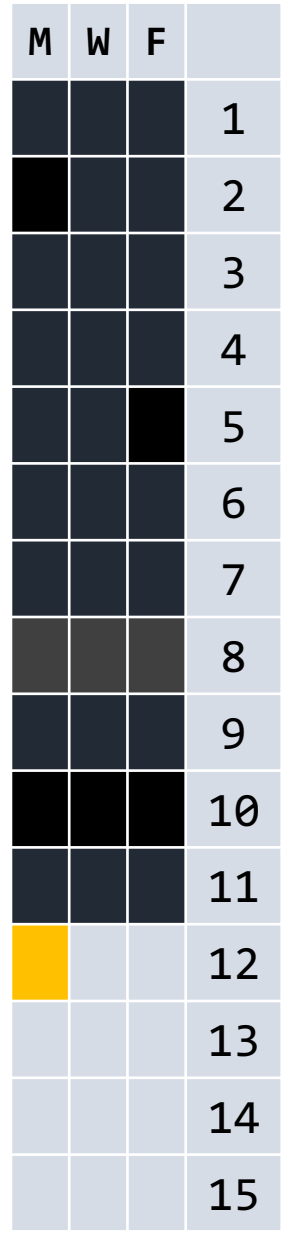
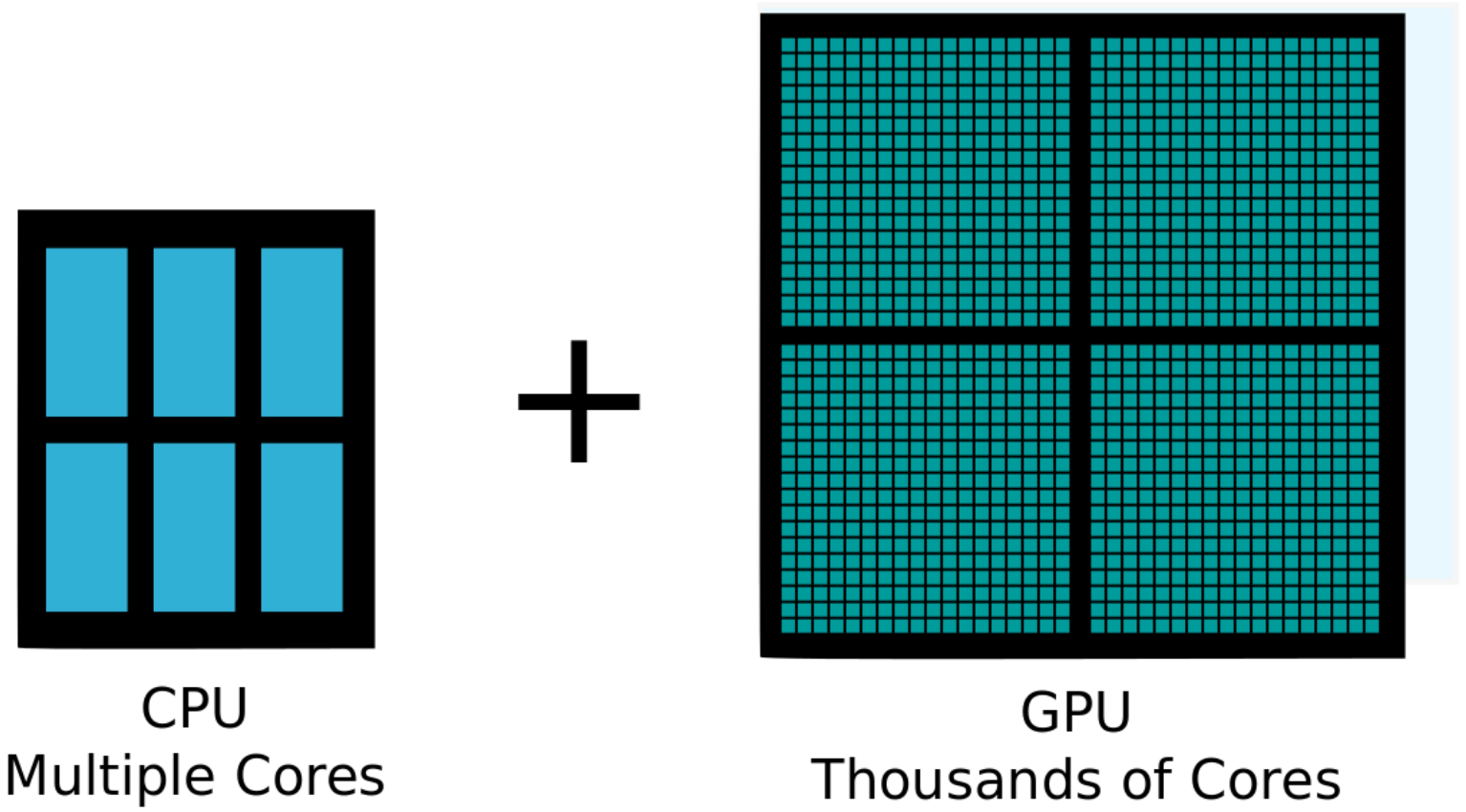
CMOR 421/521:
GPU computing



Why are GPUs so popular for computing?

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

GPUs vs CPUs (oversimplified)



https://www.cherryservers.com/blog/gpu-vs-cpu-what-are-the-key-differences

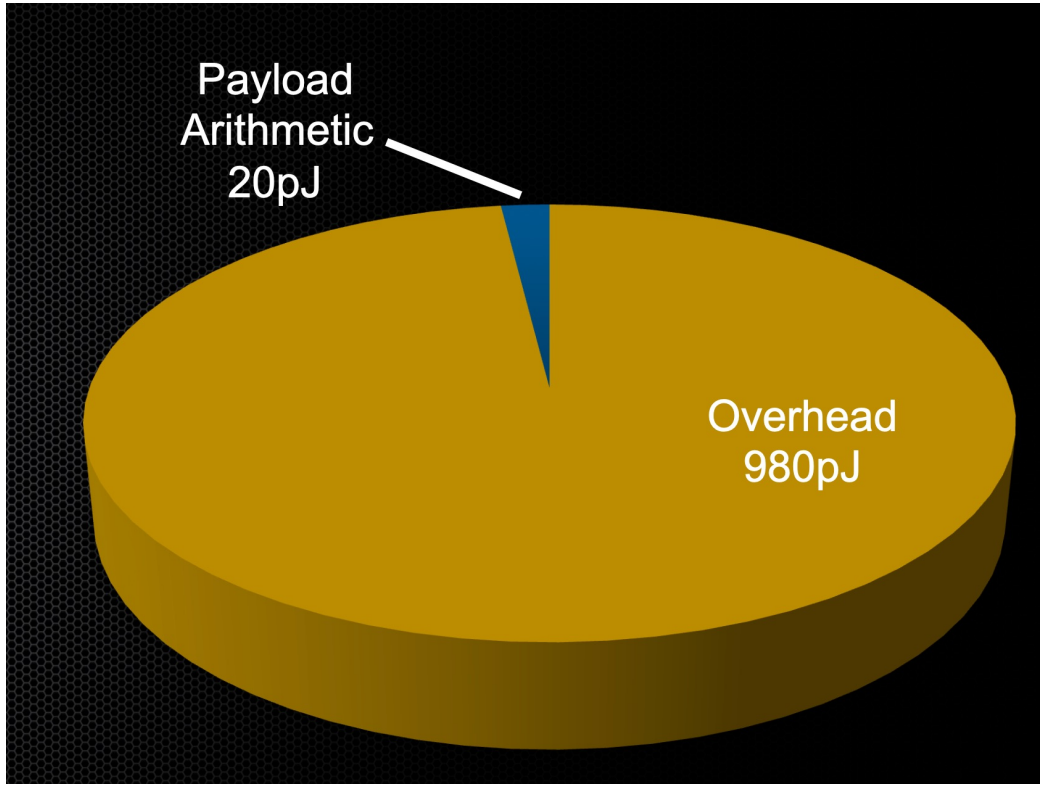
GPUs vs CPUs

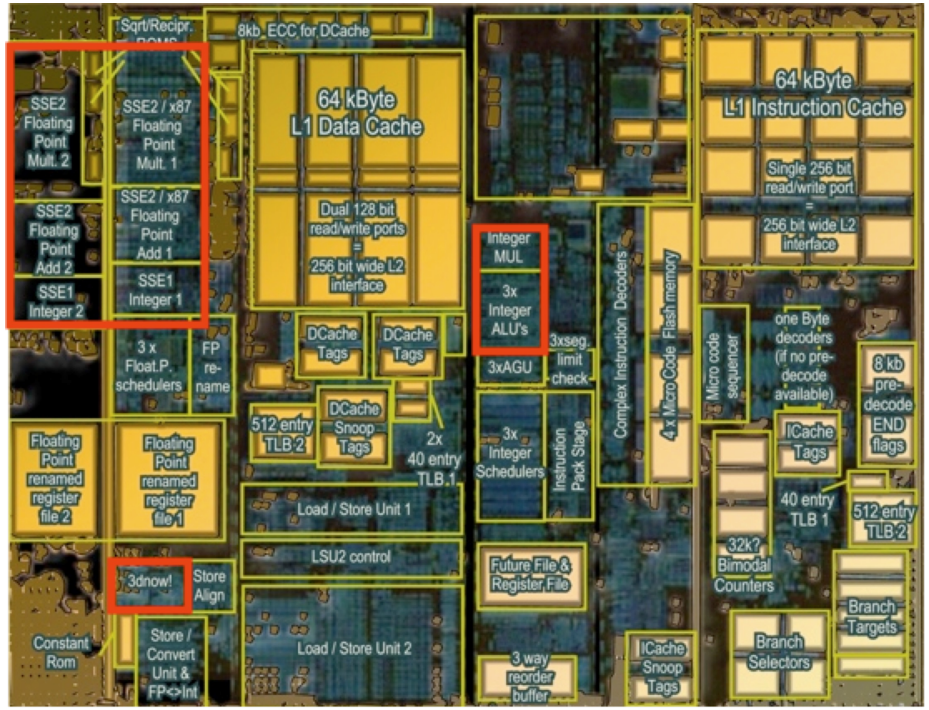
- CPUs designed to execute *general* tasks in serial as quickly as possible.
  - Low latency – focus on each task being responsive.

- GPUs designed to … render graphics
  - Specialized operations, simpler architecture
  - Designed for high throughput: millions of pixels, rapid refresh rate

Why did GPUs become popular?

- Most CPU energy and space is not used for computing





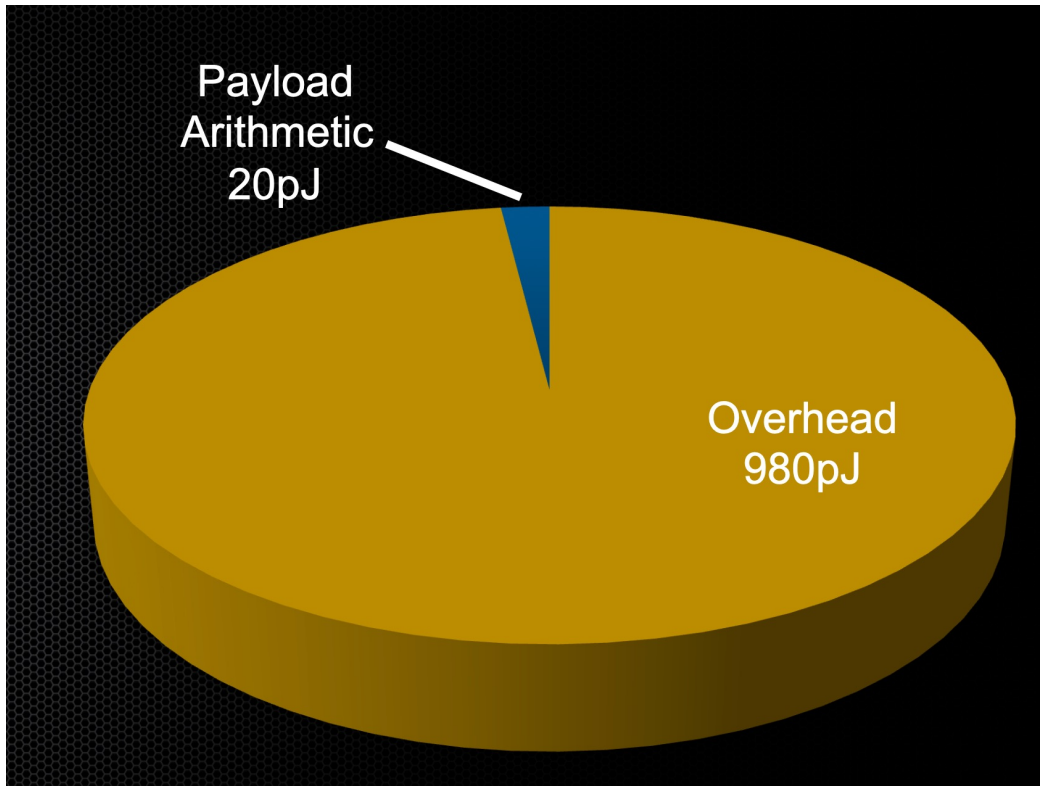
chip-architect.com

AMD K7 “Deerhound”

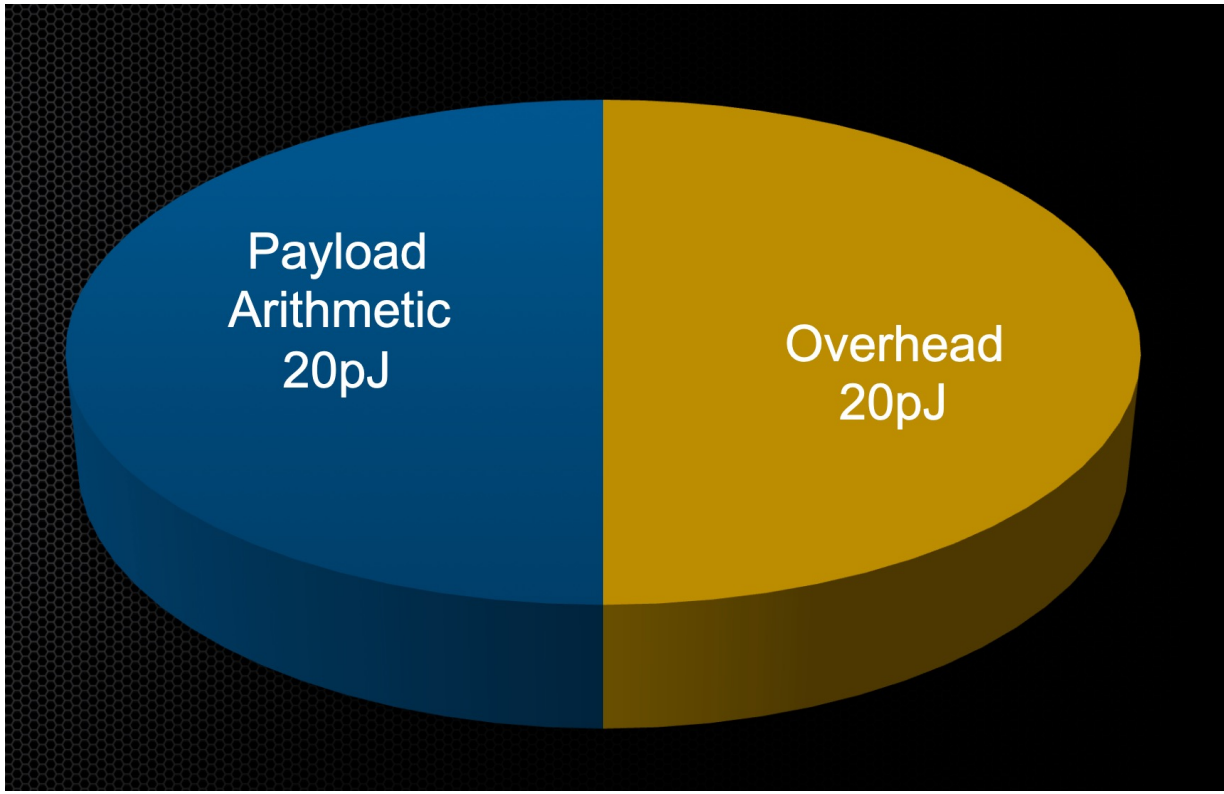
Figure from Aydin Buluc, Bill Dally (2015)

Why did GPUs become popular?

- Most CPU energy and space is not used for computing



CPU



GPU

Figure from Aydin Buluc, Bill Dally (2015)

Why did GPUs become popular?

- 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, ...

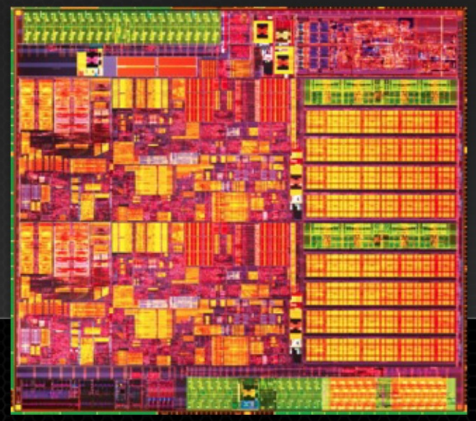
https://spectrum.ieee.org/nextgeneration-supercomputers

Why did GPUs become popular?

CPU
1690 pJ/flop

Optimized for Latency

Caches

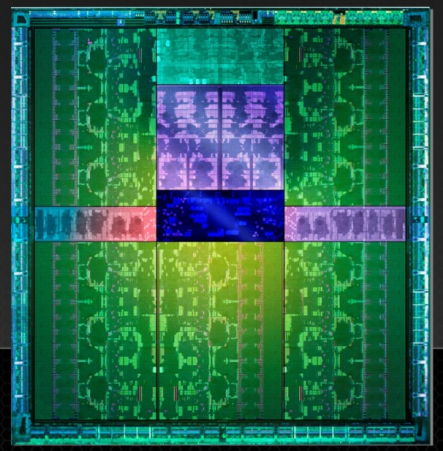


Westmere
32 nm

GPU
140 pJ/flop

Optimized for Throughput

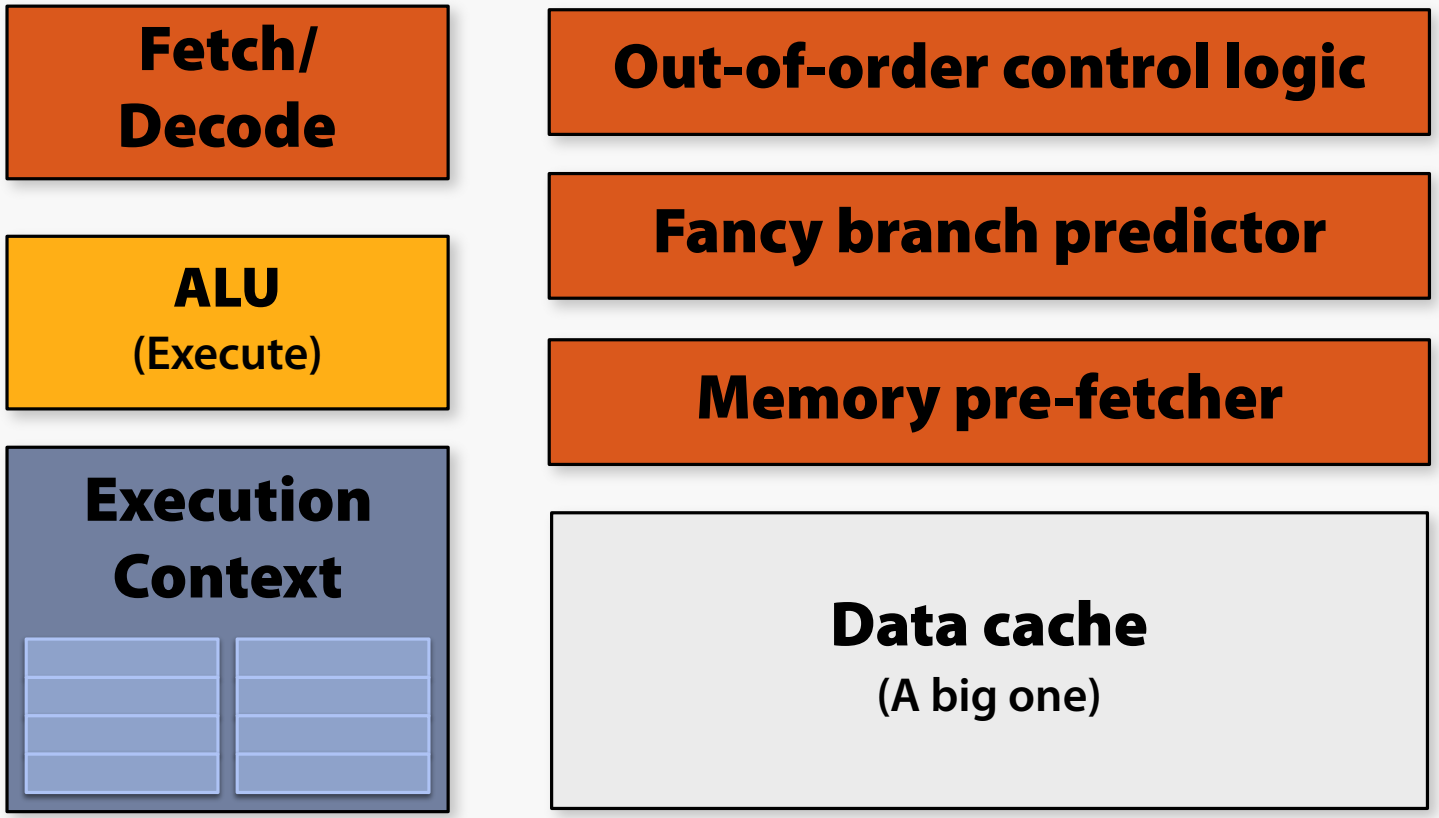
Explicit Management
of On-chip Memory



Kepler
28 nm

Figure from Bill Dally (2015)

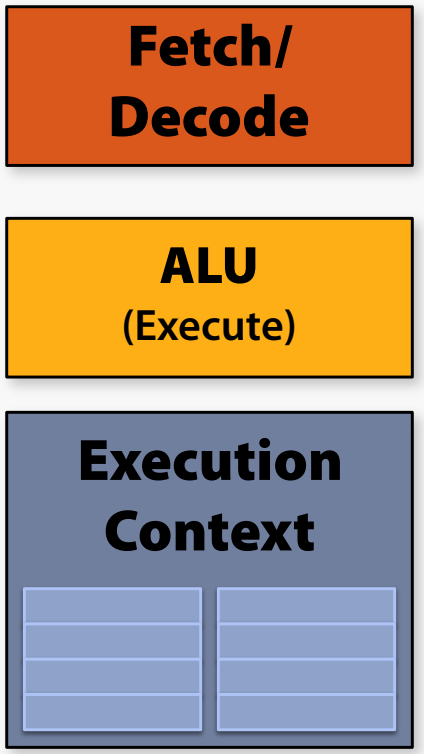
CPU architecture



- Left: hardware used for actual computations
- Right side: hardware introduced to minimize latency:
  - Pack instruction cycles
  - Avoid memory latency
- Cache = L1, L2, etc.

Figure from Aydin Buluc

From CPUs to GPUs

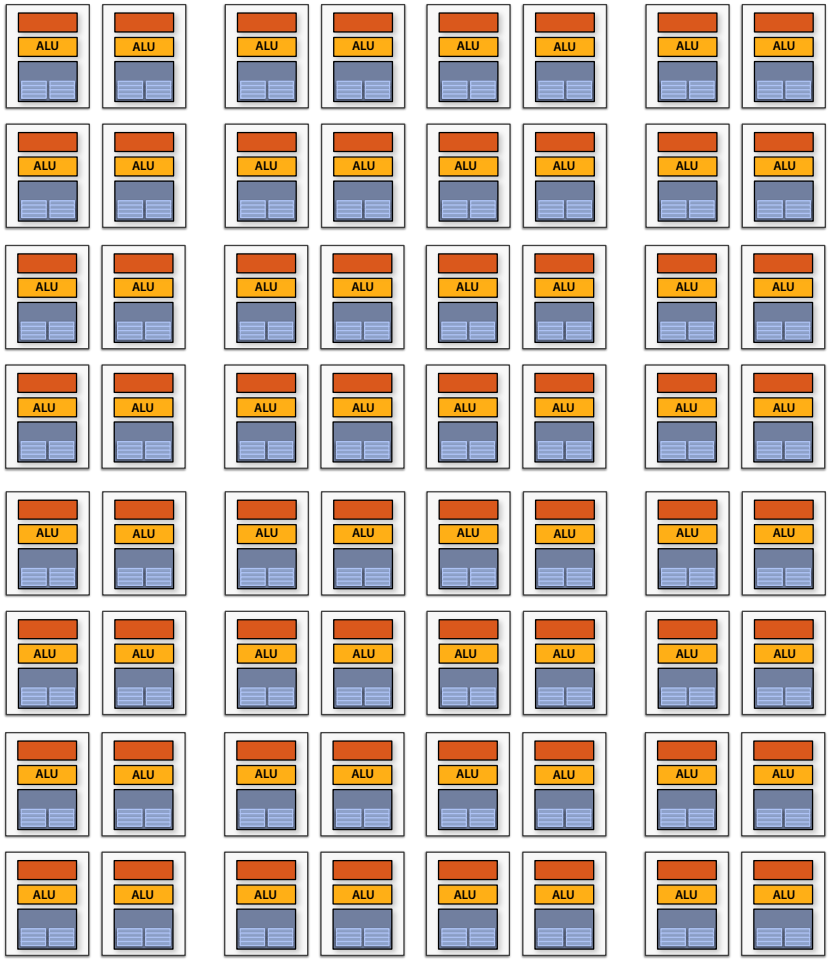


Hardware Idea #1:

Remove components that focus on making a single instruction stream run fast. These components identify parallelism (ILP, data parallelism, etc), but also consume energy

Figure from Aydin Buluc

From CPUs to GPUs



Hardware Idea #2:

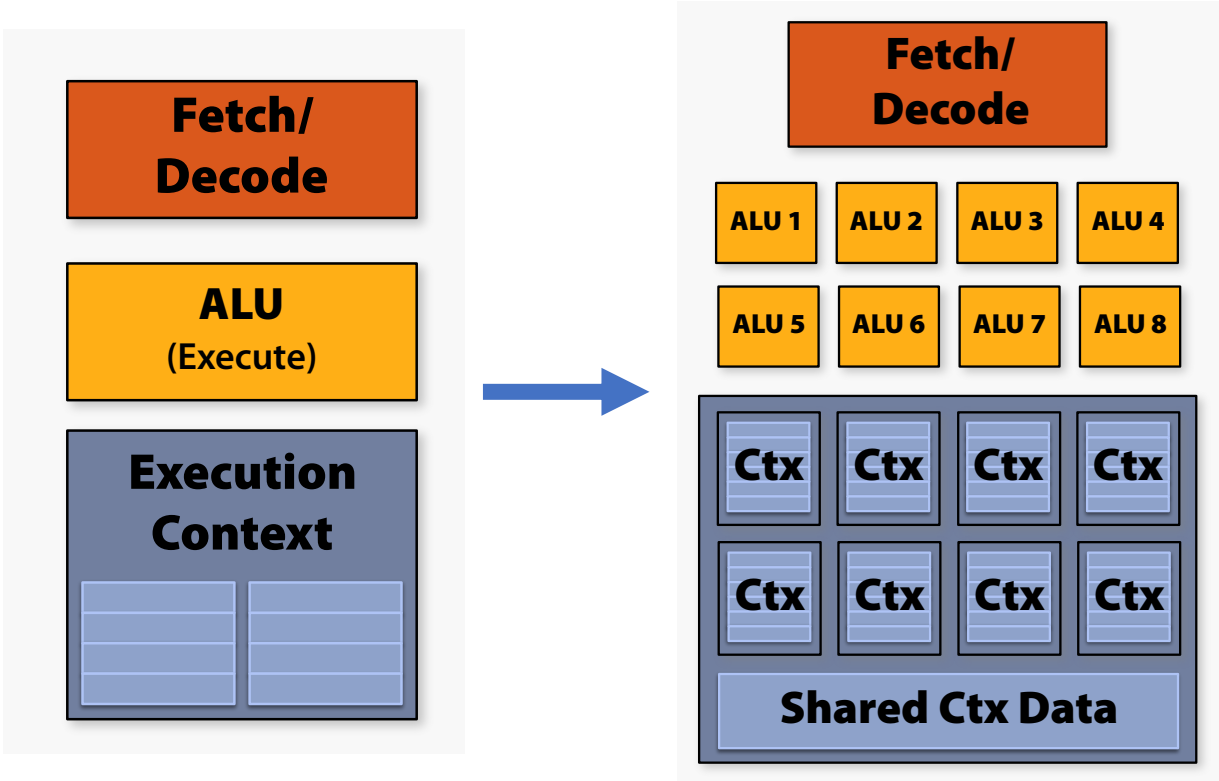
- A larger number of (smaller and simpler) cores
- Encourages data parallelism (same operation applied to multiple instances of data)

Figure from Aydin Buluc

From CPUs to GPUs

- 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 execute the same instruction simultaneously.

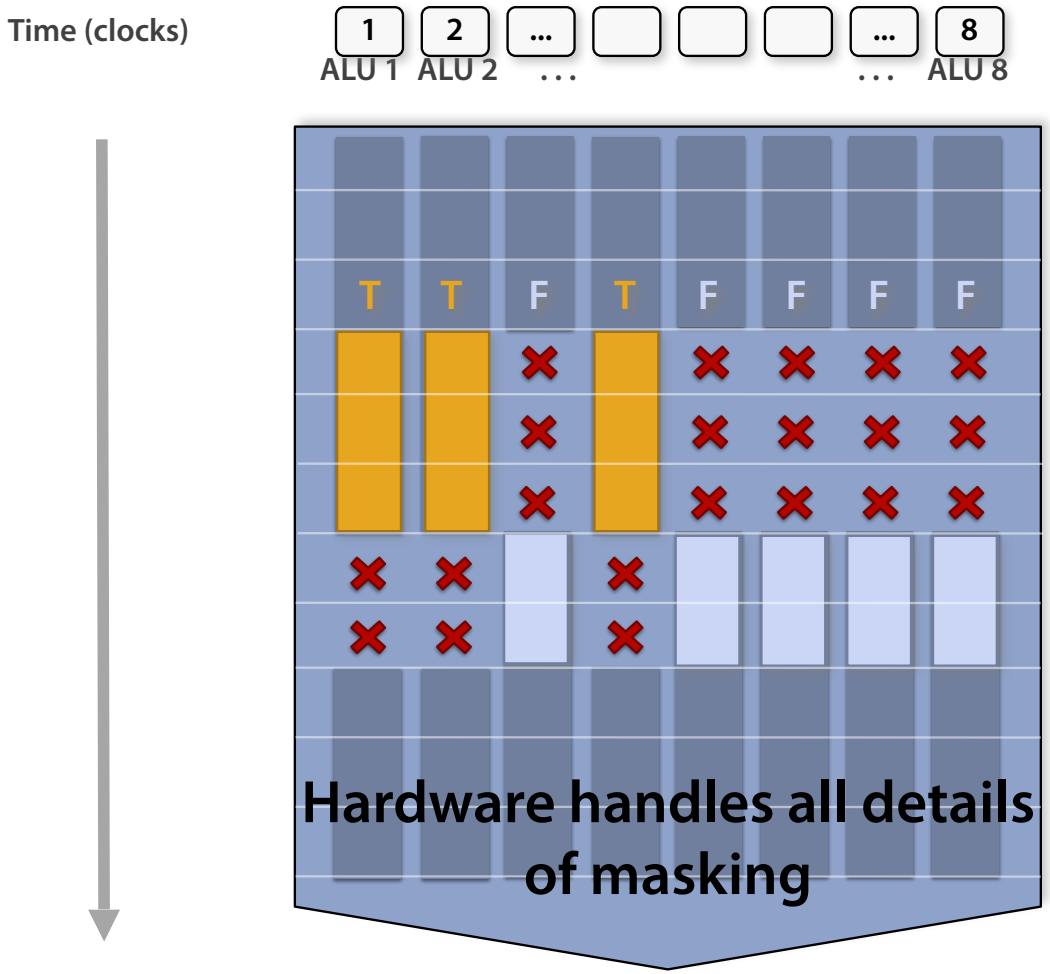
Hardware Idea #3:
Share the instruction stream



CPU GPU

Figure from Aydin Buluc

From CPUs to GPUs

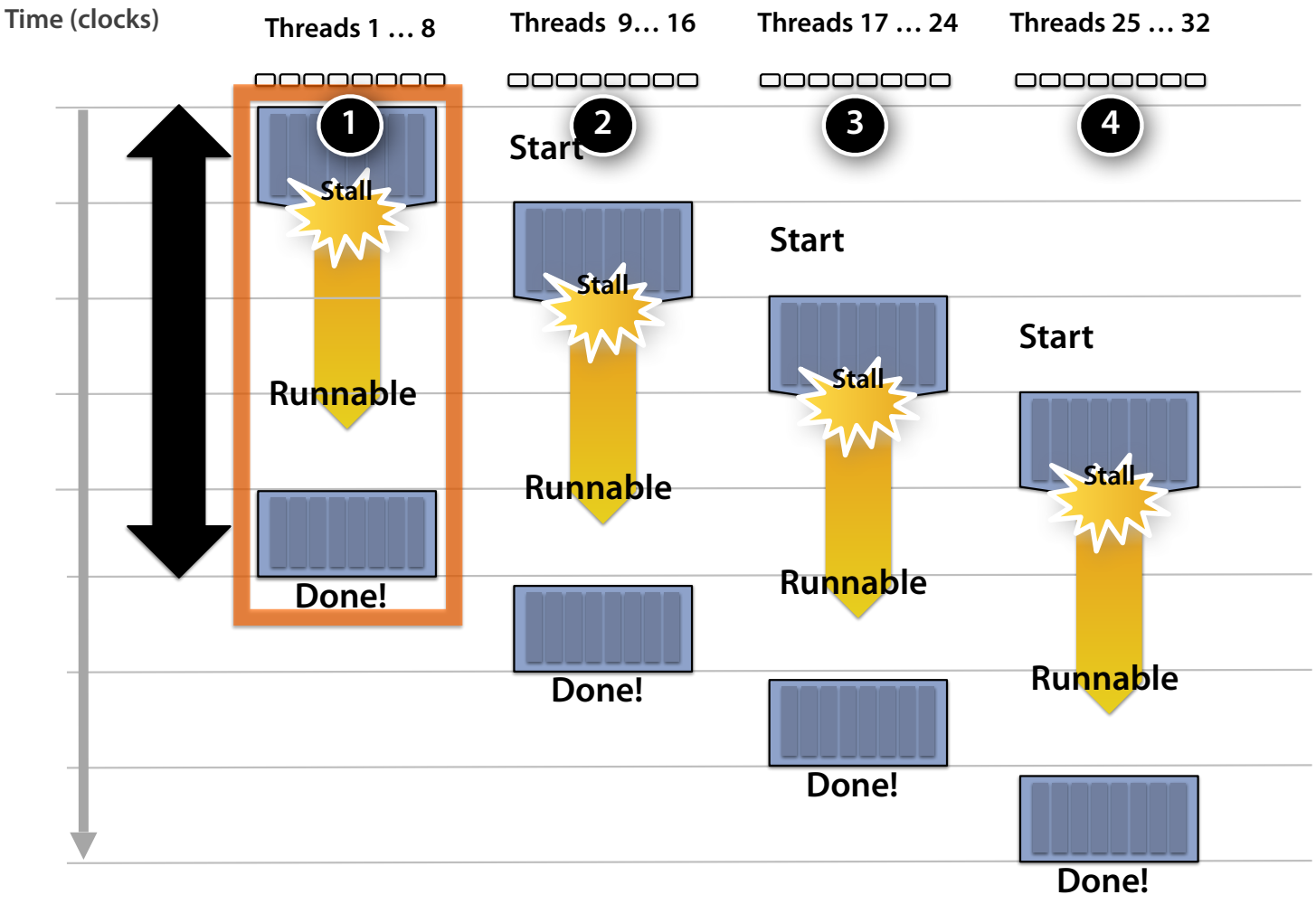



```
<unconditional
 program code>

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:

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

From CPUs to GPUs

If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?

-Seymour Cray

“The Father of Supercomputing”

GPU caveats

- **GPUs aren't that "smart"**
  - They are highly efficient/optimized for a certain kind of problem -> they lack the generalizability of CPUs
- They are quite specialized in comparison to CPUs
  - Think of thread branching…
  - "A well used system is never idle…"
- **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

Disclaimer

- **My familiarity with GPUs is in the context of scientific computing, not at all for graphics**

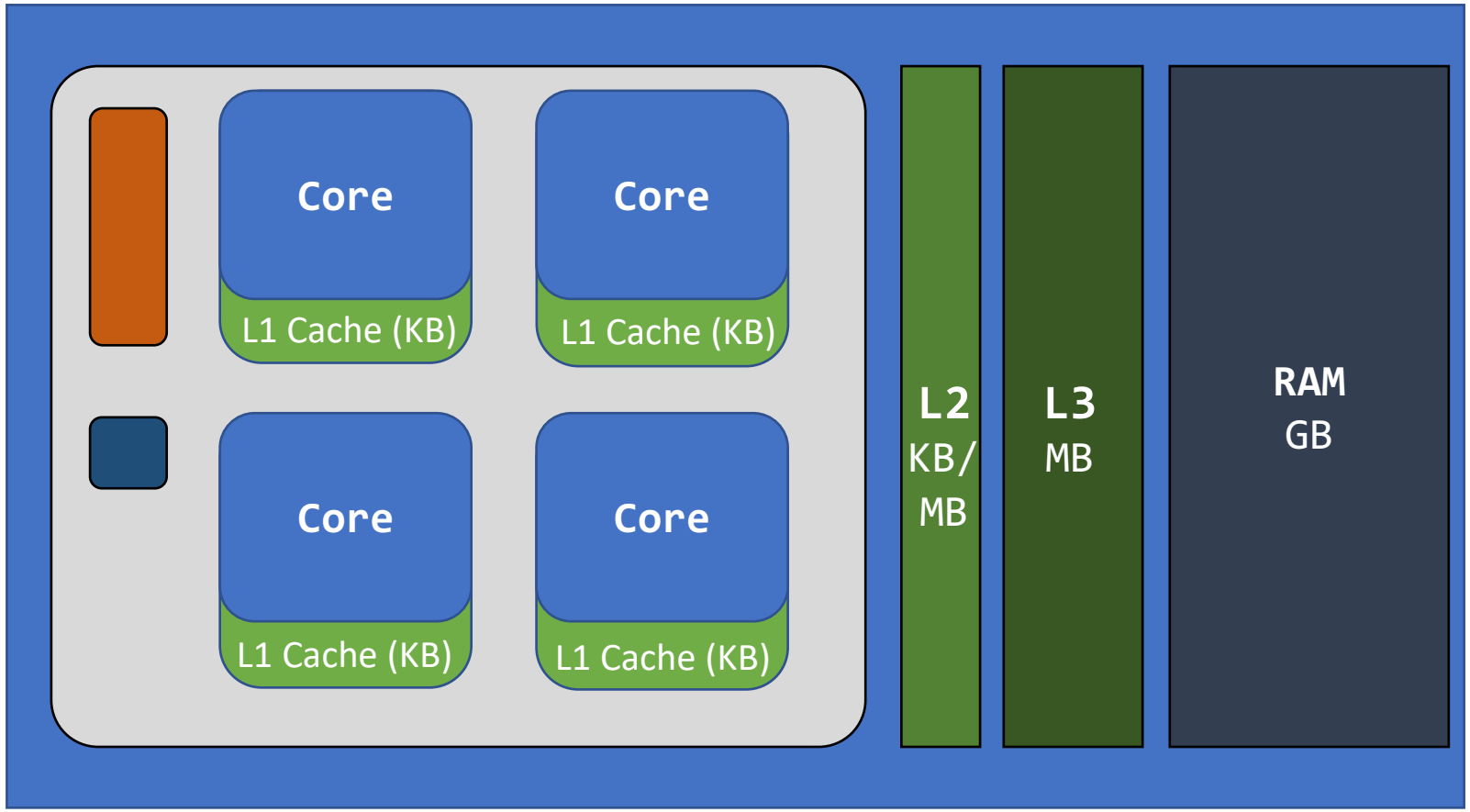
- There are several GPU (and GPU-like) manufacturers
  - Nvidia, AMD, Intel, Apple …
  - They can have different terminology for the same things

- **We are using Nvidia's terminology**

CPU Architecture

CPUs: Trade specialization for robustness

This part is truly the CPU

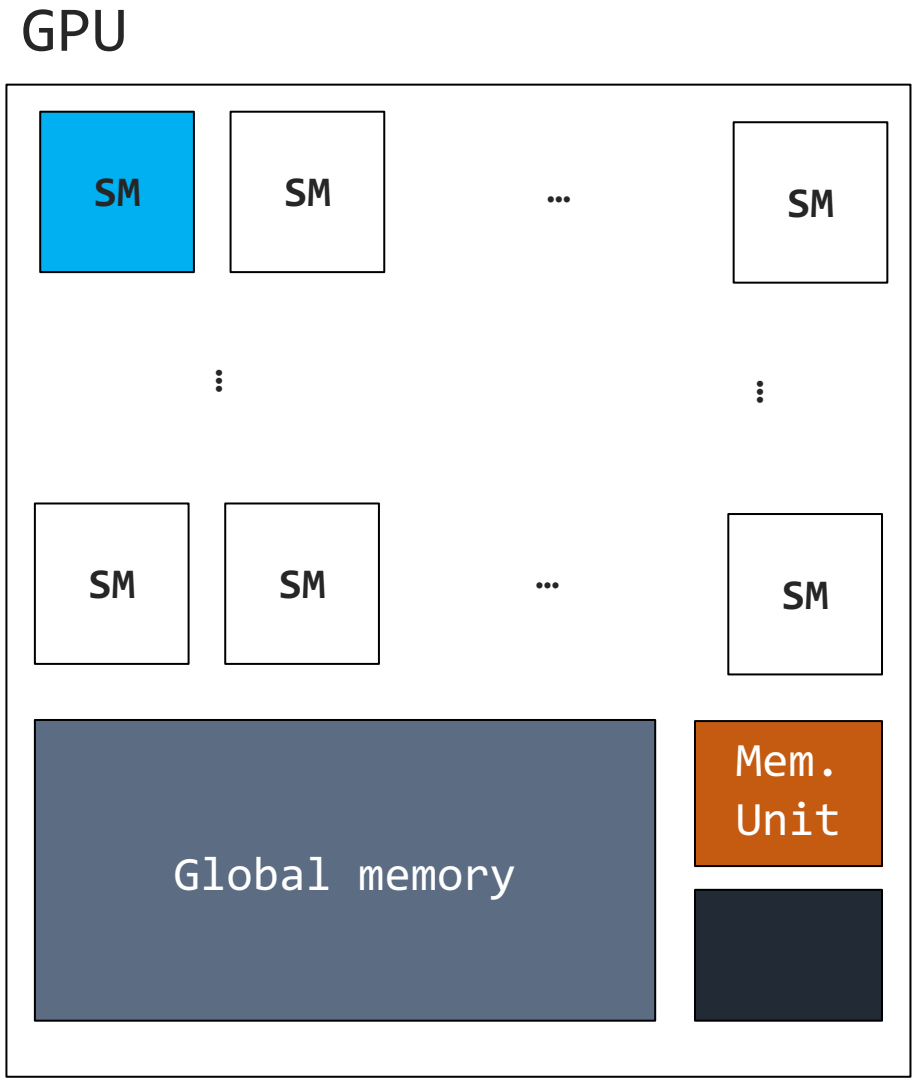


This part can be swapped out

GPU Architecture

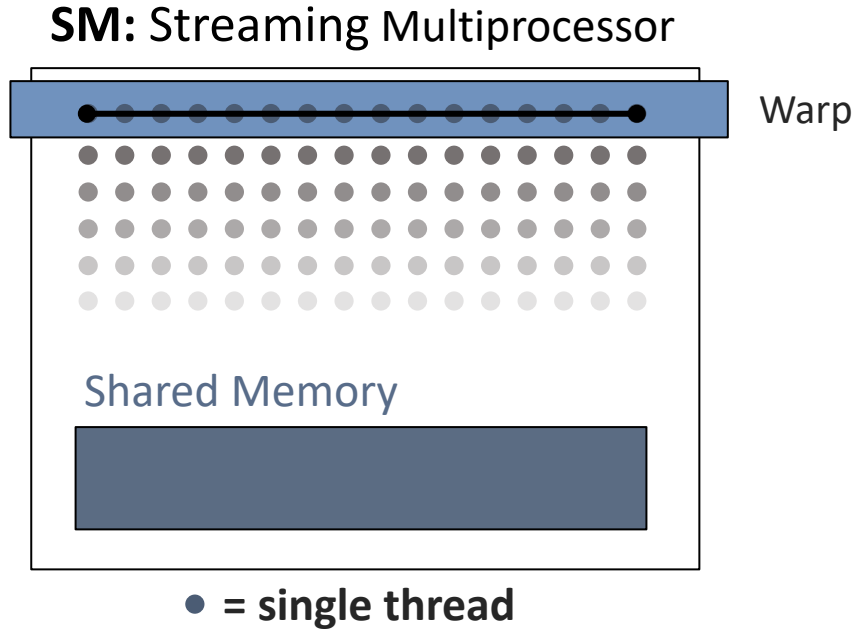
GPU's have parallelism on top of parallelism

- GPUs are made up of
  - *Streaming multiprocessors*
  - Global memory (think RAM)
  - **Memory (copy) unit**
  - Other stuff
- Streaming multiprocessors are what process commands



GPU architectures: some numbers

- Tesla K80 (2014, NOTS):
  - 2 K40 GPU glued together
  - Peak performance: 1.37 TFLOPS (double precision), 4.1 TFLOPS (single precision)
  - Bandwidth: 480 GB/s
- Tesla P100 (2020):
  - Peak performance: 4.7 TFLOPS (double precision), 9.3 TFLOPS (single precision)
  - Bandwidth: 720 GB/s



For comparison, NOTS CPUs:

- Peak performance: 166.4 GFLOPS
- Bandwidth: about 60 GB/s

GPU Architecture: some numbers

CPU (Skylake)

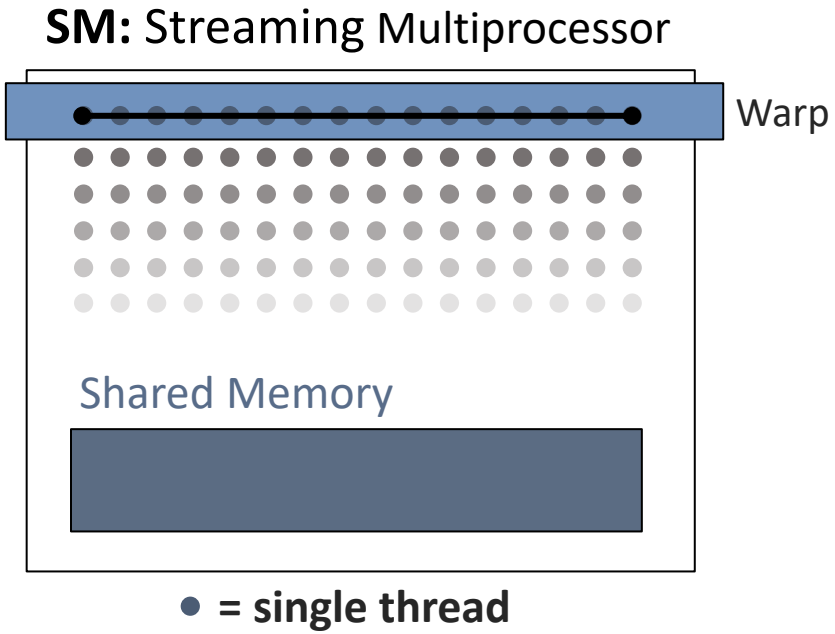
- 40 cores (2 20-core chips)
- 2 threads each
- 2 x AVX-512 vectorization, so 2x8 in double precision
- ~640 way parallelism (40*2*8)

GPU (V100)

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

GPU Architecture

- 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
  - Limit on total number of active warps

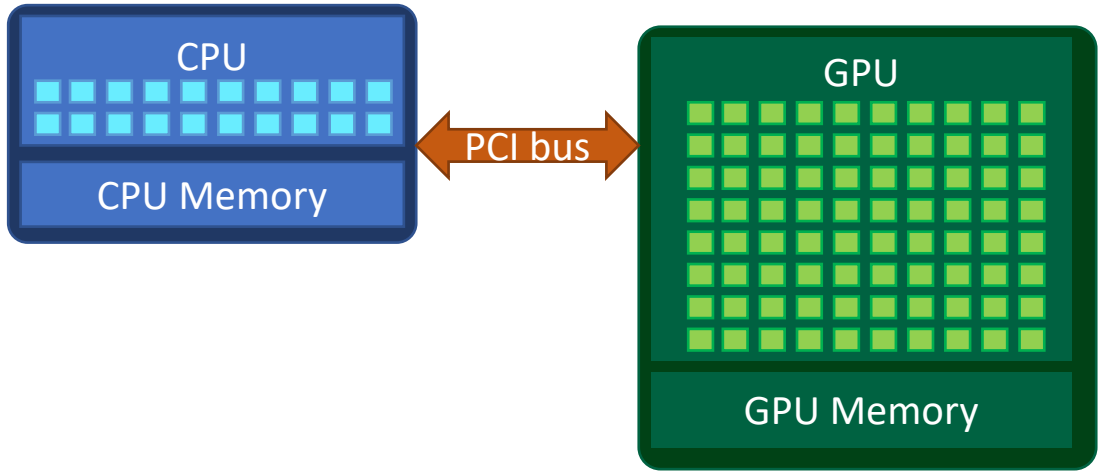


GPUs: massively parallel

- Due to their immense computing power, we are **more likely to be bottlenecked by communication/memory accesses on a GPU** than computation
- Thus, memory hierarchy is **extremely** important
  - Accessing CPU memory from the GPU is *very slow*
  - Global memory (on the GPU) is slow
    - *Contiguous* memory access on CPU → *coalesced* memory access on GPU
  - (SM) Shared memory (local to a SM) is fast
  - Register memory (local to a thread) is fastest

Host-device paradigm

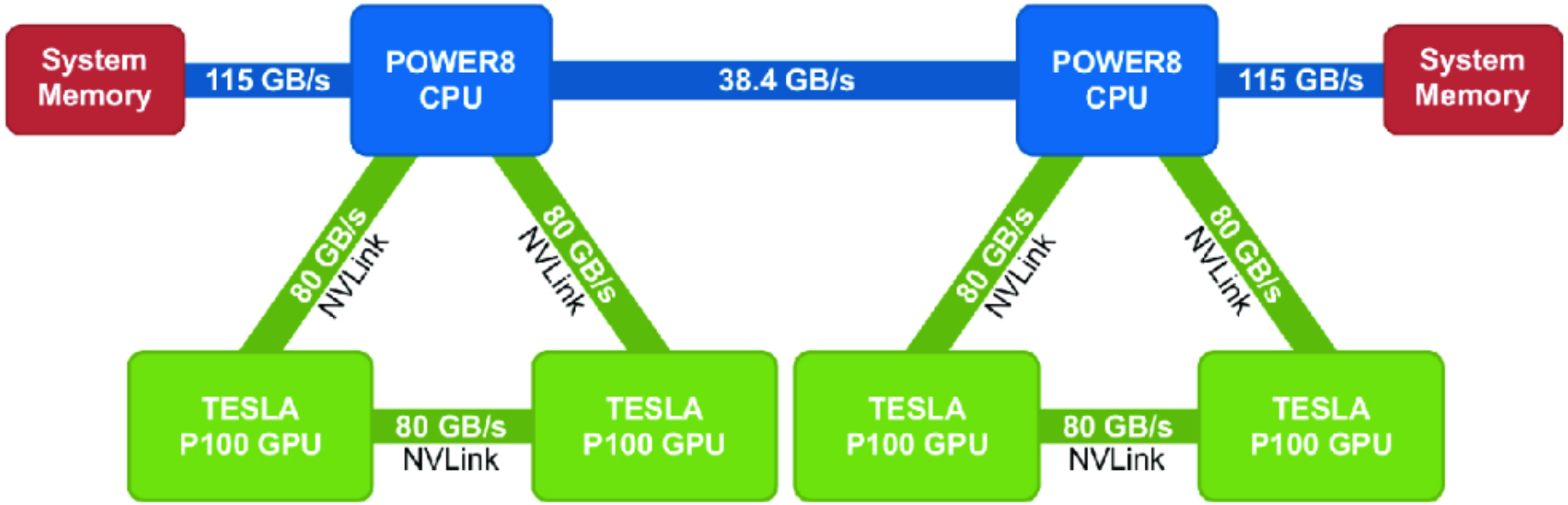
- CPUs may not have a GPU, or may have multiple GPUs
- **GPU (device) always has a CPU (host) which we interact with**
- Think of device/host as separate memory spaces
  - Memory transfer from CPU to GPU is very slow (through PCIE)
  - Maybe an exception for some "unified memory"



Example Real Cluster: Power8

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



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 dominant, everyone else has been playing catch-up to NVIDIA since ~2010 (e.g., OpenCL, OpenACC, ROCm/HIP, etc)

Running on NOTS

- srun --pty --partition=interactive --**reservation=cmor421** --ntasks=1 --mem=1G --gres=gpu:1 --time=04:00:00 $SHELL
  - Can also use “--partition=scavenge”
  - The “cmor421” reservation should reduce wait time.

- Our NOTS partition needs CUDA/7.0.28 to run! K80 GPUs are limited to older versions of CUDA (current version 12.x.x)

How Do We Program a GPU?

- There are several tools:
  - CUDA (made by Nvidia)
  - OpenCL
  - OpenACC
  - SYCL
  - etc
- We will use CUDA

Writing a GPU program

- Simple example: add_vectors.cu
- Use “esc-x c-mode” in emacs to get syntax highlighting

Memory: CUDA Kernels and CPU

- **Programmers launch CUDA kernels from the CPU, but CUDA kernels are run on the GPU**
- CUDA is a subset of C/C++, but not all functionality is efficient in CUDA
  - If you make a call to “new” in your main function, it will allocate on the CPU
  - If you allocate using “new” in a kernel, it will be extremely slow (every thread will try to allocate)

Main CUDA Construct: CUDA Kernels

- **CUDA kernels are functions to be run on the GPU**
  - They are just like a regular function + CUDA keywords
  - **The rest of the program is run by the CPU**
- When CUDA kernels are invoked (called), thread block and grid dimensions are specified
  - A thread block reflects the number of threads you want that call to the kernel to be run with
  - The thread block can be 1D, 2D, or 3D…
  - Grid dimensions specify how many blocks to run

CUDA Kernel Declaration and Invocation

```
// Declaring a CUDA kernel
// Note: it is two underscores, not one
__global__ void func(int arg1, double arg2) {
…
}

int main() {
    int x;
    double y;
    …
    // Invoking (calling) the kernel
    int threadsPerBlock = 32;
    func<<<num_blocks, threadsPerBlock>>>(x, y)
}
```

CUDA Kernel Declaration and Invocation

```
// Declaring a CUDA kernel
// Note: it is two underscores, not one
__global__ void func(int arg1, double arg2) {
…
}

int main() {
    int x;
    double y;
    …
    // Invoking (calling) the kernel
    dim3 threadsPerBlock(NT_x, NT_y)
    func<<<num_blocks, threadsPerBlock>>>(x, y)
}
```

CUDA Kernel Declaration and Invocation

```
// Declaring a CUDA kernel
// Note: it is two underscores, not one
__global__ void func(int arg1, double arg2) {
…
}

int main() {
    int x;
    double y;
    …
    dim3 threadsPerBlock(NT_x, NT_y);
    // The number of blocks can be multi-dimensional too!
    dim3 num_blocks(Bx, By);
    func<<<num_blocks, threadsPerBlock>>>(x, y)
}
```

CUDA Thread Blocks

- You can specify thread block dimensions when calling a CUDA kernel
- CUDA has constructs available within the kernel (i.e. the function) for determining the size of the block in use and which thread we are on
  - Similar to OpenMP/MPI: querying num threads, thread ID
- However, now the thread ID can have (x, y, z) components

Thread Blocks: ThreadID

```
// Declaring a CUDA kernel
__global__ void func(int arg1, double arg2) {

    // Get the current thread's ID using threadIdx
    // Notice the extra 'x' at the end and lowercase 'd'!
    int i = threadIdx.x;
    int j = threadIdx.y;
    int k = threadIdx.z;

    // Flat threadID; D* = dim of the block in dim *
    int ID = k*Dx*Dy + j*Dx + i;

}
```

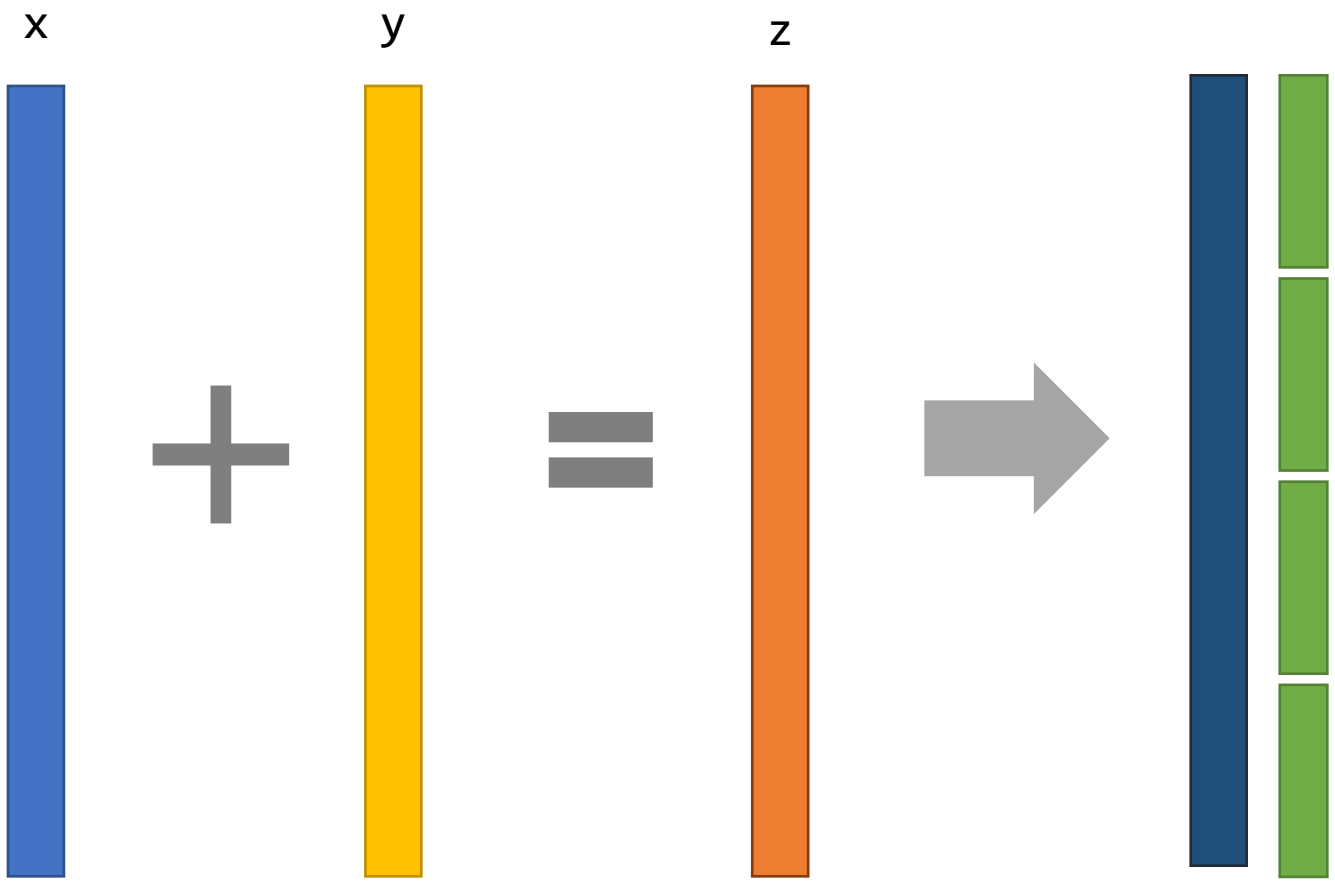
Thread Blocks: Block ID

```
// Declaring a CUDA kernel
__global__ void func(int arg1, double arg2) {

    // Get the current thread's ID using blockIdx
    // Notice the extra 'x' at the end!
    int I = blockIdx.x;
    int J = blockIdx.y;
    int K = blockIdx.z;

    // Get the current block's size
    int Dx = blockDim.x;
    int Dy = blockDim.y;
    int Dz = blockDim.z;

}
```

Multi-D Thread IDs in CUDA

- Why? GPUs are useful for dense, easily parallelizable problems.
- Such problems almost always involve multi-dimensional arrays
- Thread blocks make it easier to delegate work
  - Still probably have to do 1D to mD threadID conversions

1D Example: Addition on Vectors

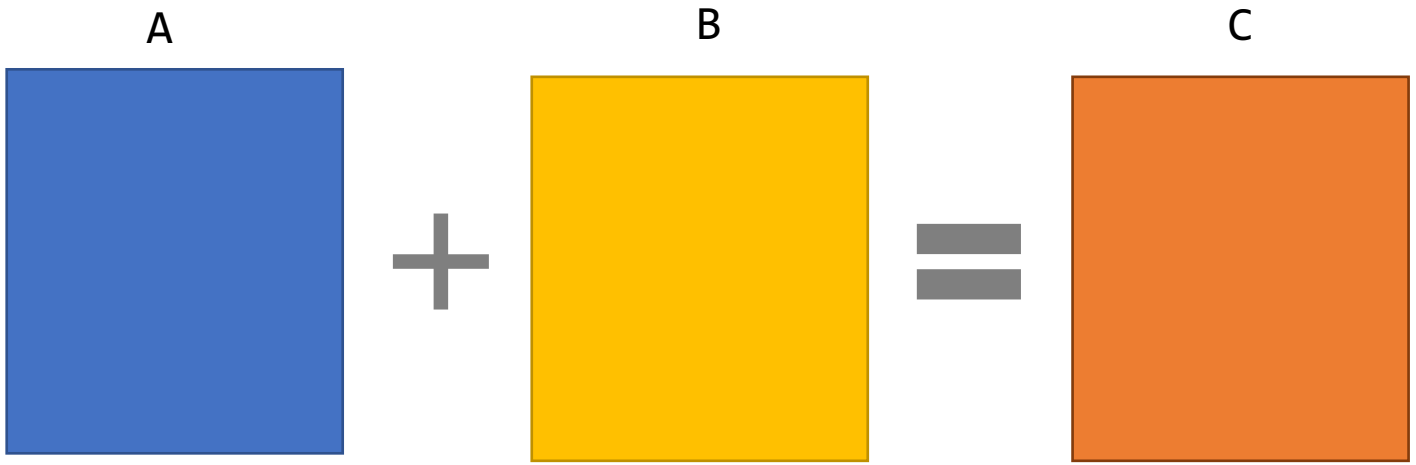


We could use 1D thread blocks to assign a portion of the vectors to each thread block

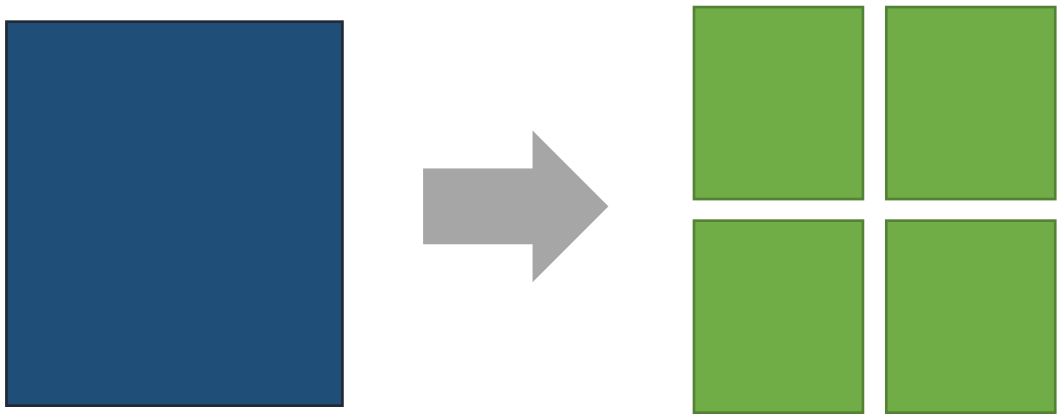
Example:

- `int threadsPerBlock = n_warp`
- `int num_blocks = ceil(n/n_warp)`

2D Example: Addition on Matrices



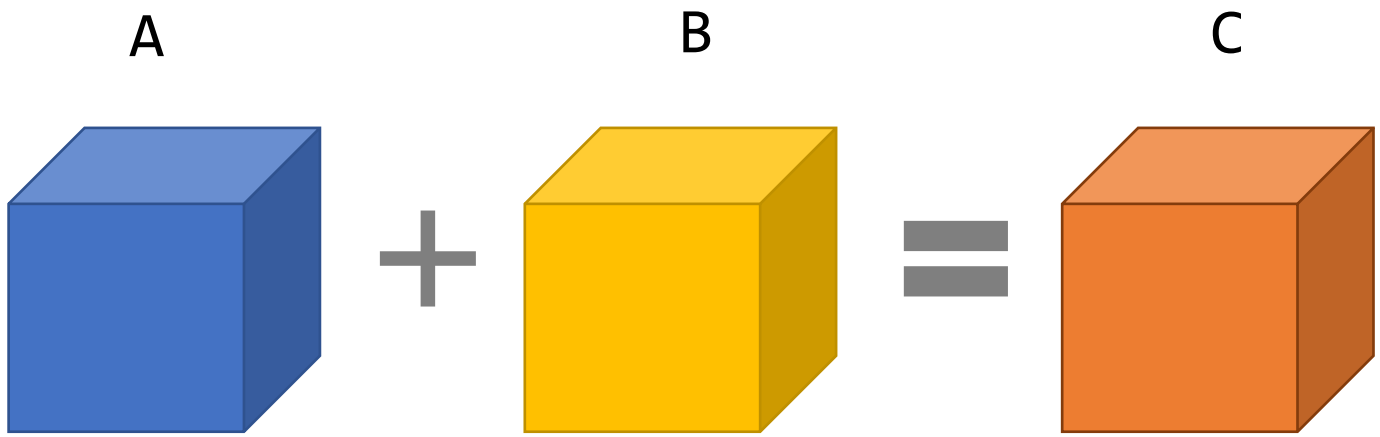
We could use **2D** thread blocks to assign a portion of the matrices to each thread block



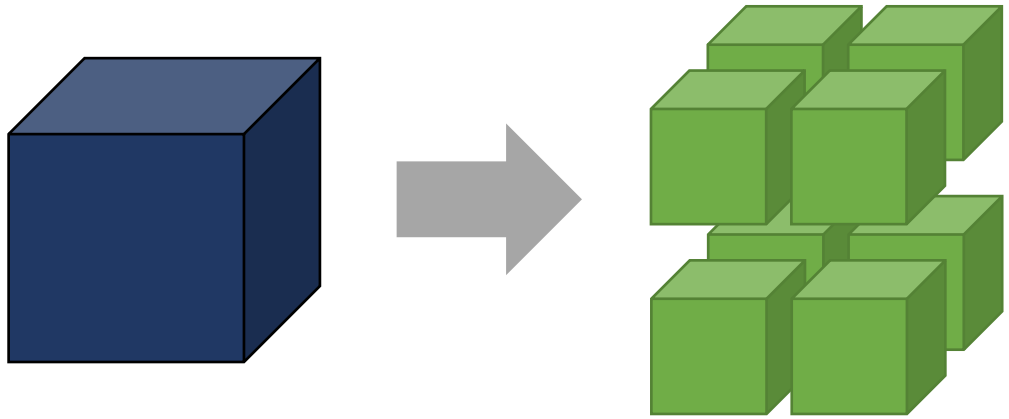
Example:

- `dim3 threadsPerBlock(n_warp, m)`
- `dim3 num_blocks(nx/n_warp, ny/m)`

3D Example: Addition on Tensors



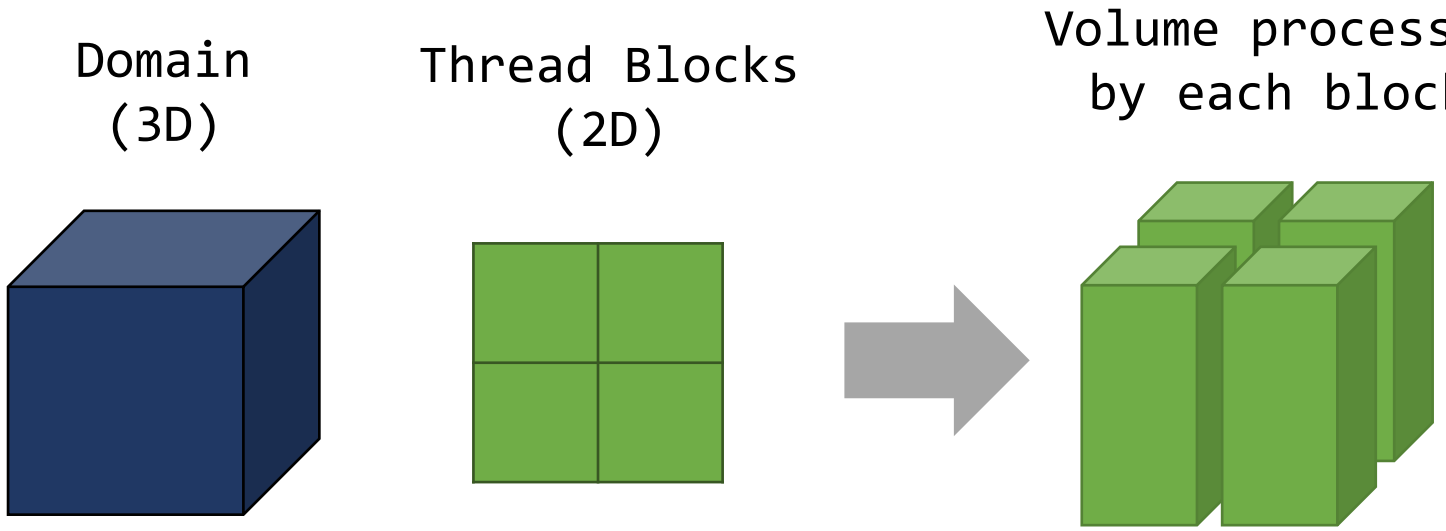
We could use **3D** thread blocks to assign a portion of the tensor to each thread block



Example:

- `dim3 threadsPerBlock(n_warp, my, mz)`
- `dim3 num_blocks(nx/n_warp, ny/my, nz/mz)`

Lower-Dimension Thread Blocks



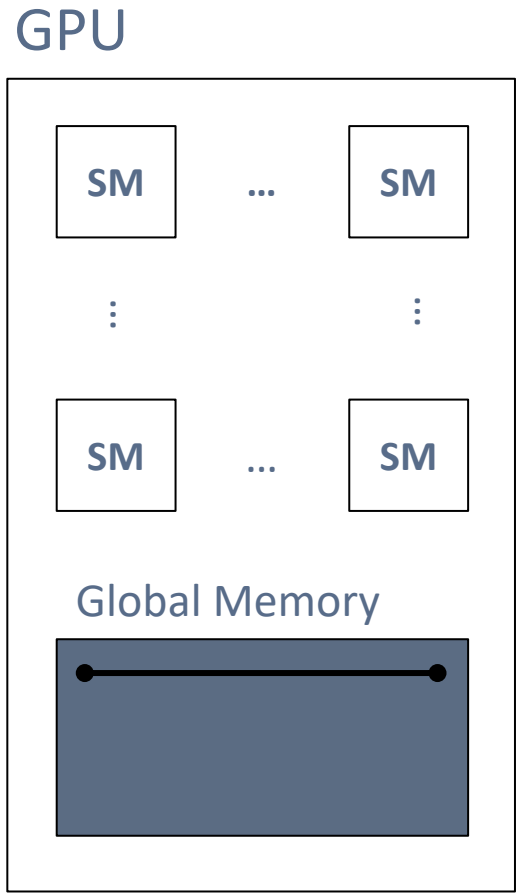
Example:

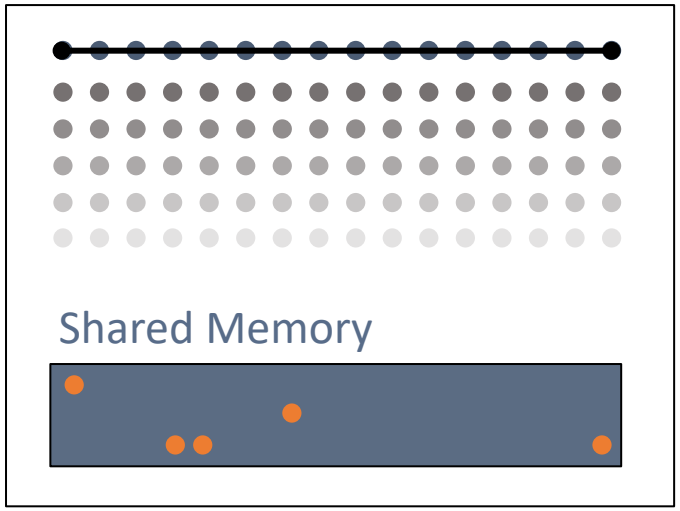
- dim3 threadsPerBlock(n_warp, my)
- dim3 num_blocks(nx/n_warp, ny/my)
- **Each block processes the entire z domain**

- 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 present a real issue
- Can also be better than using dim(thread block) = dim(data) for some applications due to memory

Memory Coalescing

- **Memory coalescing is adjacent threads reading from adjacent memory**
- Memory is loaded via caches from global memory
  - Cache lines: typically 128 bytes = (4 per float) x 32
  - There are 32 thread/warp
- A single cache hit supplies memory to an entire warp of threads



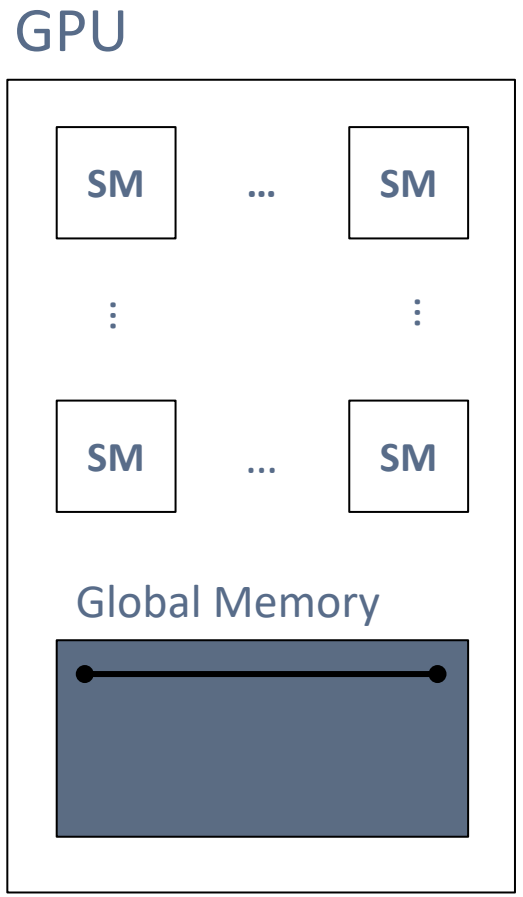


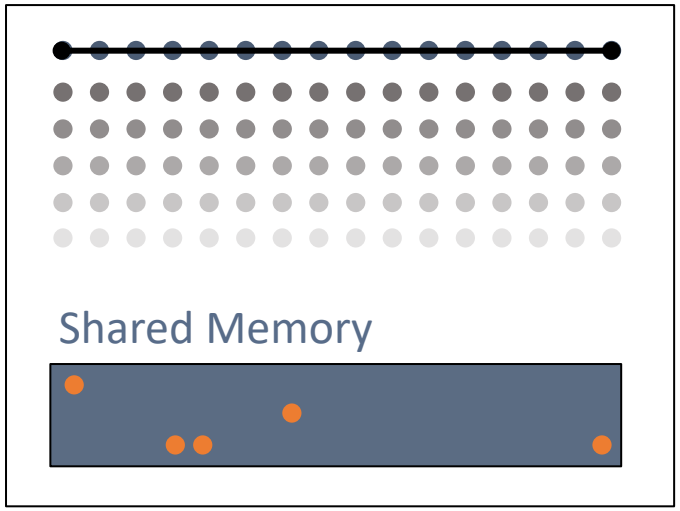
Memory Coalescing

- Some examples…
  - Matrix-vector multiplication
  - Writing to a matrix using a 2D grid

Shared memory

- **Shared memory is fast**
  - ~100x faster to access than global memory
  - Local to **each block**
- Access can be random without incurring non-coalesced read penalty
  - Ex: coalesced loads from global memory into shared, random-access shared memory





https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

Synchronization and communication

- Shared memory is typically used to communicate between a block of threads
- __syncthreads() is a barrier for threads **in a block**
  - Often used to ensure that shared memory is ready to read
- Communication between blocks of threads must be done by sharing via global memory
  - Synchronization happens between blocking kernel calls

Allocating Shared Memory

- Distributing kernel thread blocks across the GPU is handled for us
- Blocks are never split across SMs
  - How much memory blocks require can limit the number of blocks that can run on a single SM
  - Example: thread-wise, 4 blocks can fit on a SM, but 1 block uses 0.5 the shared memory -> only 2 blocks will run
- If CUDA handles mapping blocks to SMs and memory can influence how many blocks per SM, **we cannot arbitrarily allocated shared memory**
  - **Plus, shared memory needs to be allocated at a block-level**

Allocating Shared Memory

- Shared memory can be allocated 2 ways, **statically** (stack) and **dynamically** (heap)
  - The size of statically declared memory is known at compile time; works great for managing block memory
  - The whole point of dynamically allocated memory is we don't know until runtime how much memory we will need
    - When is the latest we can allocate it that still allows CUDA to manage blocks/SMs?
  - **Dynamically allocated stack memory must be specified at the kernel invocation**
  - i.e. when we declare the number of blocks and block size

Allocating Shared Memory: Static

```
// Declaring a CUDA kernel with static share memory
__global__ void func(int arg1, double arg2) {
    __shared__ int array[64];
    …
}

int main() {
    int x;
    double y;
    …
    // Invoking (calling) the kernel
    func<<<num_blocks, threadsPerBlock>>>(x, y)
}
```

Allocating Shared Memory: Dynamic

```
// Declaring a CUDA kernel with dynamic share memory
__global__ void func(int arg1, double arg2) {
    extern __shared__ int array[]; // Note no size is specified
    …
}

int main() {
    int x;
    double y;
    …
    // Invoking (calling) the kernel
    int n_sharedMem = n*sizeof(int); // Size in bytes!!!
    func<<<num_blocks, threadsPerBlock, n_sharedMem>>>(x, y)
}
```

Dynamic Shared Memory

- What if we want multiple dynamically allocated arrays?
- Notice the shared memory block was given in bytes…
- **You can only allocate one block of shared memory**
  - This is like how we allocated window memory in RMA/One-sided communication in MPI
  - If you want multiple arrays, you need to declare pointers for each and then point them to the appropriate byte-measured location in the shared memory block

Allocating Multiple Shared Arrays

```
// Declaring a CUDA kernel with dynamic share memory
__global__ void func(int n_fl, int n_int) {
    extern __shared__ int shared_mem[];
    float* vec_fl = shared_mem;

    int offset = n_fl * sizeof(float) / sizeof(int);
    int* vec_int = shared_mem[n_offset];
}

int main() {
    // Invoking (calling) the kernel
    int n_sharedMem = n_fl*sizeof(float) + n_int*sizeof(int);
    func<<<num_blocks, threadsPerBlock, n_sharedMem>>>(x, y)
}
```

Example: Reversing a Vector

- Assume the vector has an even number of elements
- Each thread works elements i and i_rev = n – 1 – i
- This is usually bad for global memory (not a coalesced access pattern)
- Solution:
  - Load a chunk of x into shared memory using coalescing
  - Perform permutation in shared memory (can do at the same time as the write)
  - Write the result back to global mem using coalescing

Reversing a Vector

```
// x lives in global memory
__global__ void reverse(float* x, int n) {
    extern __shared__ float shared_mem[];

    // Calculate each thread's index into the array
    int i = threadIdx.x + blockIdx.x*blockDim.x;

    shared_mem[i] = x[i]; // This is the coalesced memory read
    __syncthreads(); // Make sure everyone finished the read operation
    x[i] = shared_mem[n-i-1]; // This is the coalesced write
}
```

Review of GPU memory so far

- Host memory
  - On the CPU, no size limit, PCIE transfer (O(1-10) GB/s)
- Global memory
  - On the GPU, size O(10) GB
  - Transfer speed: 50-100 GB/s bandwidth
  - Need “coalesced” reads for fast memory accesses
- Shared memory
  - On GPU, local to each “streaming multiprocessor” (SM)
  - Roughly 10x faster than global memory

Matvec using shared memory

- Multiplication of a tall, narrow matrix with a small vector.
- General matmul will be treated later.
- See matvec_smem.cu

GPU reduction using shared memory

- Tree-based implementation, similar to the implementation of MPI_Reduce
- See reduction.cu

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, GPUs have fairly large register memory
  - Still expensive if overused (register “spilling”)

Counting registers

```
__global__ void vector_add(const float * A, const float * B, float * C,
                           const int size){

   int item = (blockIdx.x * blockDim.x) + threadIdx.x;

   if ( item < size ){
      C[item] = A[item] + B[item];
   }
}
```

How many registers do you think this uses?

Counting registers

```
__global__ void vector_add(const float * A, const float * B, float * C,
                           const int size){

    int item = (blockIdx.x * blockDim.x) + threadIdx.x;
    float temp[3]; // this also gets put in register memory

    if ( item < size ){
        temp[0] = A[item];
        temp[1] = B[item];
        temp[2] = temp[0] + temp[1];
        C[item] = temp[2];
    }
}
```

We don’t have to count register usage manually; can use ‘nvcc –ptxas-options=“-v” …’.

Local memory (not explicitly managed)

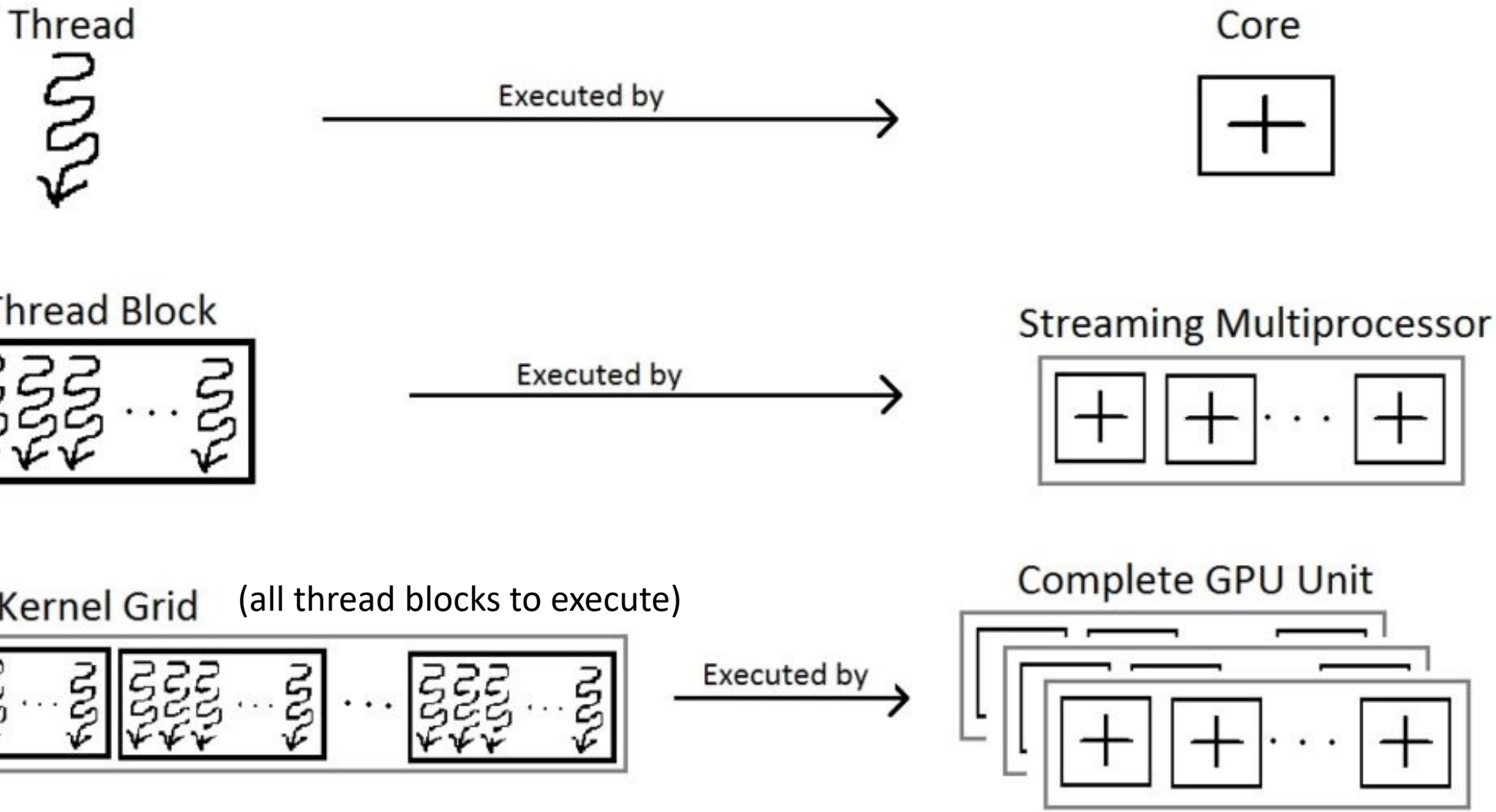
```
#define CHUNK 8
__global__ void foo(const int size, …)
{
      float temp_local[size]; // this gets put in local memory
      float temp_register[CHUNK]; // this gets put in register memory
}
```

- Thread-local memory, but not statically sized
- Local memory: large but as slow as global memory
- If you use too many registers (e.g., “register spilling”) then the GPU switches to local memory.

Summary: different GPU memories

- Host memory: slowest transfer speeds
  - Try to only access at beginning/end of simulation
- Global memory: fast if memory reads are coalesced
  - Large, shared with the entire GPU
- Shared memory: fast random-access reads
  - Small, **shared within an SM**
  - If you use too much, then…?
- Register memory: fast thread-local reads
  - Small, **shared within an SM**
  - If you use too much, then…?

Hardware vs software terms



http://thebeardsage.com/cuda-streaming-multiprocessors/

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

Data from Aydin Buluc

GPU hardware/software limits

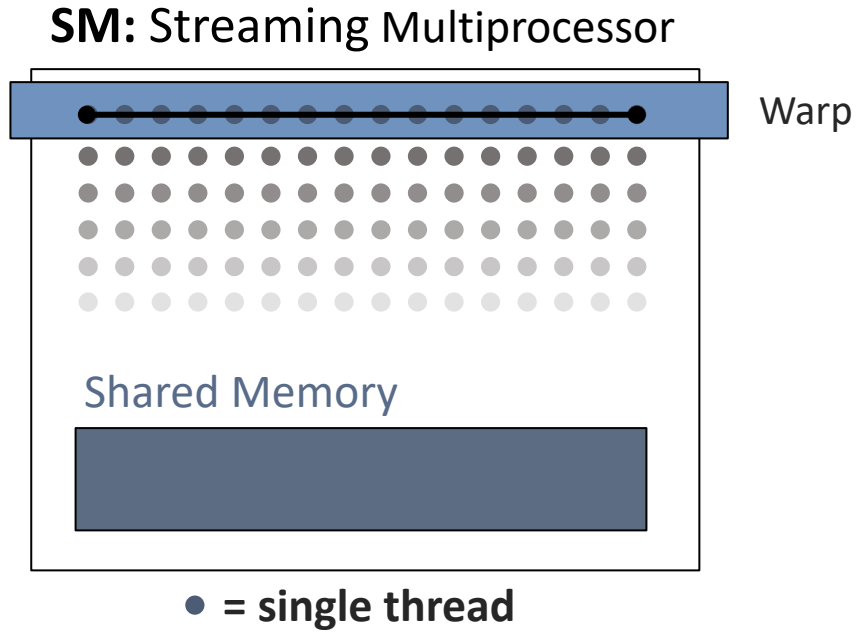
GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6	7
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	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 per thread
- But we only have 64 KB (16000 ints) of register memory…?

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

GPUs may run fewer than max number of warps

- Tesla K80 (on NOTS):
  - 2 K40s glued together
  - SM/GPU: 13 (x2)
  - Threads/warp: 32
  - Threads/SM: 2048
  - **Max active warps/SM: 192 (x2)**
  - **CUDA cores: 2496 (x2) = (13 x 192)**
- Tesla P100:
  - SM/GPU: 56
  - Thread/warp: 32
  - Threads/SM: 2048 (32 x 64)
  - **Max active warps/SM: 64**
  - **CUDA Cores: 3584 = (56 x 64)**



How do GPUs divide up work?

- "Blocks" of threads are run by streaming multiprocessors (SMs)
  - Each thread block is local to an SM, but **one SM may execute multiple thread blocks**
- Each SM has some number of warps (chunks of 32 threads), the smallest unit of execution on a GPU.
  - Each warp is made of up of multiple (32) CUDA "cores", which each execute thread instructions
  - A thread block with 33 threads still gets two warps

How do GPUs divide up work? cont.

- 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.

https://stackoverflow.com/a/9841310

GPUs limit SM utilization based on shared resources

- Each SM can execute multiple thread blocks
- **HOWEVER**, each SM has a limited pool of threads, as well as shared and register memory.
- An SM can run fewer thread blocks if each thread block uses a lot of resources
  - Many threads = fewer thread blocks (usually OK)
  - Too much shared/register memory (not as OK)
  - Unused warps = potentially inefficient GPU usage
- SM utilization efficiency measured by “occupancy”

Occupancy

- SM utilization efficiency measured by “occupancy”
- Occupancy calculator: https://xmartlabs.github.io/cuda-calculator/
- Can use "nvcc -arch=sm_37 --ptxas-options=-v matread.cu code_to_compile.cu” to analyze register usage **per thread** and shared memory usage **per block**

Note that a high occupancy code is not always more efficient than a lower occupancy code!