive access is faster when within same page, of multiple KBs
- Analyzing performance
  - 131,072 blocks -> 32 KB working set within fast random access range
    - Each block is scheduled sequentially. No interleaving between blocks on same SM!
  - Less blocks -> Larger working set per block -> Random access penalty
  - More blocks -> Smaller work per thread -> Performance penalty

Method 5: Consecutive Memory Access

- Set stride to total number of threads in grid
  - Consecutive threads access consecutive addresses
  - At least, threads in a warp always access contiguous addresses at once
- Reliably high performance!
  - 256 blocks: 9.83 ms
  - 1024 blocks: 8.3 ms
  - 8192 blocks: **7.8 ms, 550 GB/s**
  - 65,536 blocks: 7.9 ms
  - 262,144 blocks: 11.52 ms

```
while ( i < N && i < workcnt*(idx+1) ) {
    sdata[tid] += g_idata[i];
    i++;
}
__syncthreads();
```

```
unsigned int gridSize = blockDim.x*gridDim.x;
```

```
while (i<N) {
    sdata[tid] += g_idata[i];
    i += gridSize;
}
__syncthreads();
```

Some More Approaches?

- The NVIDIA guide suggests loop unrolling when active threads become less than 32
  - Within a warp, no __synchthreads needed!
  - Adding an if statement to __syncthreads also adds overhead
- On modern chips, this changes measures pretty negligible, so omitted

- 616 GB/s is ~150 GOPS…
  - Remember peak computation is 13,500 G**FL**OPS
  - Very much bandwidth bound!

Application 3: Option Pricing

- Options in Computational Finance:
  - In finance, a contract giving the buyer of an asset the right (but not the obligation) to buy or sell and underlying asset at a specified price or date.
  - Question: How much should I pay for a particular option?

Option Pricing

Black-Scholes Equation

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

Geometric Brownian Motion in Finance

$$dS_t = \mu S_t dt + v S_t dW_t$$

What we want

Random variable

"Monte Carlo Method"
Simulate massive amount of instances and average return

Option Pricing

- No memory usage
  - Not even shared memory
  - Completely computation bound
- 537x Performance vs. 1 Thread

- Assuming GTX 1080
  - 2560 CUDA cores
  - Close to linear scaling

```
****************** INFO ******************
Number of Paths: 5000000
Underlying Initial Price: 100
Strike: 100
Barrier: 95
Time to Maturity: 1 years
Risk-free Interest Rate: 0.05%
Annual drift: 0.1%
Volatility: 0.2%
****************** PRICE ******************
Option Price (GPU): 8.52652
Option Price (CPU): 8.51663
******************* TIME ******************
GPU Monte Carlo Computation: 25.1978ms
CPU Monte Carlo Computation: 13530 ms
******************* END ******************
```

Cho et. al., “Monte Carlo Method in CUDA,” 2016

Questions?