Applications of Parallel Computers

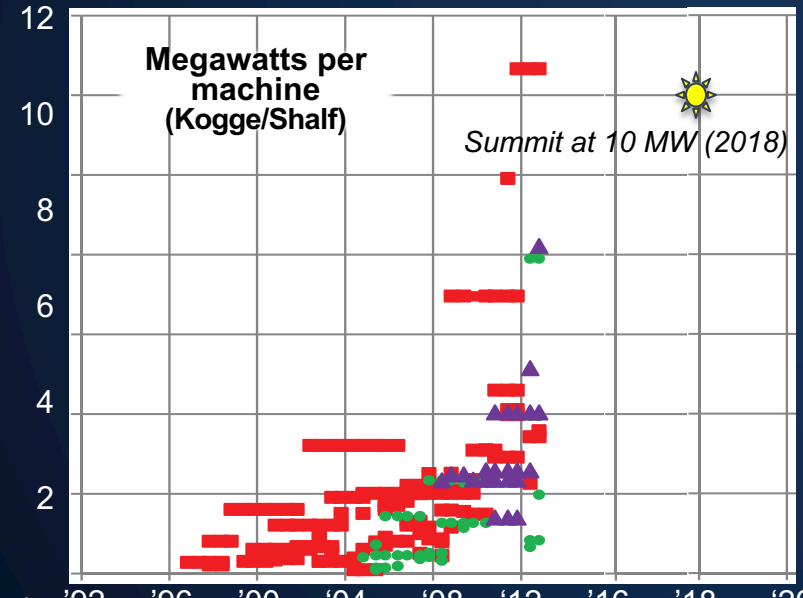
Introduction to Graphics Processing Units (GPUs)

https://sites.google.com/lbl.gov/cs267-spr2021

Acknowledgements

- Slides from today's lecture are derived from
  - John Owens, UC Davis
  - Bill Dally, Stanford / NVIDIA
  - Kayvon Fatahalian, Stanford
  - Kurt Keutzer, Berkeley
  - Brian Catanzaro, NVIDIA (Berkeley grad)

HPC Energy Use



At ~$1M per MW, energy costs are substantial

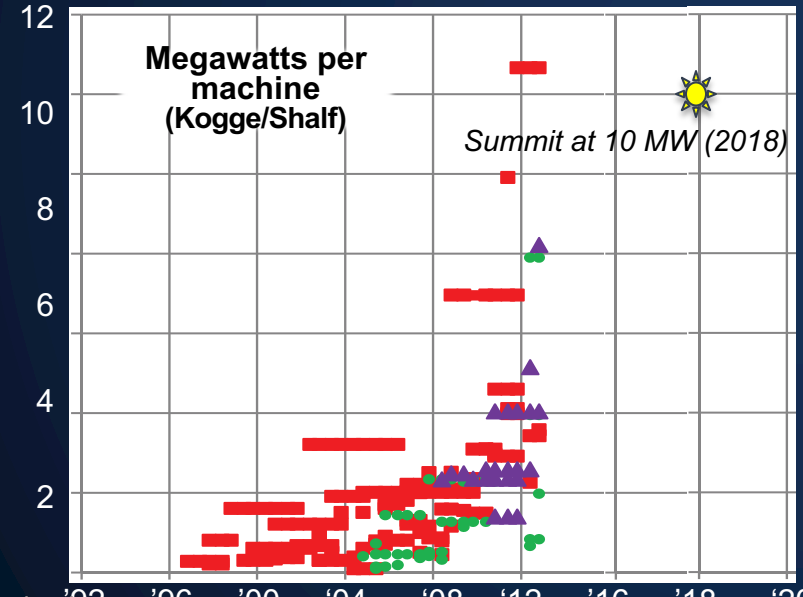
- 1 petaflop in 2008 used 3 MW
- 1 exaflop was projected for 2018 at 200 MW "usual chip scaling" so the goal was 20 MW

HPC Energy Use

Fugaku at 28 MW (2020)

TaihuLight at 15 MW (2016)

Tihanhe-2 at 18MW (2013)



At ~$1M per MW, energy costs are substantial

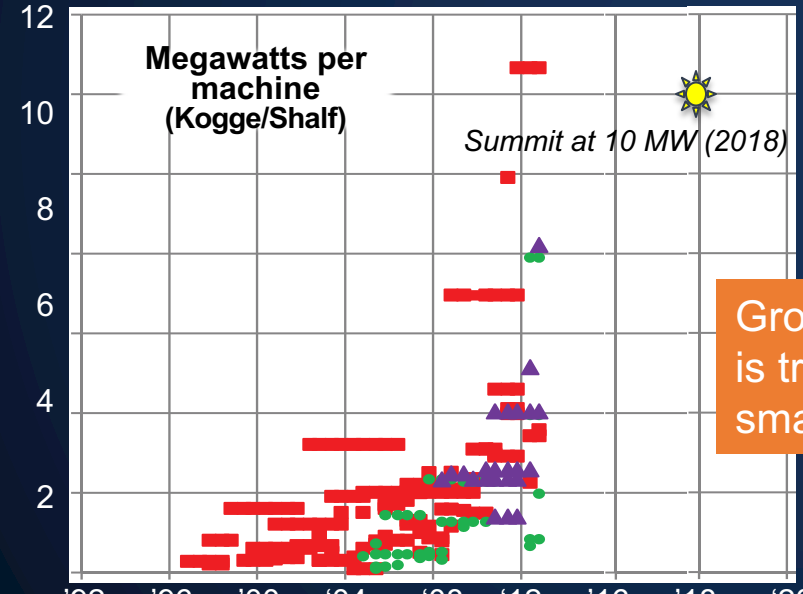
- 1 petaflop in 2008 used 3 MW
- 1 exaflop was projected for 2018 at 200 MW "usual chip scaling" so the goal was 20 MW
- Reality will probably be close to 50 MW in 2022

HPC Energy Use

Fugaku at 28 MW (2020)

TaihuLight at 15 MW (2016)

Tihanhe-2 at 18MW (2013)



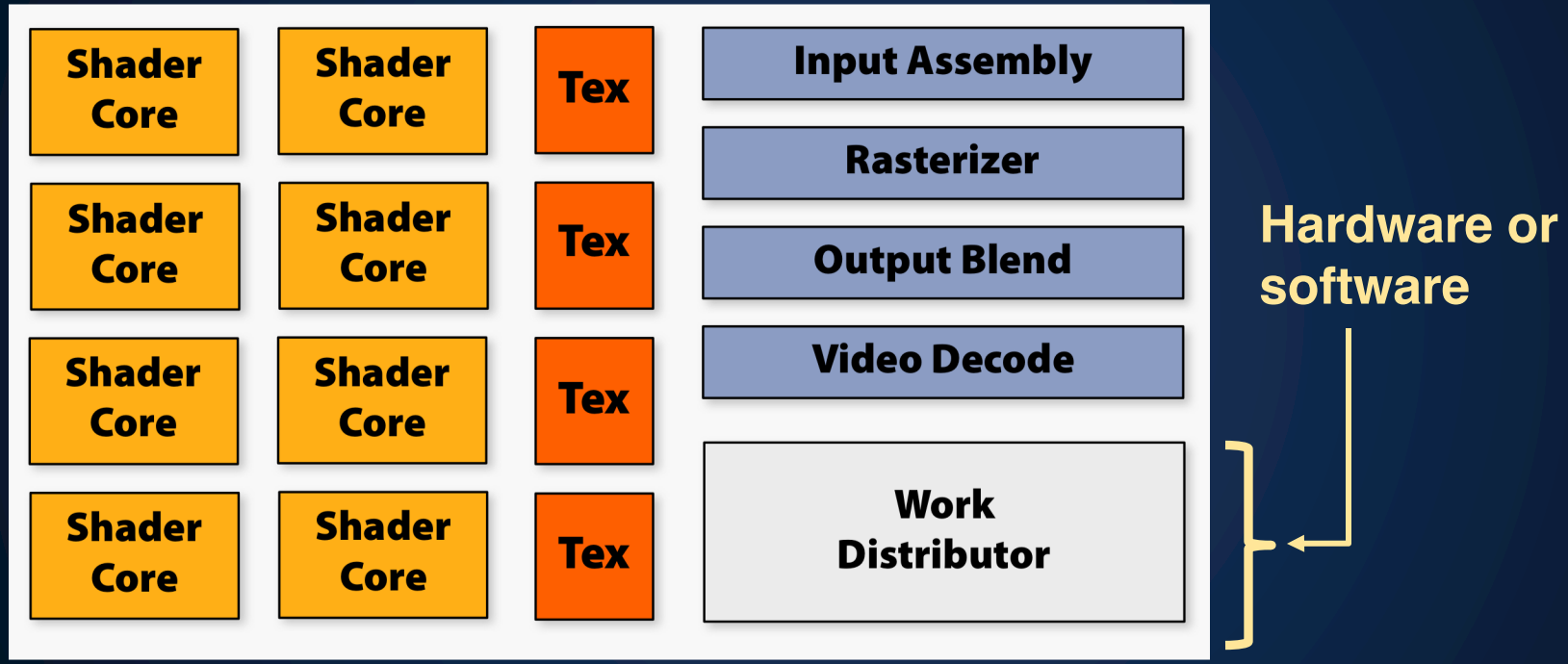
At ~$1M per MW, energy costs are substantial

- 1 petaflop in 2008 used 3 MW
- 1 exaflop was projected for 2018 at 200 MW "usual chip scaling" so the goal was 20 MW
- Reality will probably be close to 50 MW in 2022

Growth in energy use is true on a Cloud or smaller scale

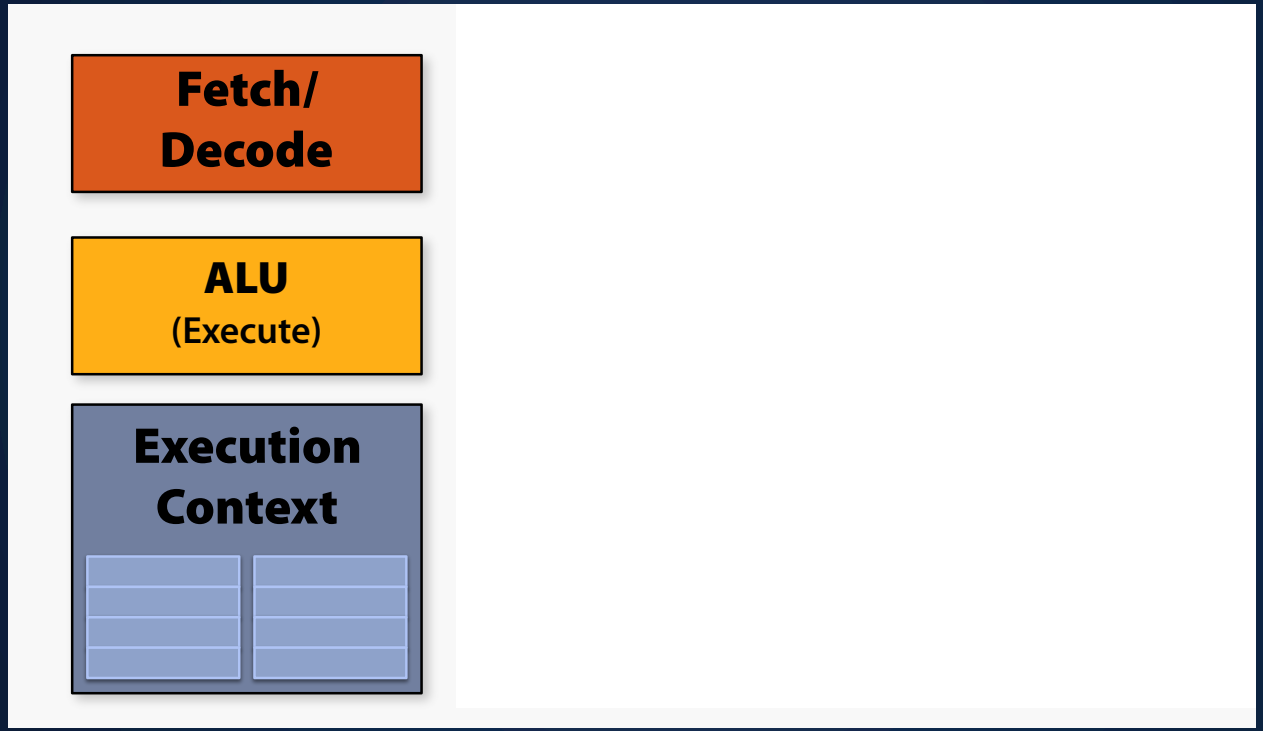
GPUs for Graphics Applications

What's in a GPU?

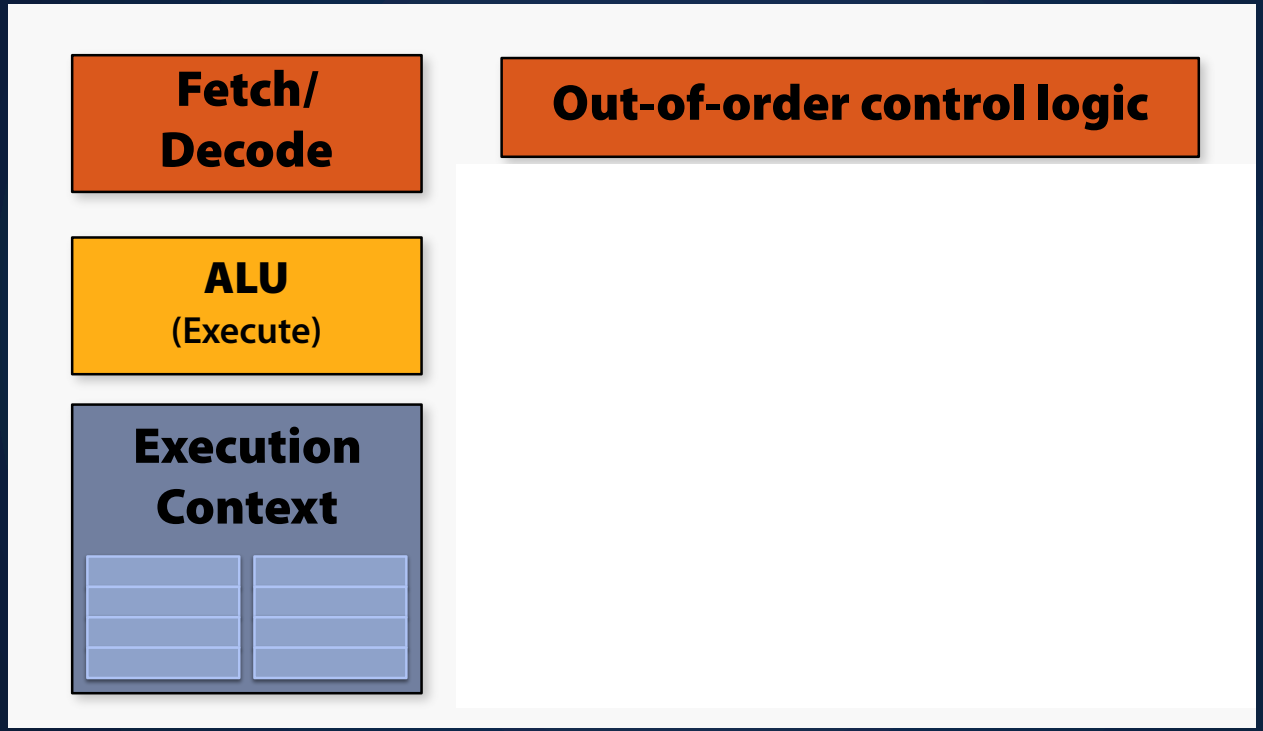


Heterogeneous chip multi-processor (tuned for graphics)

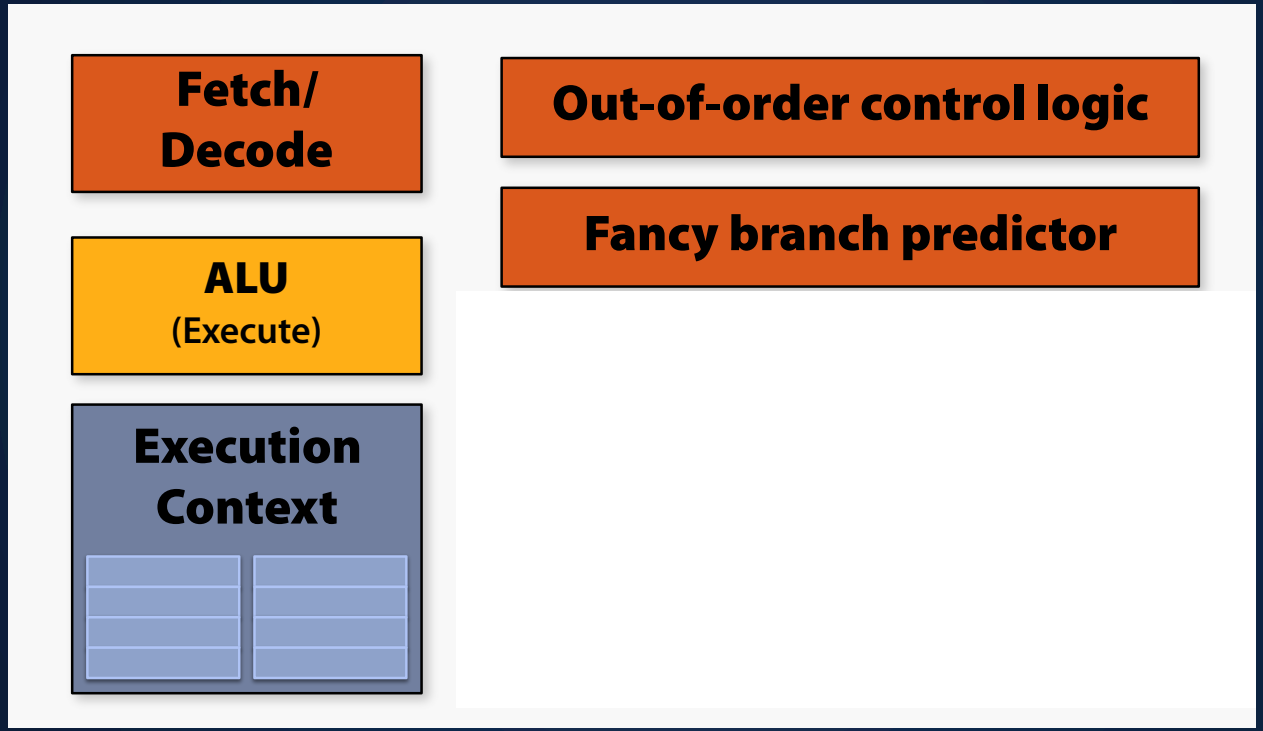
What's in a CPU?



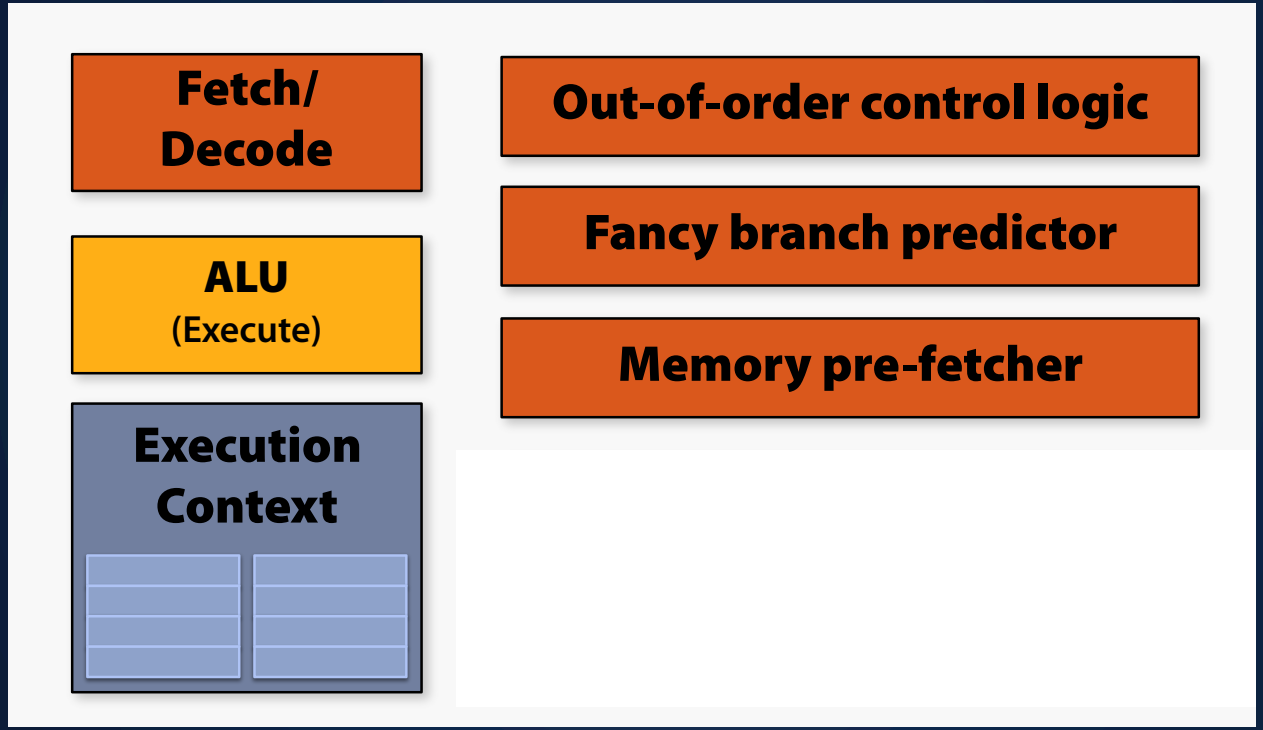
What's in a CPU?



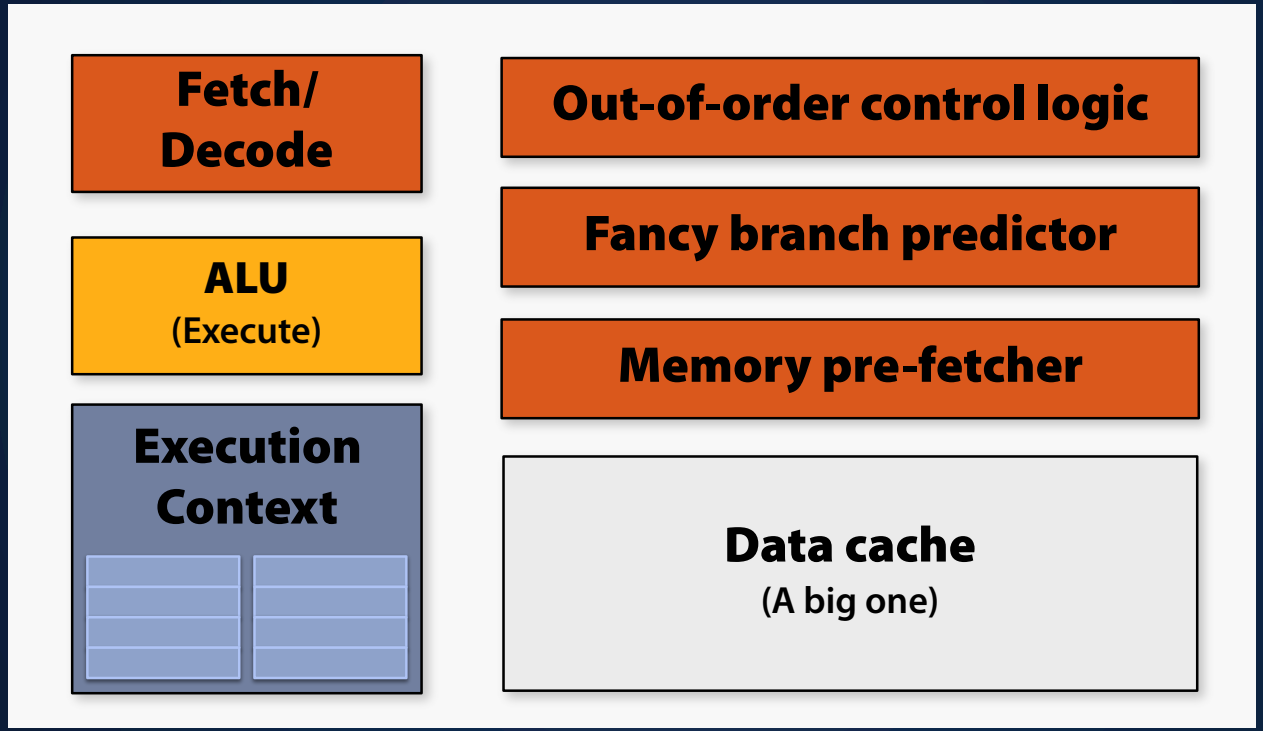
What's in a CPU?



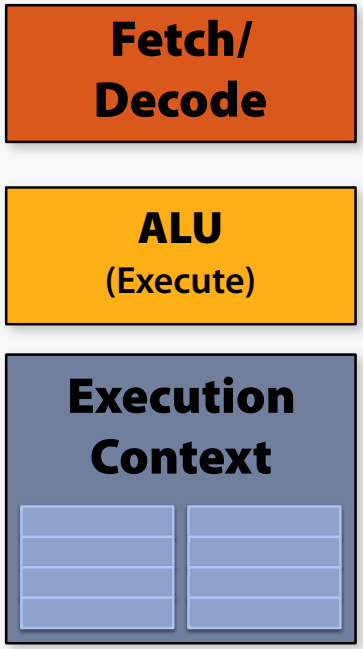
What's in a CPU?



What's in a CPU?



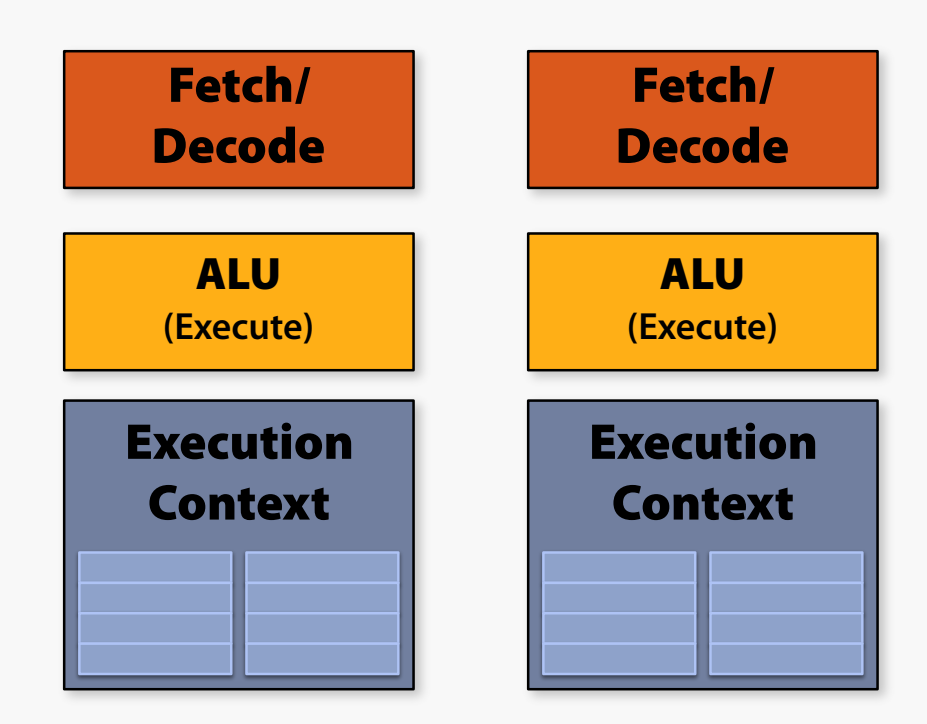
What's in a CPU?



Hardware Big Idea #1:
Remove components that help a single instruction stream run fast

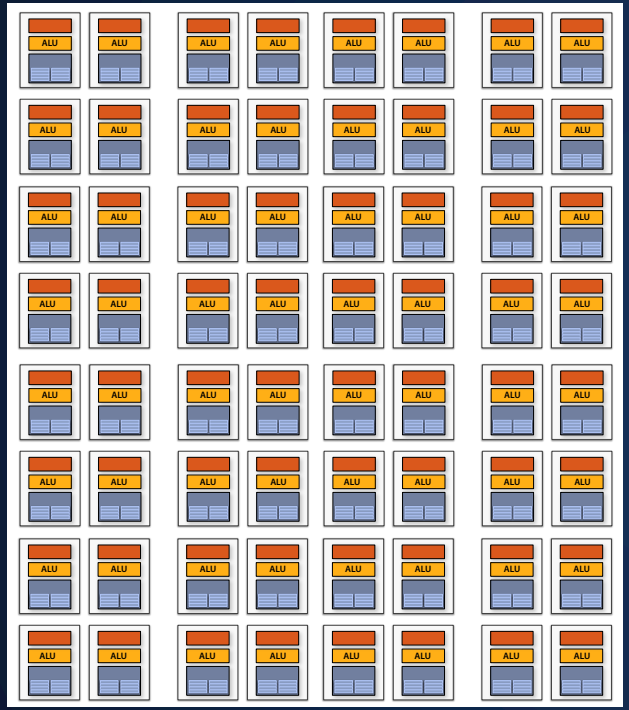
- Discover parallelism
- Consume energy

Multicore Processors



Two Cores: Two Threads in Parallel

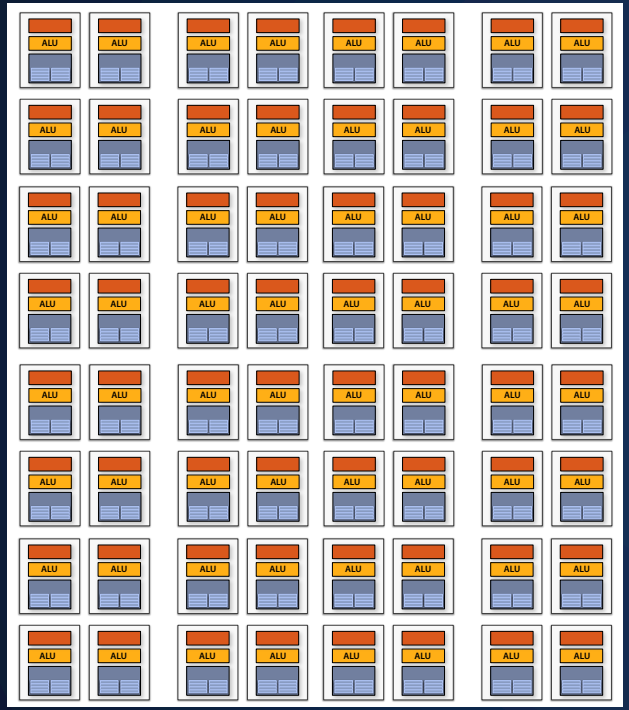
Multicore to Manycore



64 Cores: 64
Threads in Parallel

Hardware Idea #2:
A larger number of (smaller simpler) cores

Multicore to Manycore

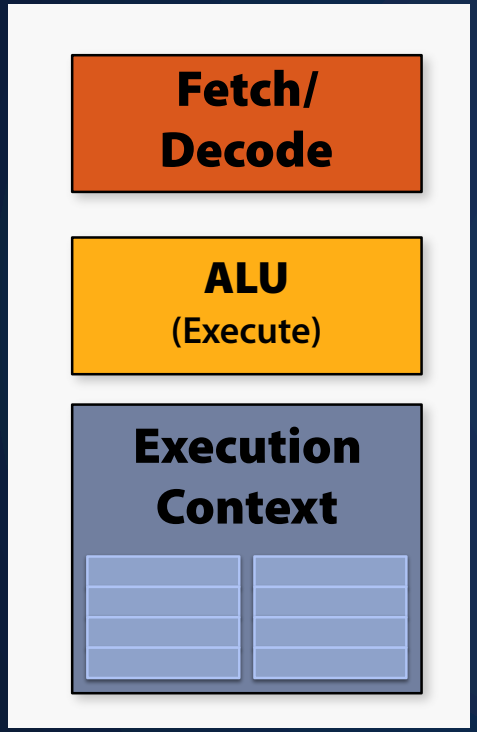


64 Cores: 64 Threads in Parallel

Hardware Idea #2:
A larger number of (smaller simpler) cores

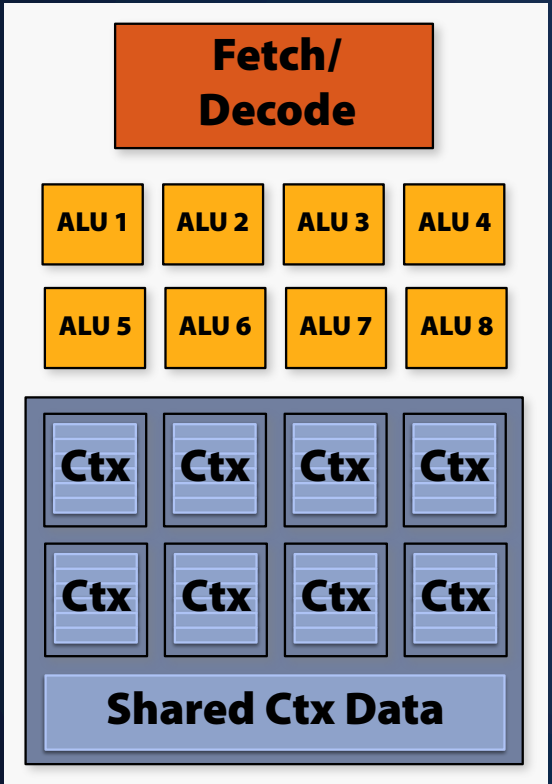
With this much parallelism, it's likely to be *data parallel* (same operation, different data)

Recall simplified core



Hardware Big Idea #3:
Share the instruction stream

Adding ALUs (Removing Decoders)

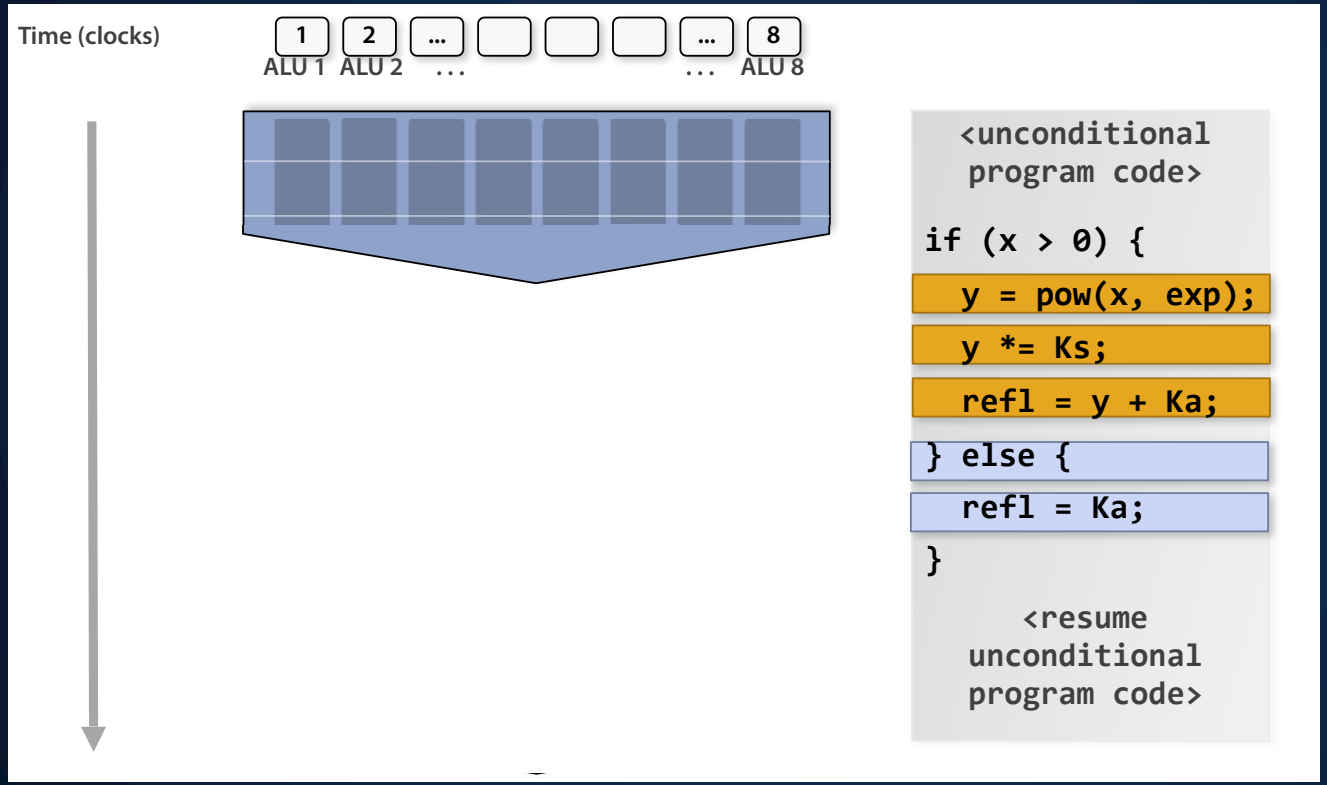


NVIDIA will call this a warp of threads, or

- SIMT: single instruction multiple threads
- (Close to SIMD: single instruction multiple data)

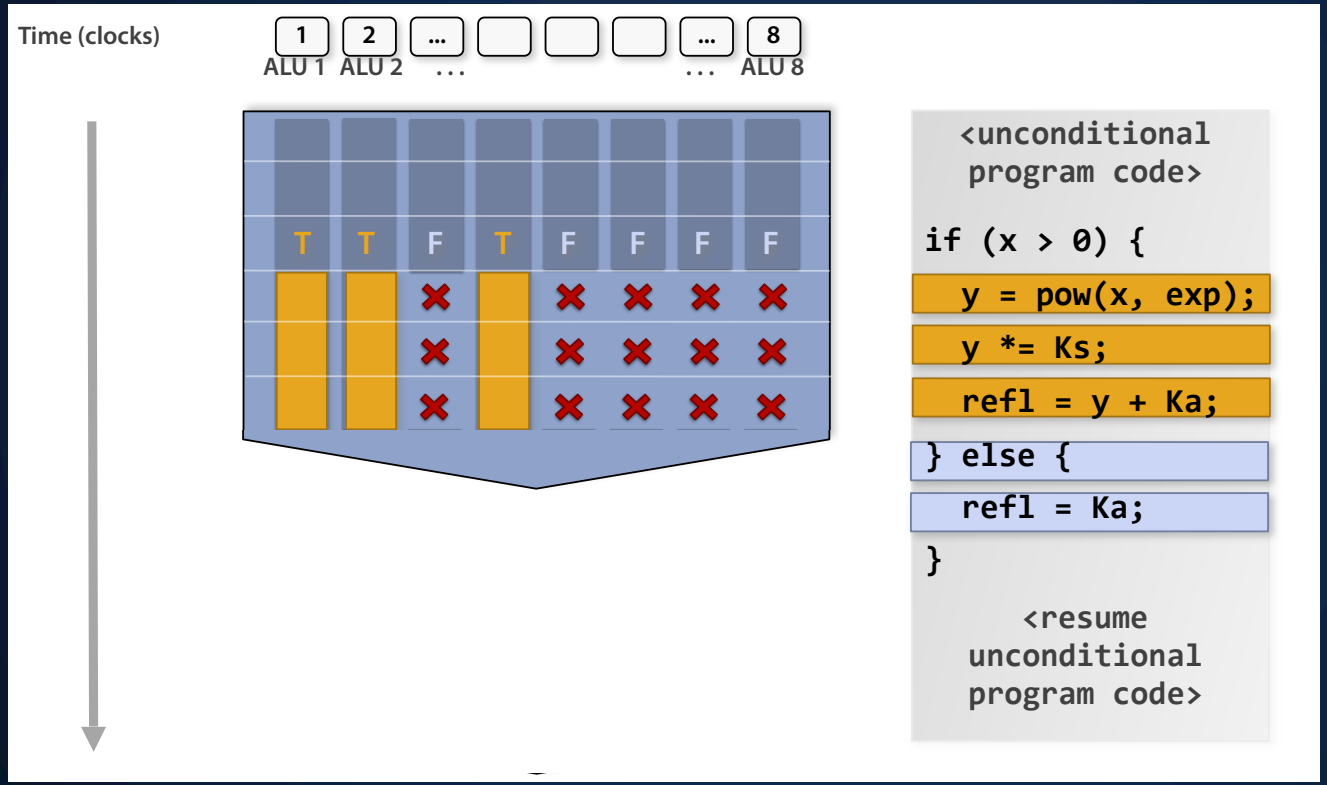
Ctx = Context

What about Branches?

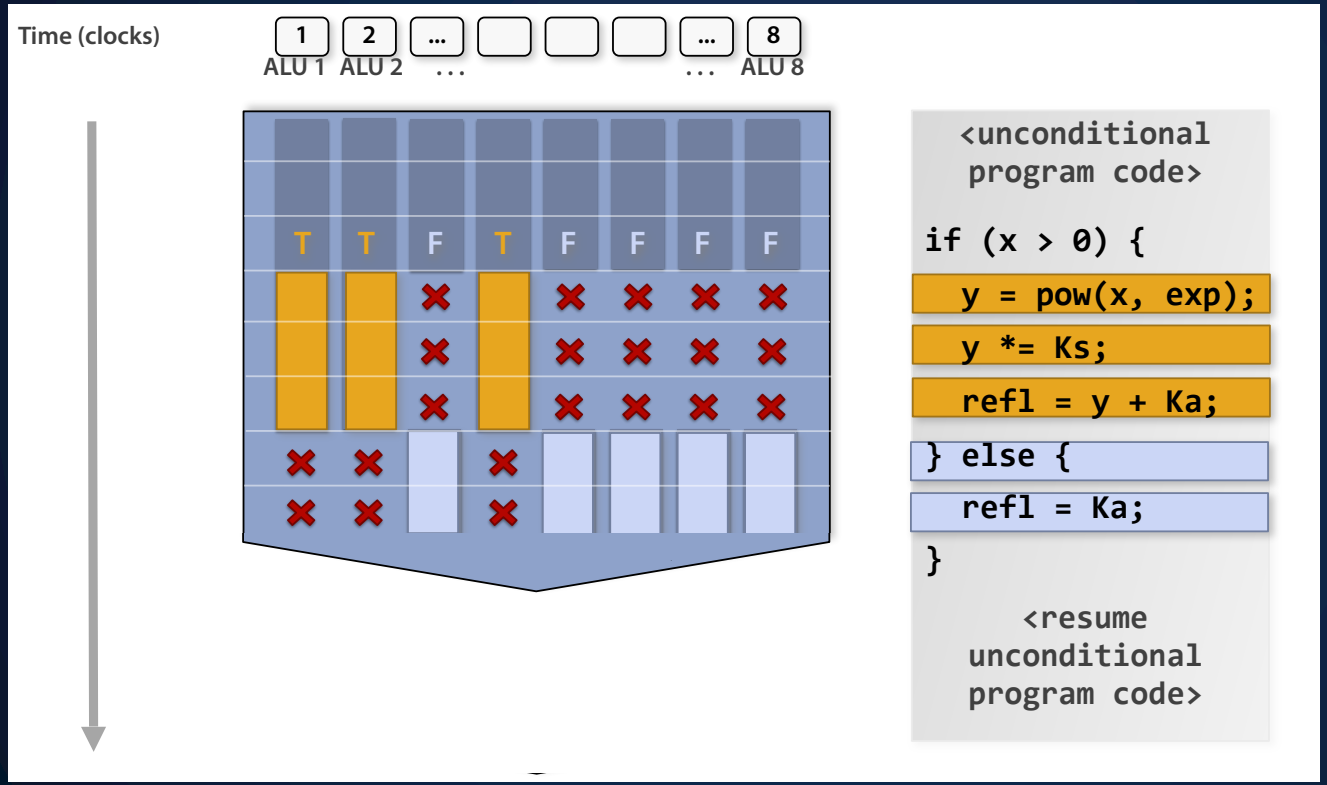


Hardware Idea #4: Masks

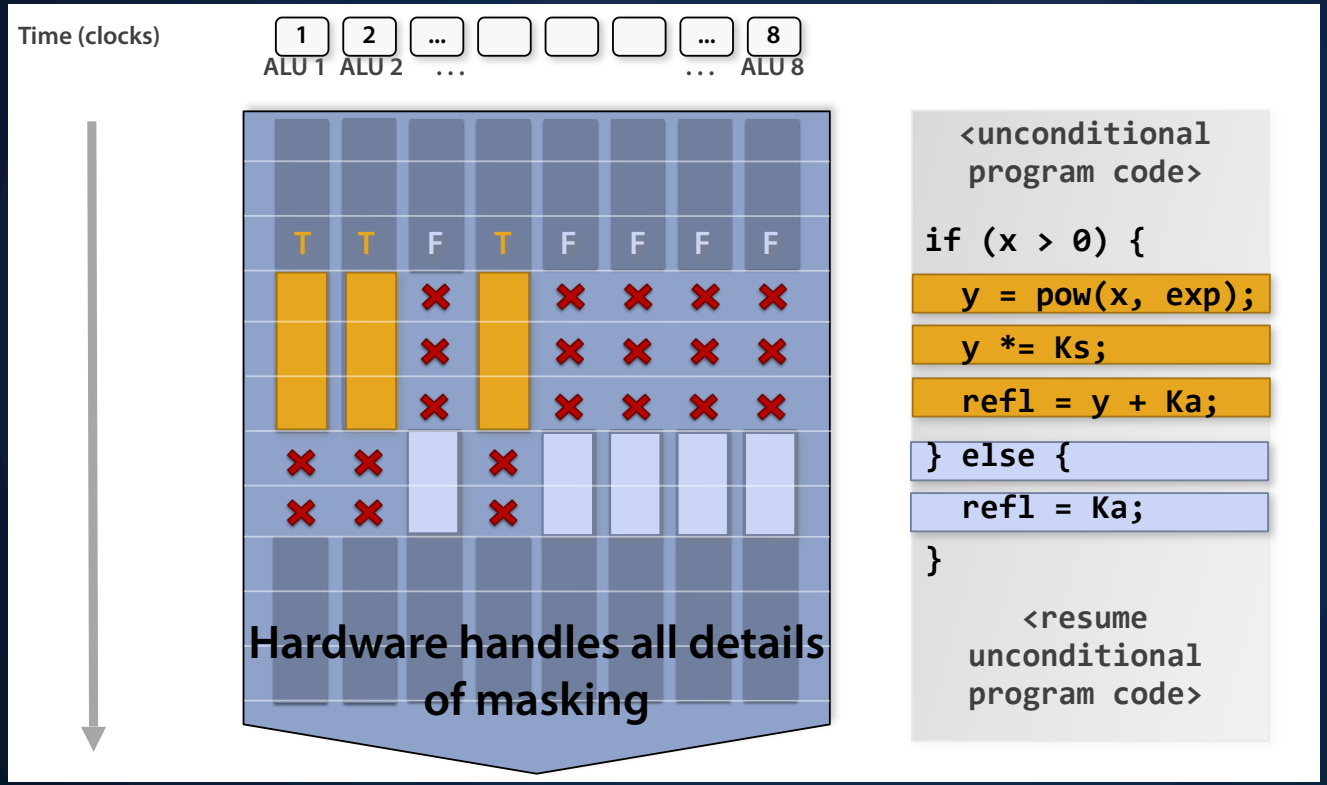
What about Branches?



What about Branches?



What about Branches?



Stalls

- **Stalls: a core cannot run the next instruction because of a dependency on a previous one**

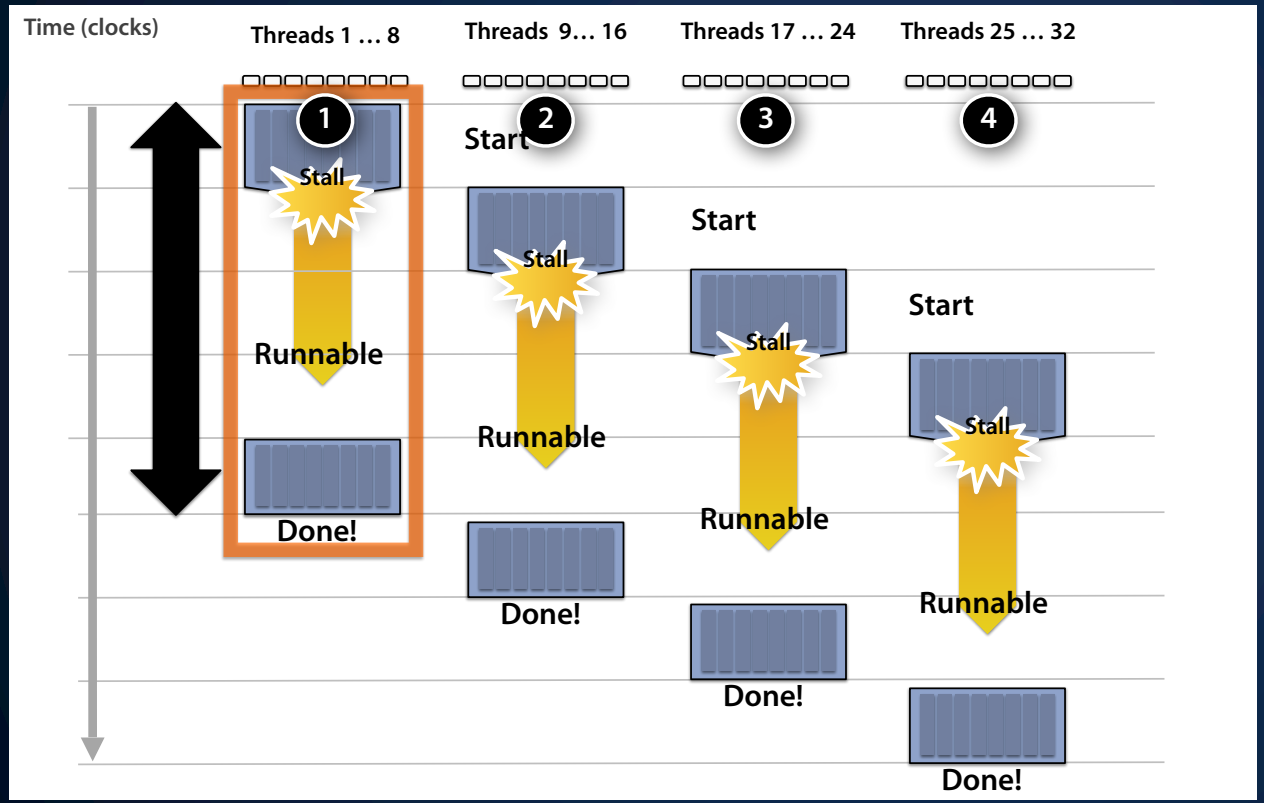
Stalls

- **Stalls: a core cannot run the next instruction because of a dependency on a previous one**
- **Memory operations are 100s-1000s of cycles**

Stalls

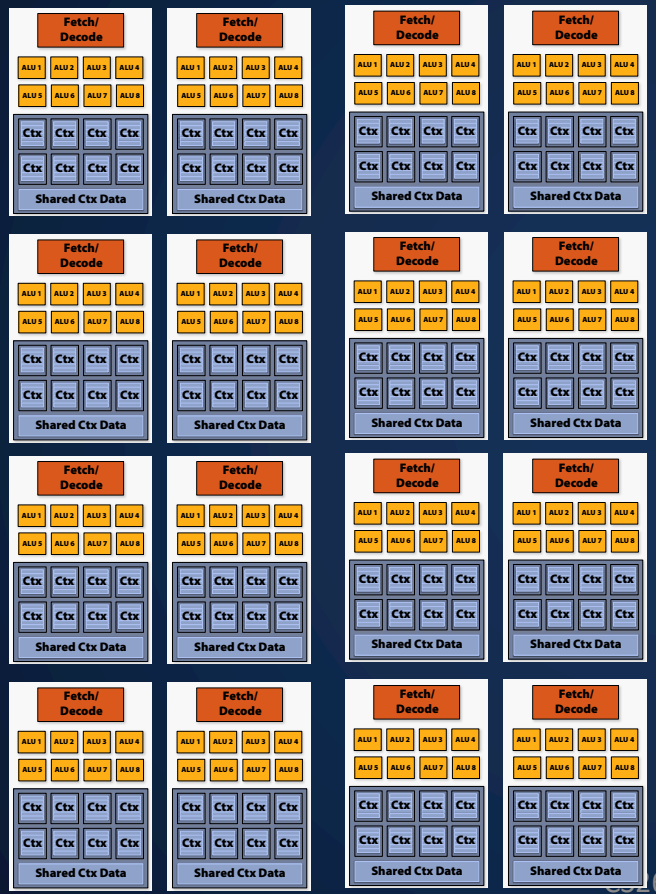
- **Stalls: a core cannot run the next instruction because of a dependency on a previous one**
- **Memory operations are 100s-1000s of cycles**
- **Removed the fancy caches and prefetch logic that helps avoid stalls**

Optimize for Throughput: More threads!



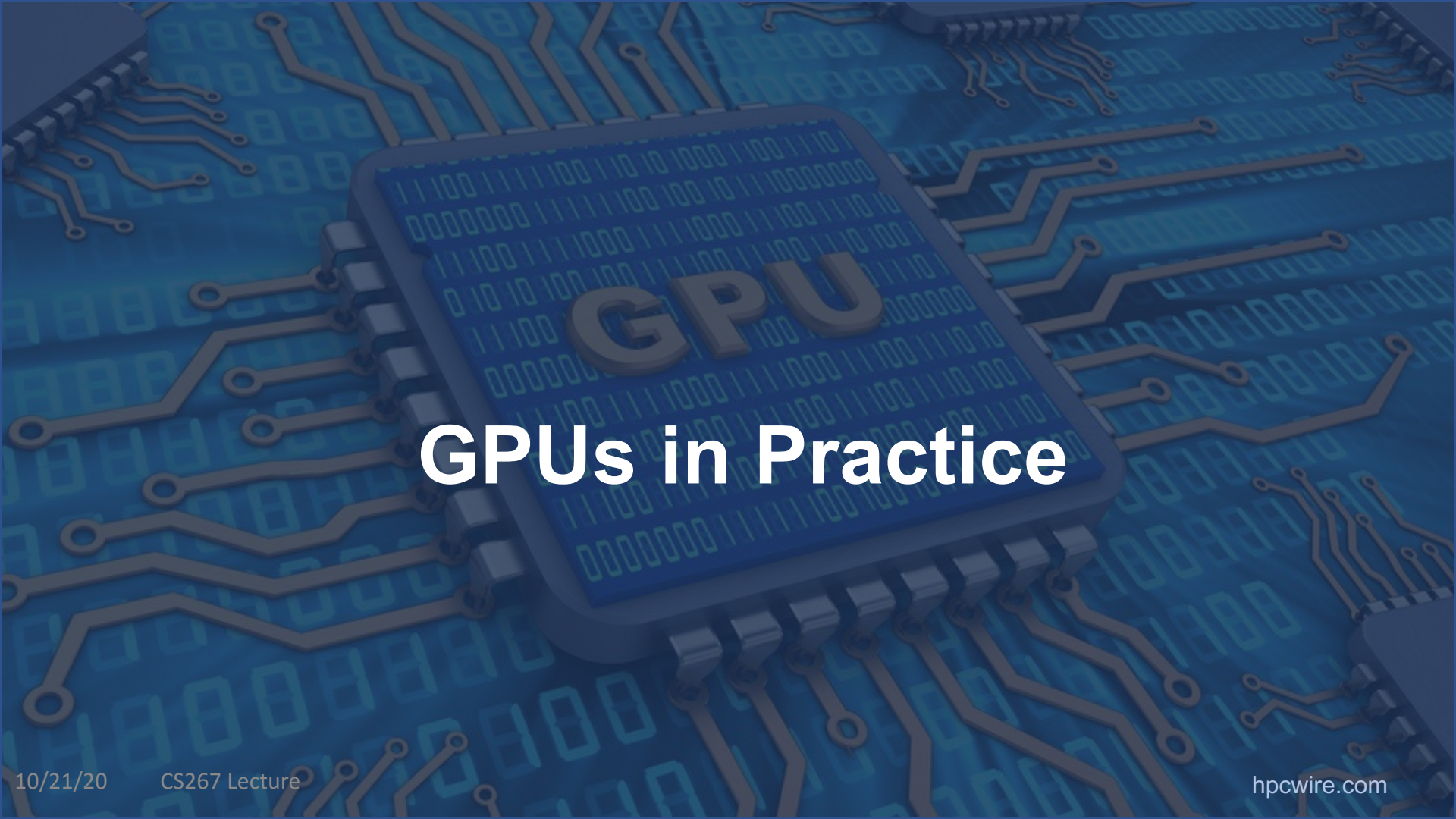
Hardware idea #5: Use threads to hide high latency operations

Two Levels of Parallelism



This “abstract” chip has:

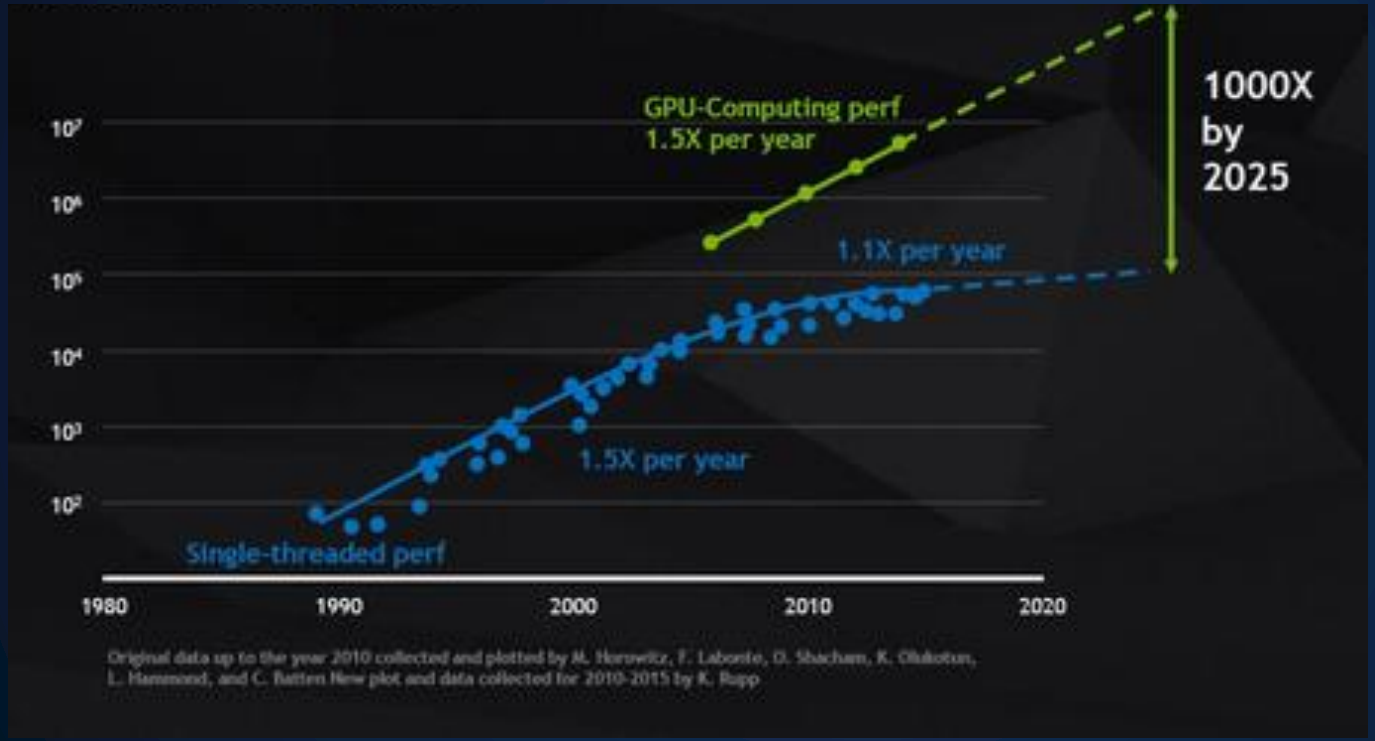
- 16 cores
- 8 MADD units per core
- 256 Gflop/s at 1 GHz



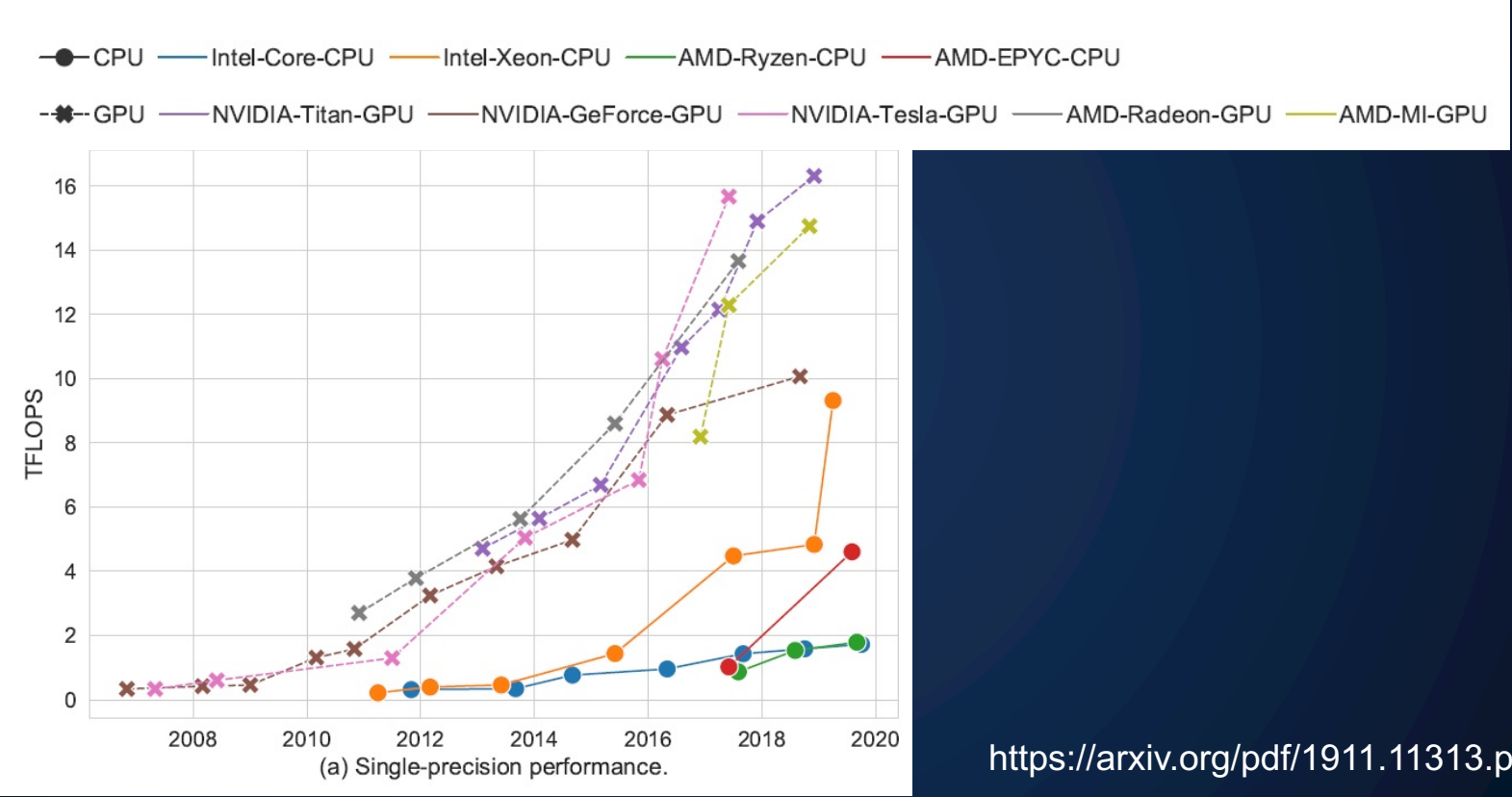
GPUs in Practice

hpcwire.com

GPUs: The Hype

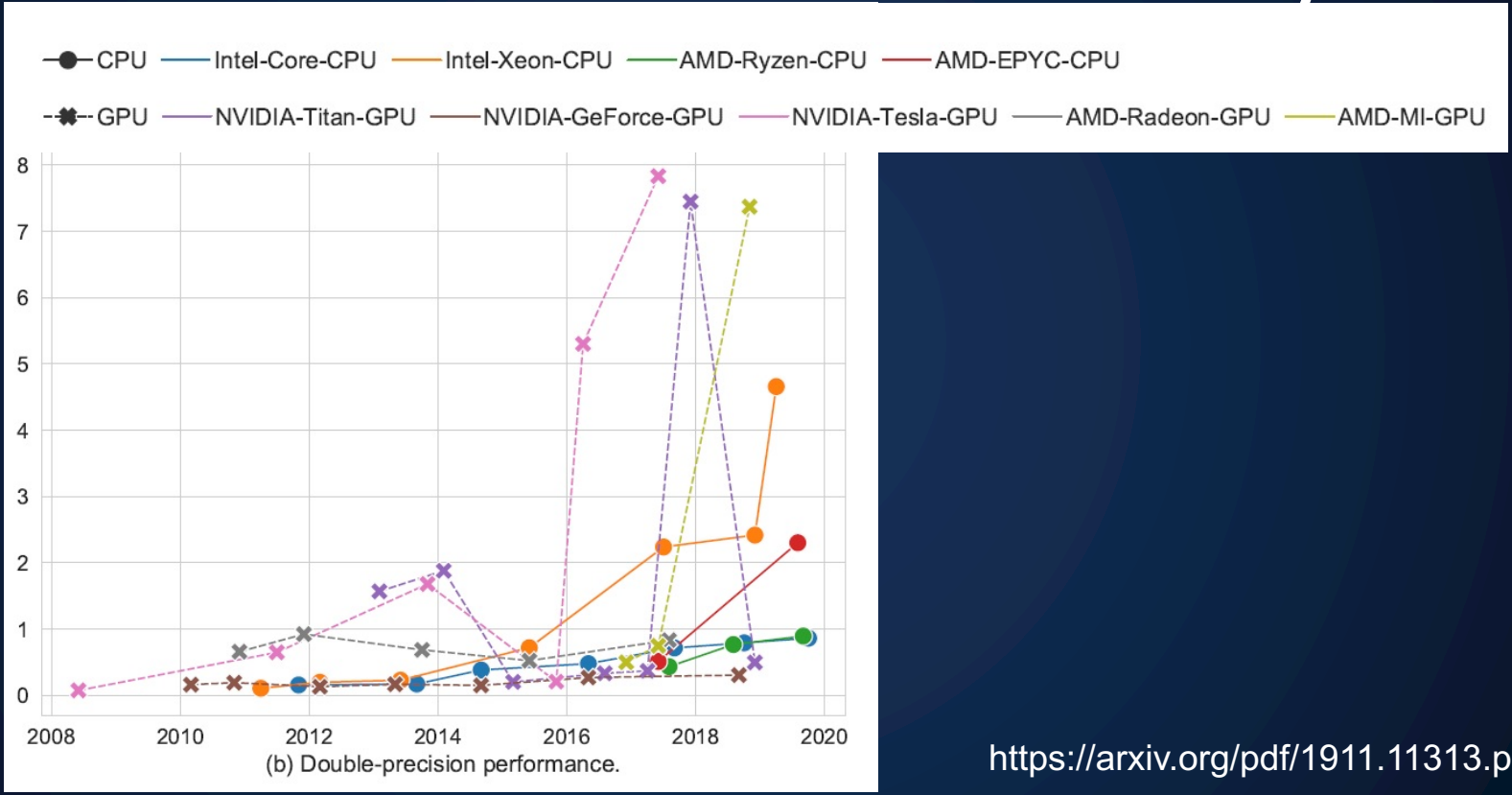


GPUs: A More Balanced Analysis



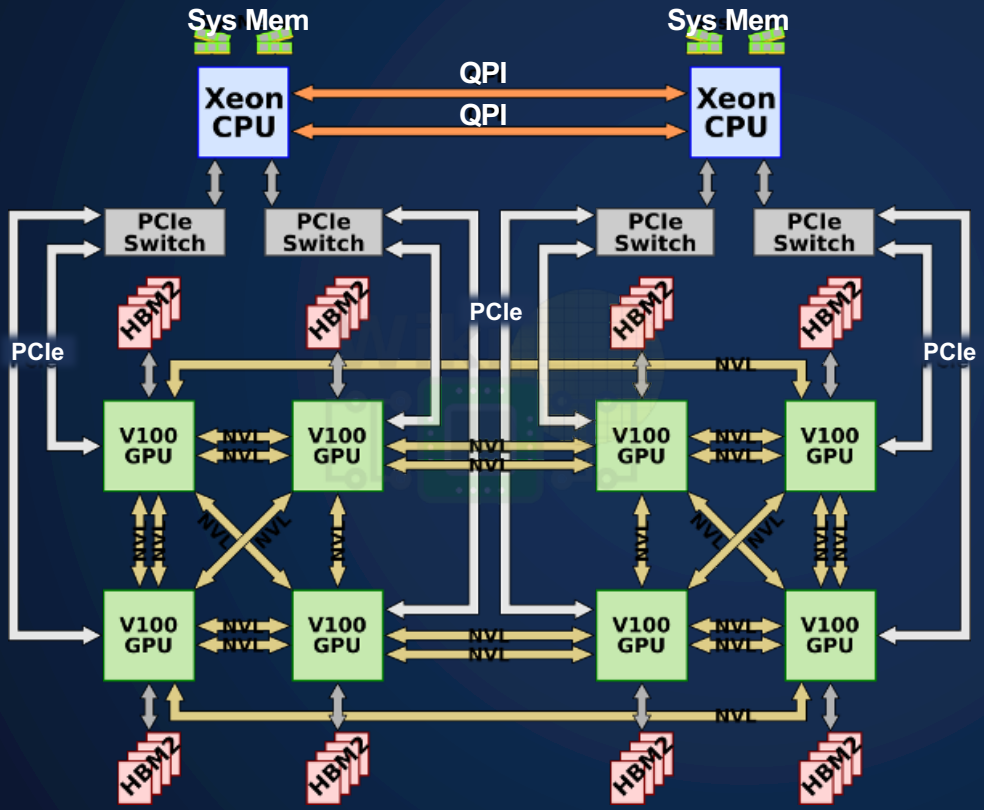
https://arxiv.org/pdf/1911.11313.pdf

GPUs: A More Balanced Analysis



https://arxiv.org/pdf/1911.11313.pdf

Cori GPU Node Architecture



Volta Streaming Multiprocessor (SM)

GV100

Each SM has:

- 64 FP32 cores
- 32 FP64 cores
- 64 KB registers
- 128 KB L1 + shared mem

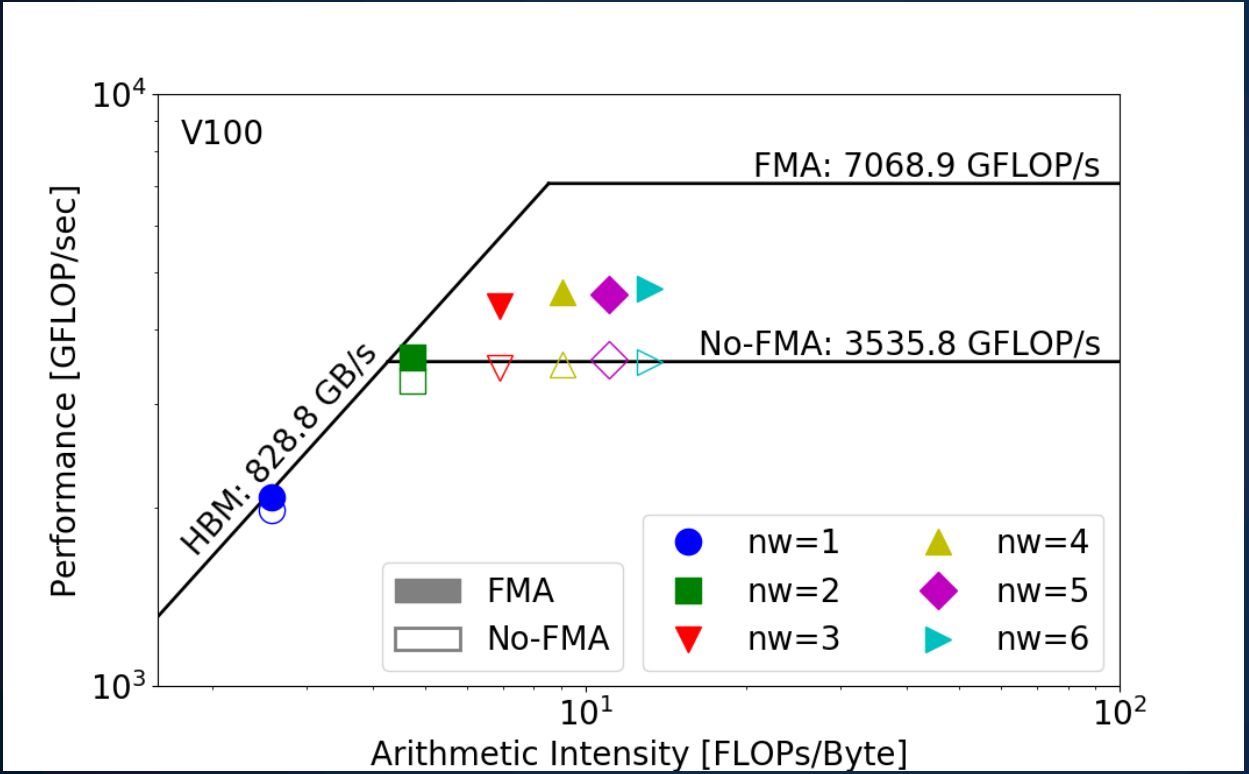
GPUs on Cori

There are 18 GPU nodes on Cori – use batch job submission and start early!

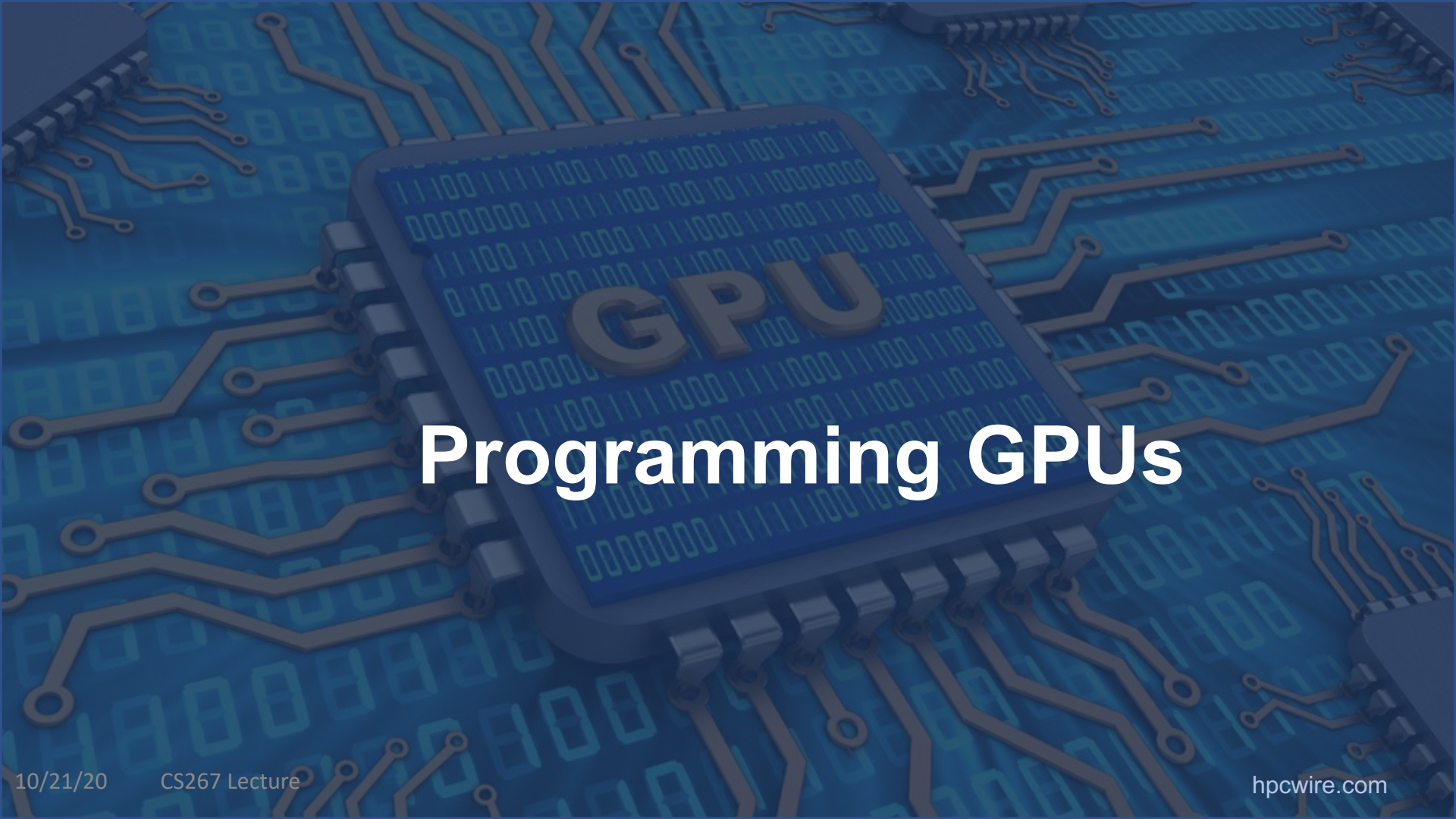
Each GPU node contains:

- 2 20-core Intel Xeon Gold 6148 ('Skylake') @ 2.40 GHz
- 384 GB DDR4 memory plus 930 GB on-node NVMe storage
- 8 NVIDIA V100 ('Volta') GPUs, each with with 16 GB HBM2 memory
  - Connected to each other with NVLink interconnect
- 4 dual-port InfiniBand network cards
  - Mellanox MT27800 (ConnectX-5) EDR

Roofline Model for V100



nw is a synthetic app, varying CI



Programming GPUs

hpcwire.com

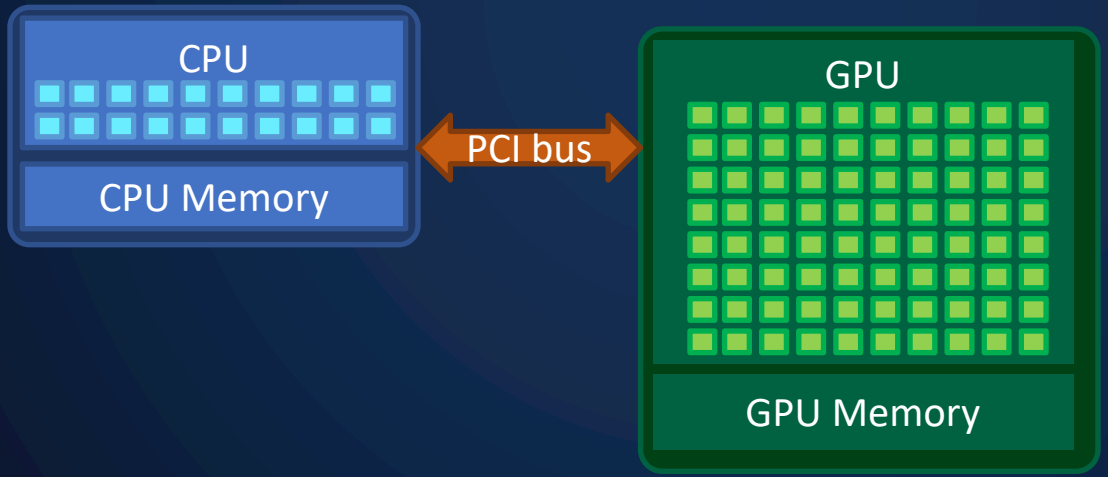
1 CPU + 1 GPU Nodes on Cori node

CPU (Skylake)

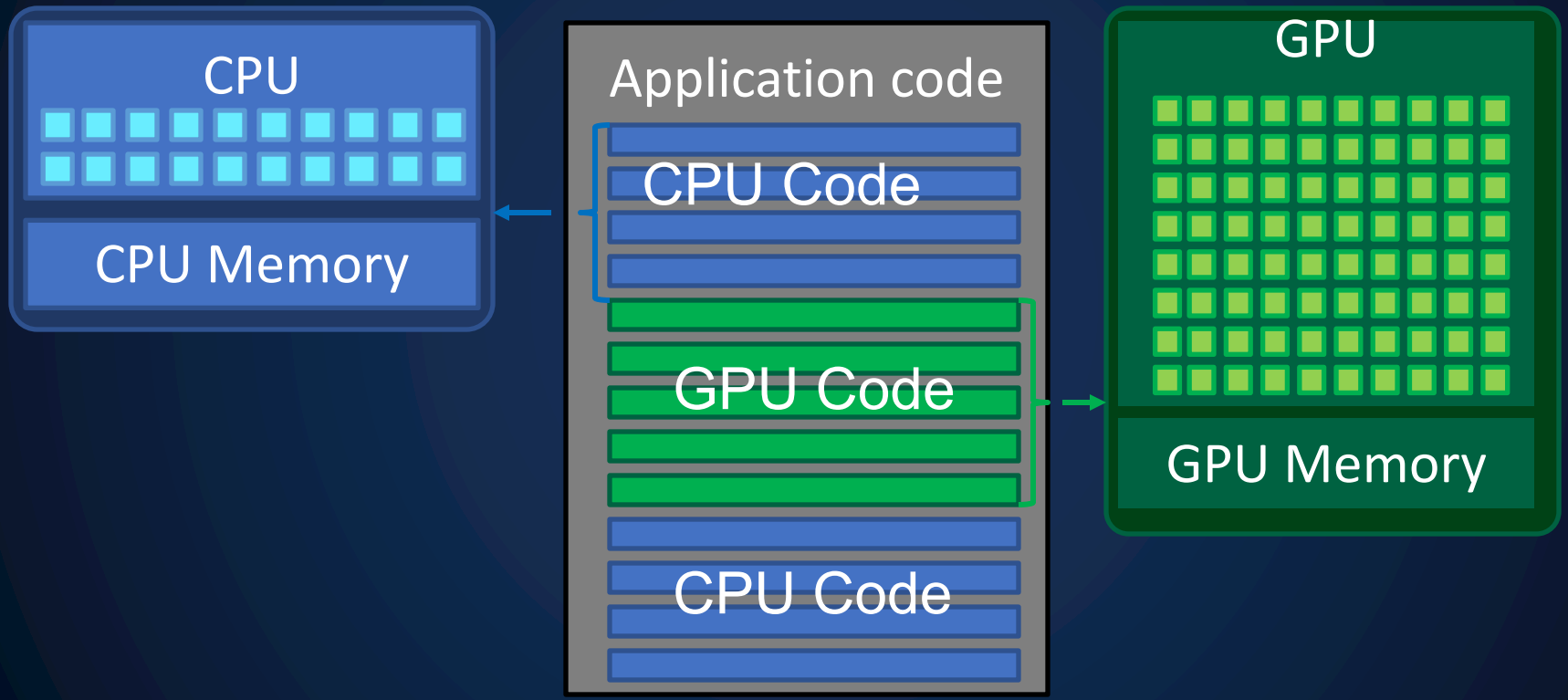
- 40 cores (2 20-core chips)
- 2 threads each
- 2xAVX-512, so 2x8 in double precision
- **~640** way parallelism (40*2*8)

GPU (V100)

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



Heterogeneous Programming (CPU+GPU)



Example: Vector Addition (CPU)

```
#include <iostream>
int main(void) {
  int N = 1<<20; // 1M elements
  float *x = new float[N]; // Allocate memory
  float *y = new float[N];

  // Free memory
  delete [] x; delete [] y;
  return 0;
}
```

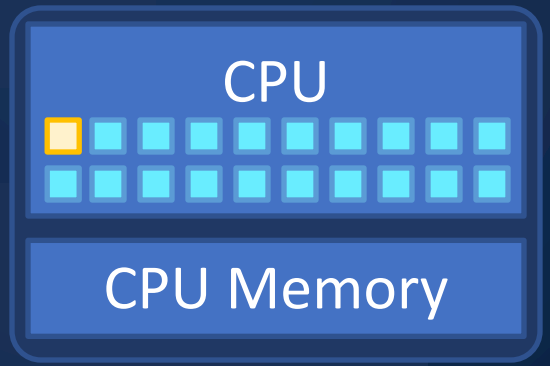
Example: Vector Addition (CPU)

```
#include <iostream>
int main(void) {
  int N = 1<<20; // 1M elements
  float *x = new float[N]; // Allocate memory
  float *y = new float[N];
  // initialize x and y on the CPU
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
  }



  // Free memory
  delete [] x; delete [] y;
  return 0;
}
```

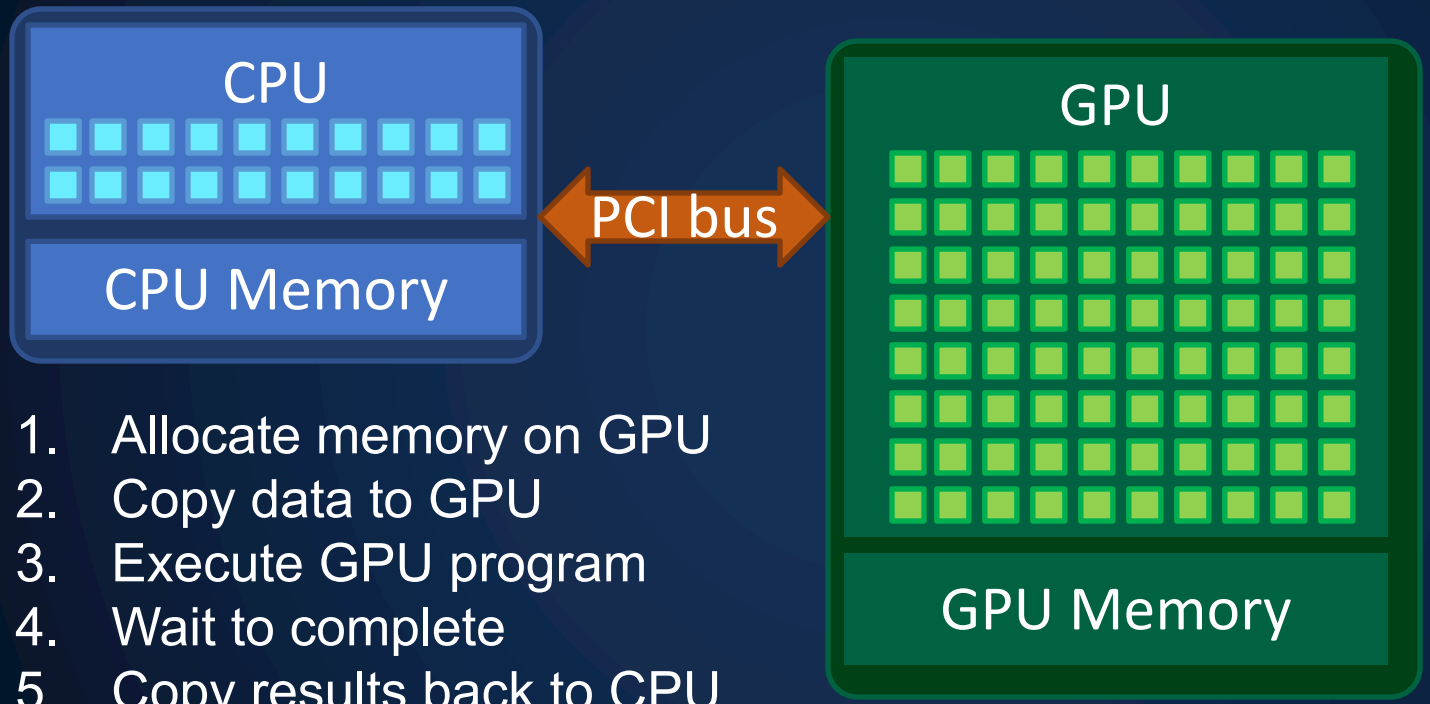
Example: Vector Addition (CPU)

```
#include <iostream>
int main(void) {
  int N = 1<<20; // 1M elements
  float *x = new float[N];
  float *y = new float[N];
  // initialize x and y on the CPU
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f; y[i] = 2.0f;
  }
  // Run on the CPU
  add(N, x, y);
  // Free memory
  delete [] x; delete [] y;
  return 0;
}
```

```
// CPU function to add two arrays
void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
```



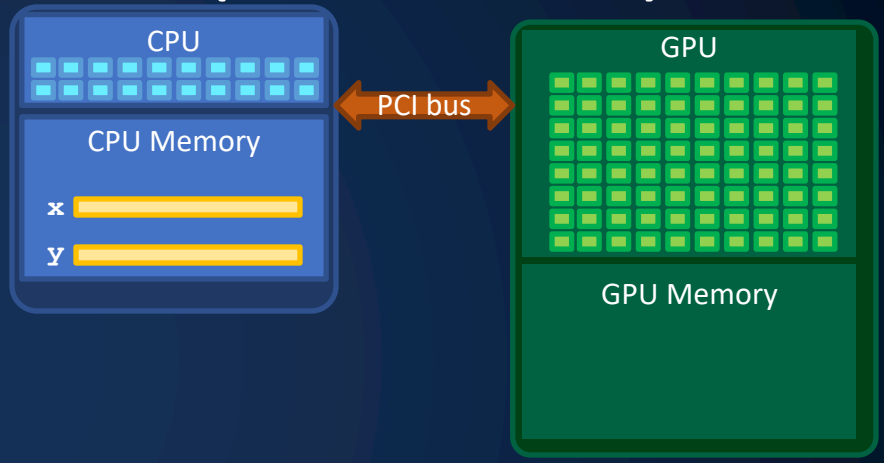
Running GPU Code (Kernel)



1. Allocate memory on GPU
2. Copy data to GPU
3. Execute GPU program
4. Wait to complete
5. Copy results back to CPU

Example: Vector Addition (GPU Serial)

```
float *x = new float[N];
float *y = new float[N];
// initialize x and y on the CPU
for (int i = 0; i < N; i++) {
  x[i] = 1.0f; y[i] = 2.0f;
}
```



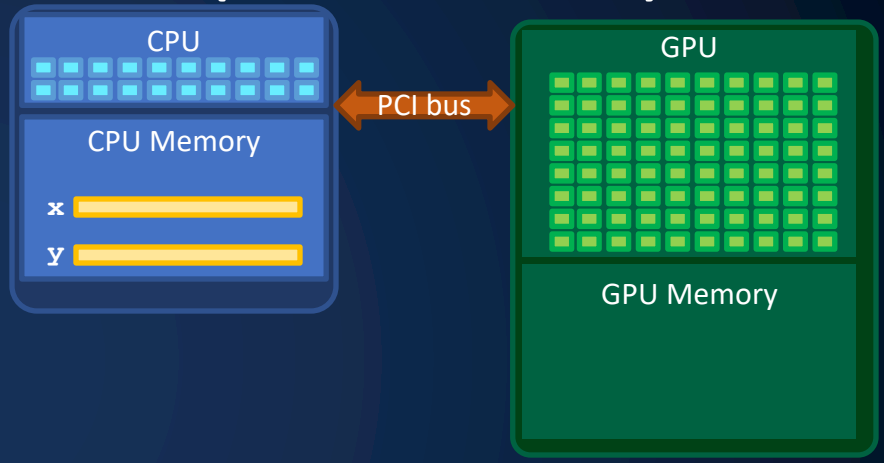

```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
delete [] x; delete [] y;
```

Example: Vector Addition (GPU Serial)

```
float *x = new float[N];
float *y = new float[N];
```

initialization not show

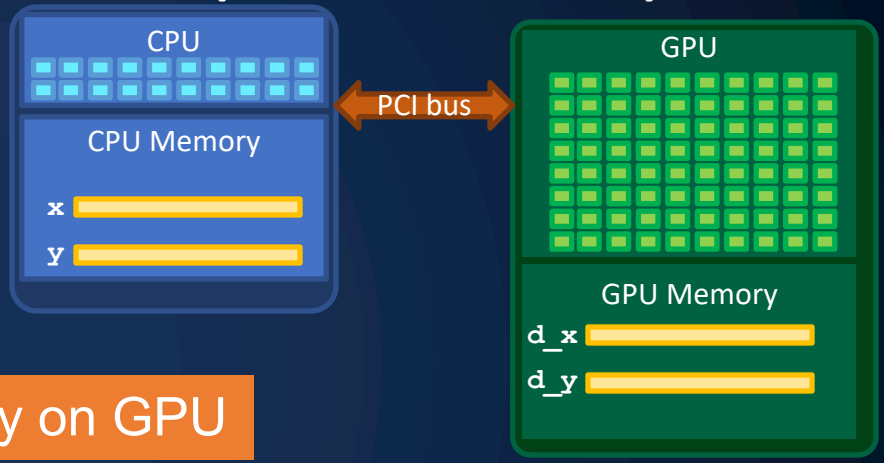



```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
delete [] x; delete [] y;
```

Example: Vector Addition (GPU Serial)

```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
```



1) Allocate Memory on GPU

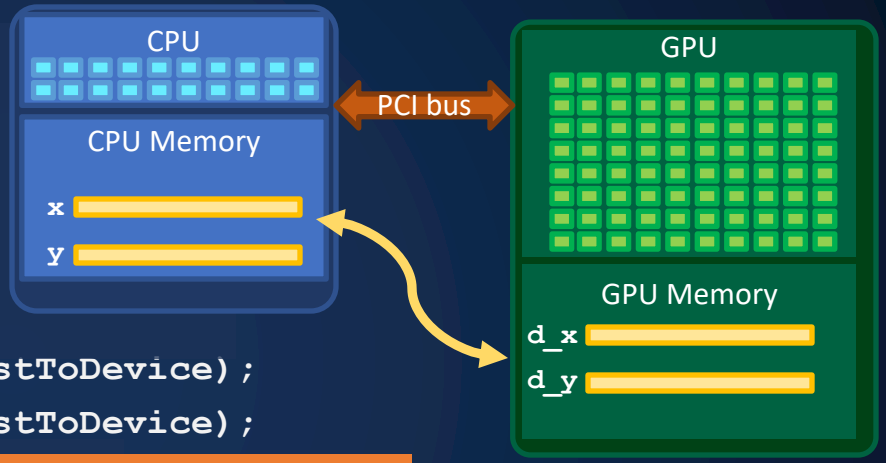
```
// Run on the CPU
add(N, x, y);
```

```
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

And free on GPU

Example: Vector Addition

```
float *x = new float[N];
float *y = new float[N];
int size = N*sizeof(float);
float *d_x, *d_y; // device copies of
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
// Run kernel on GPU
add<<<1,1>>>(d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```



2. Copy data to GPU

5. Copy results back to CPU

Example: Vector Addition

Function runs on GPU, callable from CPU

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}
```

```
float *x = n
float *y = n
int size = N
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
// Run kernel on GPU
add<<<1,1>>>(N, d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

3. Run on GPU
4. Wait to complete

Example: Vector Addition

```
float *x = n
float *y = n
int size = N
float *d_x, *d_y; // device copies of x y
cudaMalloc((void **)&d_x, size);
cudaMalloc((void **)&d_y, size);
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
// Run kernel on GPU
add<<<1,1>>>(N, d_x, d_y);
// Copy result back to host
cudaMemcpy(y, d_y, size, cudaMemcpyDeviceToHost);
// Free memory
cudaFree(d_x); cudaFree(d_y);
delete [] x; delete [] y;
```

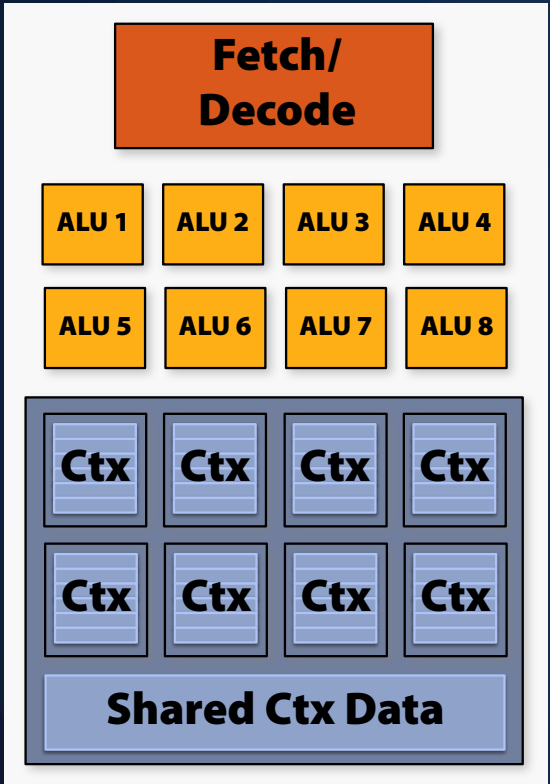
Function runs on GPU, callable from CPU

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  for (int i = 0; i < n; i++)
    y[i] = x[i] + y[i];
}
```

3. Run on GPU
4. Wait to complete

Use only 1 GPU thread

Programming as Threads



Program with multiple copies of program

Have each thread compute on different elements

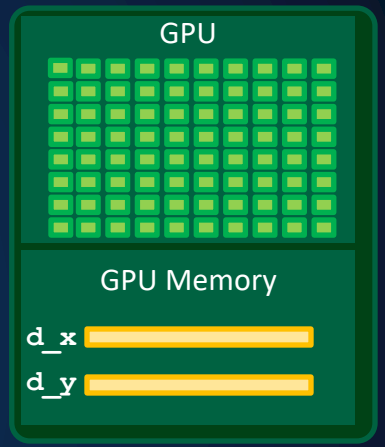
Ctx = Context

Example: Vector Addition (GPU)

```
// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);
```

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  int index = threadIdx.x;
  y[index] = x[index] + y[index];
}
```

1 block of
256 threads



CUDA's get thread it

Works iff N = 256!!

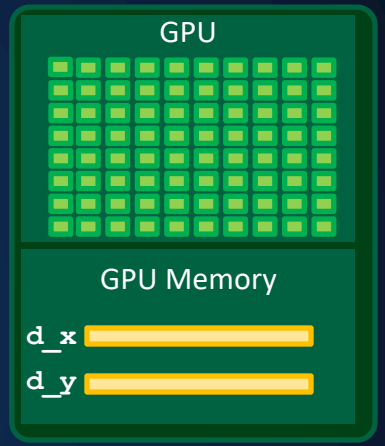
Example: Vector Addition (GPU)

```
// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);
```

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  int index = threadIdx.x;

  int stride = blockDim.x;
  for (int i = index; i < n; i+=stride)
    y[i] = x[i] + y[i];
}
```

1 block of *stride* threads

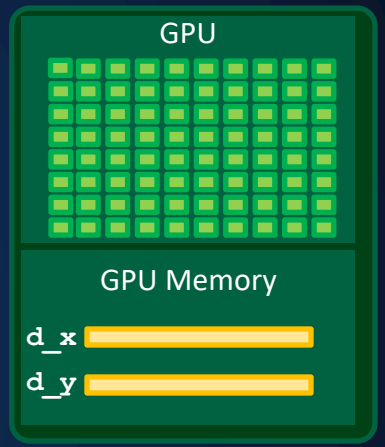
CUDA's # threads



Example: Vector Addition (GPU)

```
// Run kernel on GPU
add<<<1,256>>>(N, d_x, d_y);
```

Threads (256 here) must be ≤ 1024 on Volta (previously 512)




```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  int index = threadIdx.x;

  int stride = blockDim.x;
  for (int i = index; i < n; i+=stride)
    y[i] = x[i] + y[i];
}
```

1 block of *stride* threads

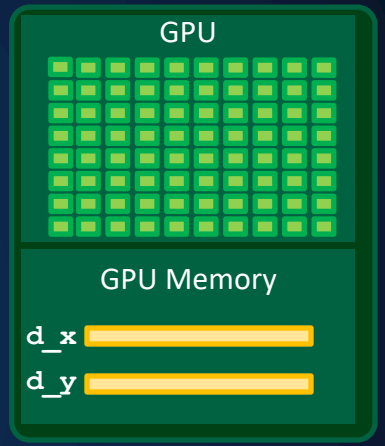
CUDA's # threads

Works for arbitrary N and # threads / block, only one block

Example: Vector Addition

```
// Run kernel on GPU
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

```
// GPU function to add two vectors
__global__
void add(int n, float *x, float *y) {
  int index = blockDim.x * blockIdx.x + threadIdx.x;

 int stride = gridDim.x;
 for (int i = index; i < n; i+=stride)
    y[i] = x[i] + y[i];
  }
}
```

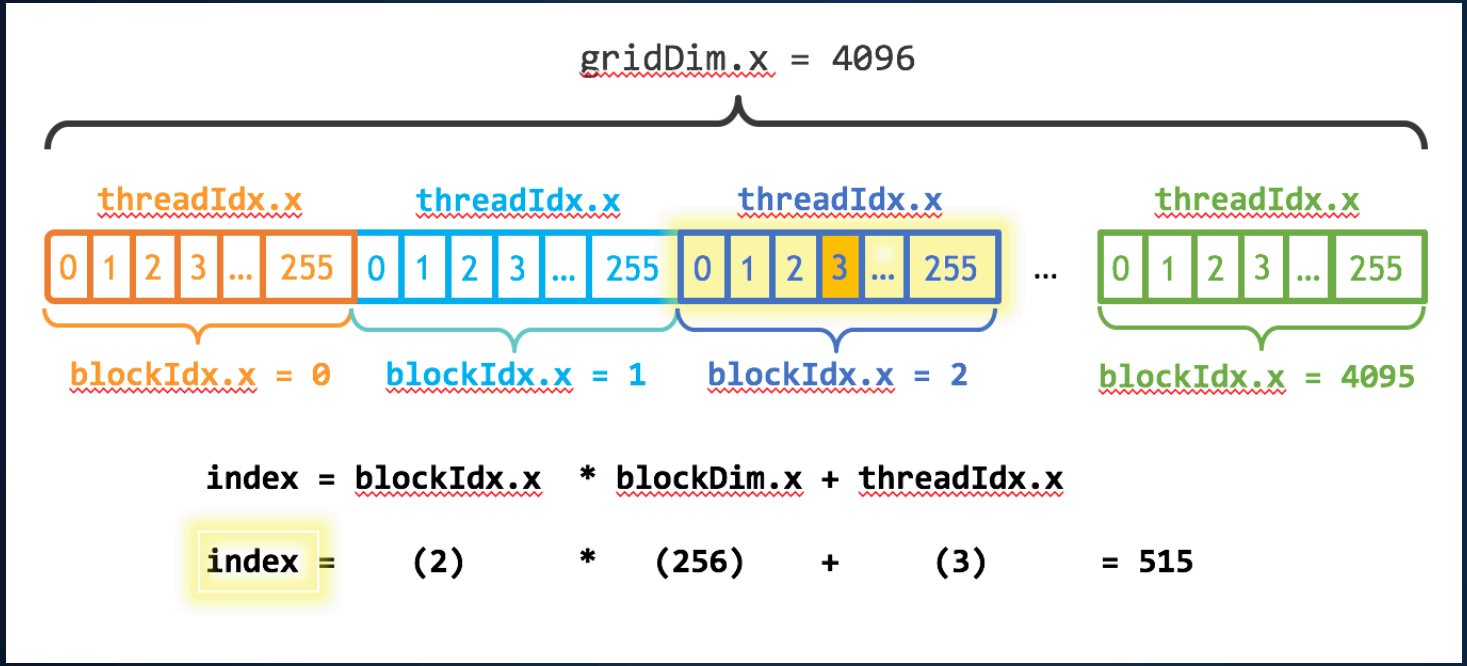


Works for arbitrary N and # threads / block

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

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

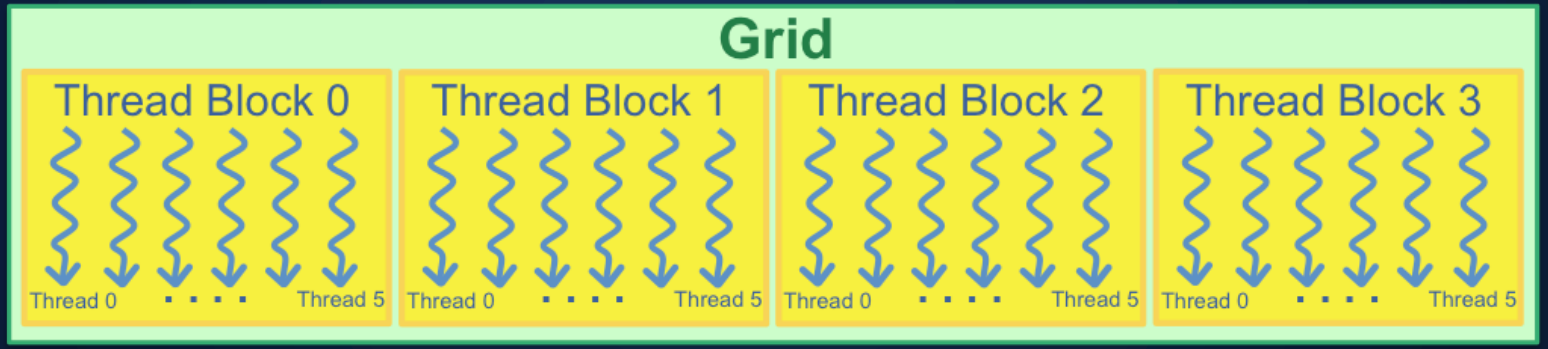
Blocks and Threads (and Grids)



https://developer.nvidia.com/blog/even-easier-introduction-cuda/

Grids and Thread Blocks

- A 1D Grid of 1D Blocks

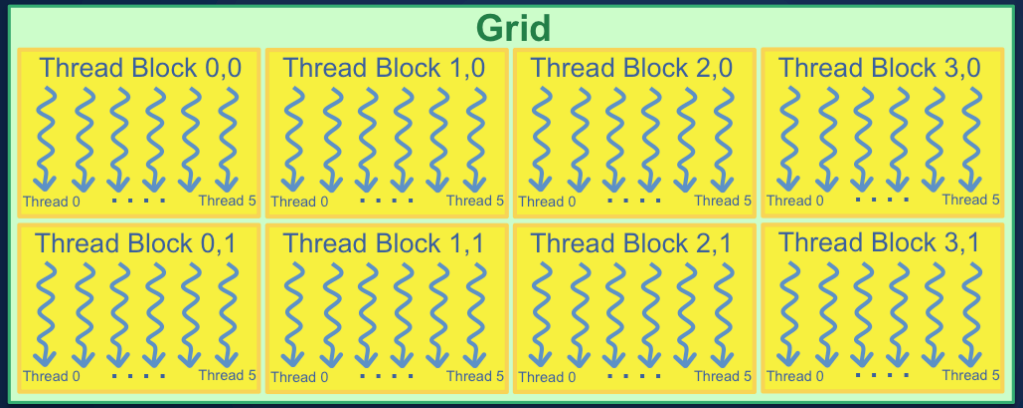


- blockIdx.x *blockDim.x + threadIdx.x;

https://cs.calvin.edu/courses/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf

Grids and Thread Blocks

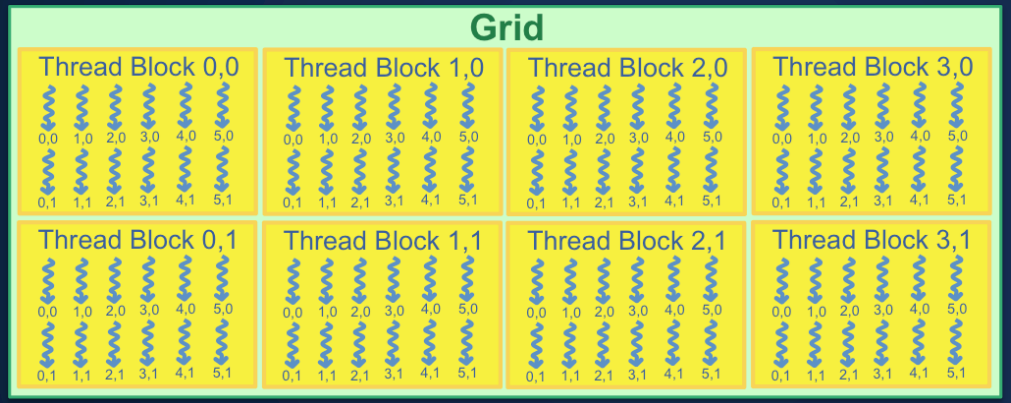
- A 2D Grid of 1D Blocks



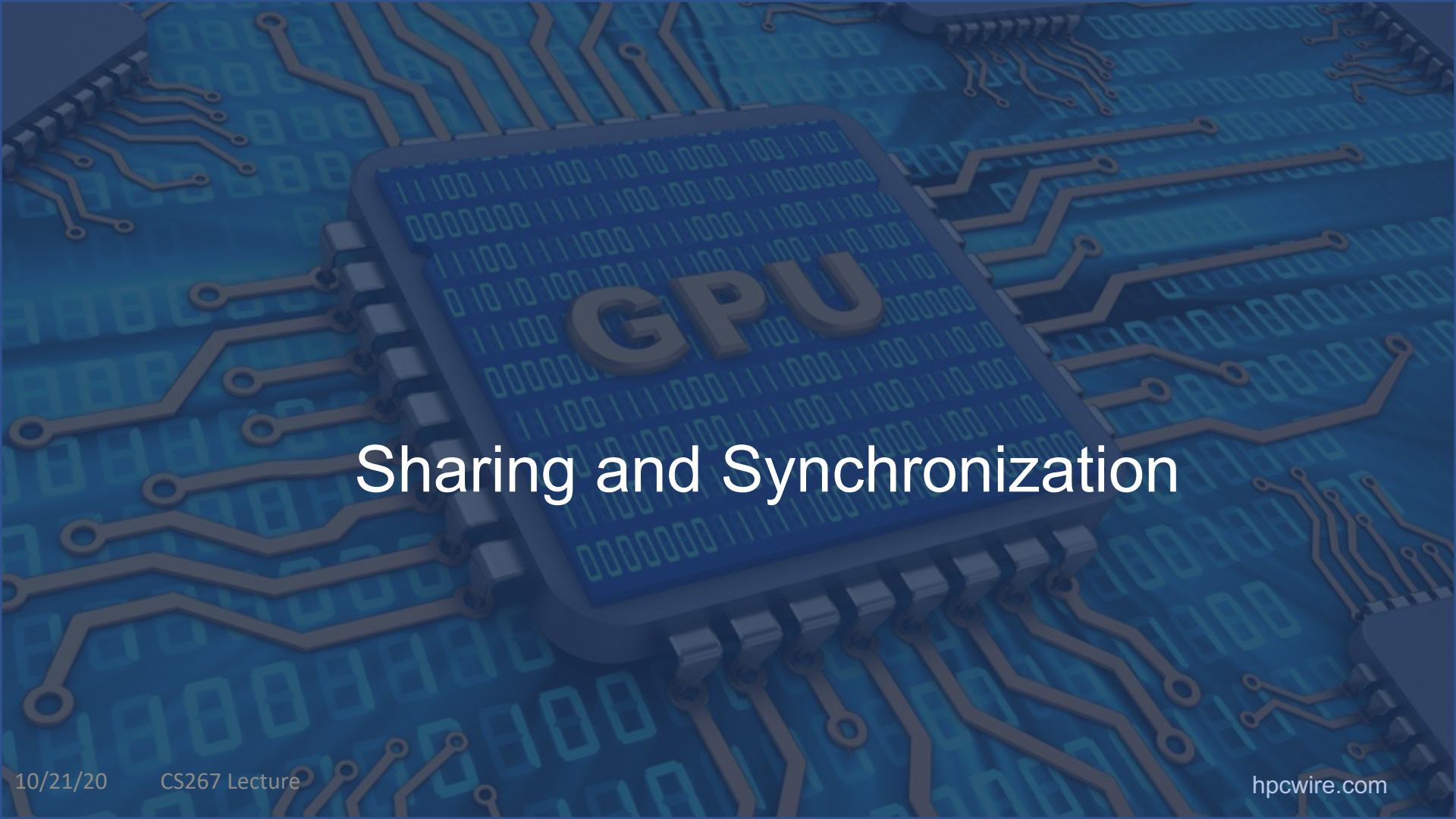

```
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int threadId = blockId * blockDim.x + threadIdx.x;
```

Grids and Thread Blocks

- A 2D Grid of 2D Blocks



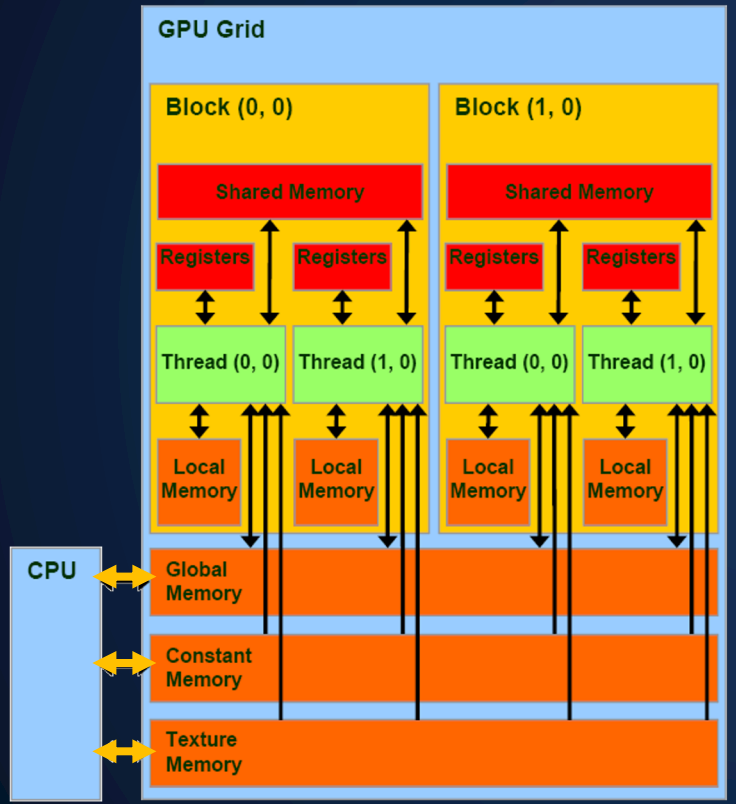

```
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y)
      + (threadIdx.y * blockDim.x) + threadIdx.x;
```



Sharing and Synchronization

hpcwire.com

Memory types on NVIDIA GPUs



- **Registers** per thread
- **Local** cached memory per thread
- **Shared** memory (shared in block)
- **Global** device level shared
- **Constant** cache shared by threads
- **Texture** cache shared by all block

Cost: local/global
100x slower

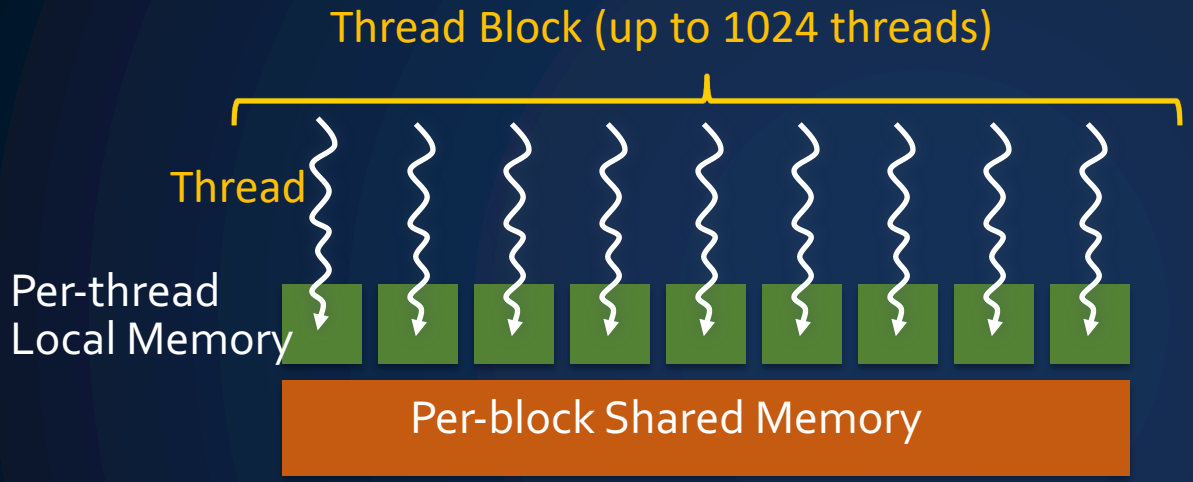
Hierarchical Parallelism Strategy

- Use both blocks and threads – Why?

```
kernel_call<<<blocks,threads>>>
```

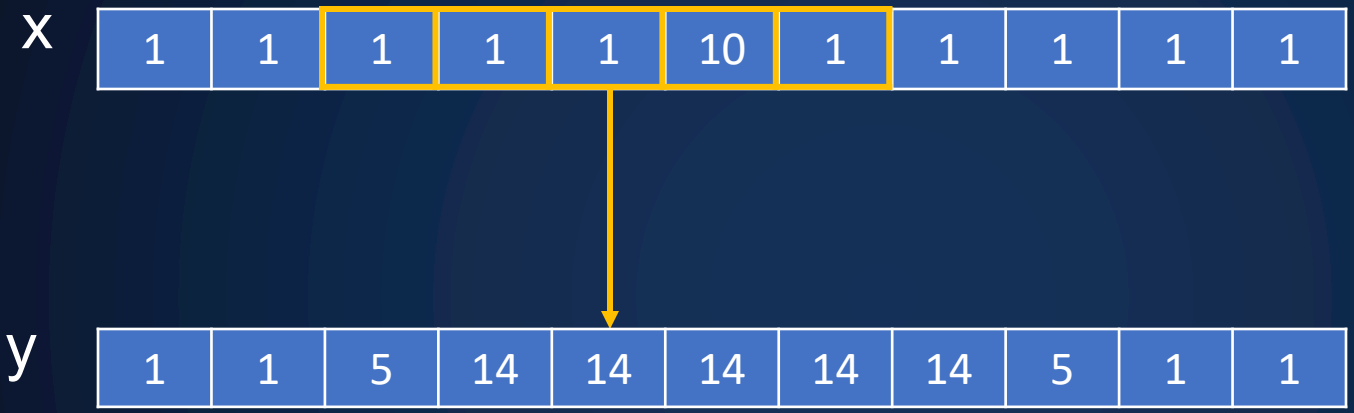
  - Limit on maximum number of threads/block
    - Threads alone won't work for large arrays
  - Fast shared memory only between threads
    - Blocks alone are slower

Shared (within a block) Memory



- Declare using __shared__, allocated per block
- Fast on-chip memory, user-managed
- Not visible to threads in other blocks

1D Stencil Example



y[i] = x[i] +
x[i-2] + x[i-1] +
x[i+2] + x[i+1]

1D 5-point stencil (with a “radius” of 2)

1D Stencil Example

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
- Reuse of inputs:
  - Radius of 2, each input element is read 5 times
  - Radius of 3, each input element is read 7 times

GPU thread strategy

Divide output array into blocks, each assigned to a thread block

- Each element within is assigned to a thread
- Compute blockDim.x output elements
- Write blockDim.x output elements to global memory

"owner computes"

What about the input?

GPU thread strategy

Cache (manually) input data in shared memory

- Have each block read **(**blockDim.x **+ 2 * radius)** input elements from global memory to shared memory
- Each block needs a halo of radius elements at each boundary
- (halos are also called ghost regions)

GPU thread strategy

The GPU Kernel will compute one element in the (global) output

```
int gindex = threadIdx.x + blockIdx.x * blockDim.x;
```

It will use a local index into the ”cached” copy of the input

```
int lindex = threadIdx.x + RADIUS
```

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {  // fill in halos
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
// to be continued
```

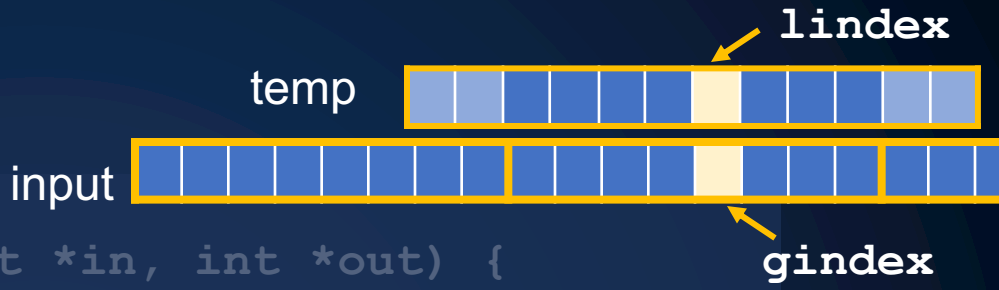
Stencil Kernel


```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {  // fill in halos
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
// to be continued
```

Stencil Kernel

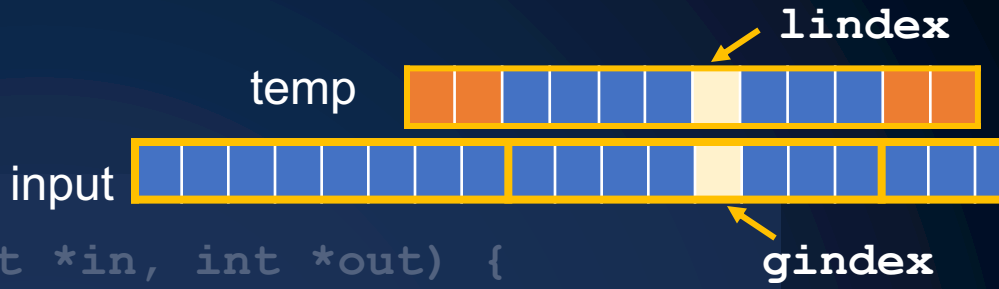

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {  // fill in halos
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
// to be continued
```

Stencil Kernel




```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {  // fill in halos
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
// to be continued
```

Stencil Kernel




```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;
  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {   // fill in halos
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
  }
// to be continued
```

Stencil Kernel


```
__global__ void stencil_1d(int *in, int *out) {
  // code from previous slide…

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
  // Store the result
  out[gindex] = result;
}
```

Stencil Kernel


```
__global__ void stencil_1d(int *in, int *out) {
  // code from previous slide…

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
  // Store the result
  out[gindex] = result;
}
```

Stencil Kernel


```
__global__ void stencil_1d(int *in, int *out) {
  // code from previous slide…

  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

Problem: Race Condition!

Suppose thread 7 (of 8) reads the halo before thread 0 has filled it in.

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS] = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

Problem: Race Condition!

Suppose thread 7 (of 8) reads the halo before thread 0 has filled it in.

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS] = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}
int result = 0;
result += temp[lindex + 1];
```

T0 writes temp[2]
T0 writes temp[0]
T7 reads temp[11]
T0 writes temp[11]

Thread Synchronization

- Synchronizes all threads within a block

```
void __syncthreads();
```

- Used to prevent RAW / WAR / WAW hazards
- All threads in the block must reach the barrier
- If used inside a conditional, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  // code from earlier slide to setup temp halos…
  // Synchronize (ensure all the data is available)
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];
  // Store the result
  out[gindex] = result;
}
```

Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared pointer with `atomicInc()`
  - Or use cooperative thread groups
- Implicit barrier between kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);
vec_dot<<<nblocks, blksize>>>(c, c);
```

Blocks must be independent

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

CUDA: Extensions to C/C++

- Declaration specifiers to indicate where things live

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

- Extend function invocation syntax for parallel kernel launch

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

- Special variables for thread identification in kernels

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

- Intrinsics that expose specific operations in kernel code

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

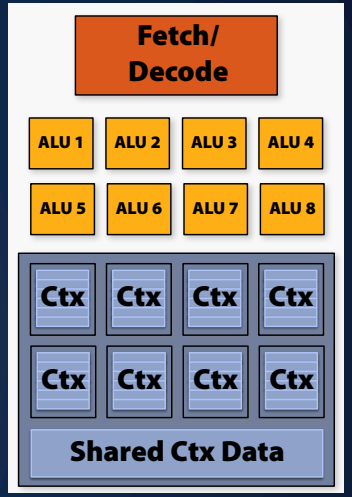
CUDA: Features available on GPU

- Double and single precision
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc(), cudaFree()`
  - `cudaMallocManaged();`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy(), cudaMemcpy2D(), ...`

Mapping CUDA to Nvidia GPUs



- Threads:
  - Each thread is a SIMD lane (ALU)
- Warps:
  - A warp executed as a logical SIMD instruction (sort of)
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
  - (Warp-level programming also possible)
- Thread blocks:
  - Each thread block is scheduled onto an SM
  - Peak efficiency requires multiple thread blocks per processor
- Kernel
  - Executes on a GPU (there is also multi-GPU programming)

Summary

- GPUs gain efficiency from simpler cores and more parallelism
  - Very wide SIMD (SIMT) for parallel arithmetic and latency-hiding
- Heterogeneous programming with manual offload
  - CPU to run OS, etc. GPU for compute
- Massive (mostly data) parallelism required
  - Not as strict as CPU-SIM (divergent addresses, instructions)
- Threads in block share faster memory and barriers
  - Blocks in kernel share slow device memory and atomics