Lecture #18 - GPU Programming

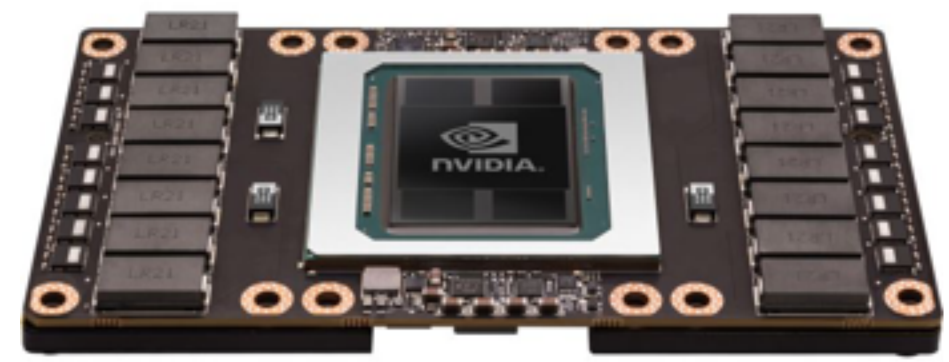
AMath 483/583

GPUs

- Well-suited for SIMD
- Massively parallel
  - many “cores”

    NVIDIA Tesla P100: *3584 cores*

  - specializes in latency hiding
- Highly energy efficient (Gflops / $ / Watt)

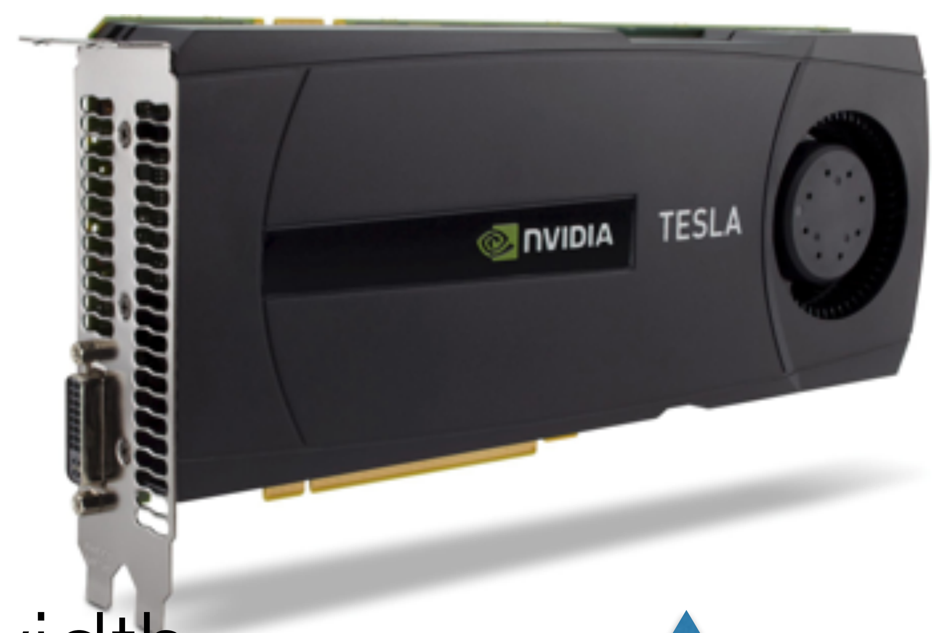


CUDA

- NVIDIA’s superset of C/C++
- Write CPU and GPU code in same source file
- Creates “Host Program” on CPU and “Device Program” on GPU.
  - communication between host and device similar to MPI

NVIDIA Tesla C2075

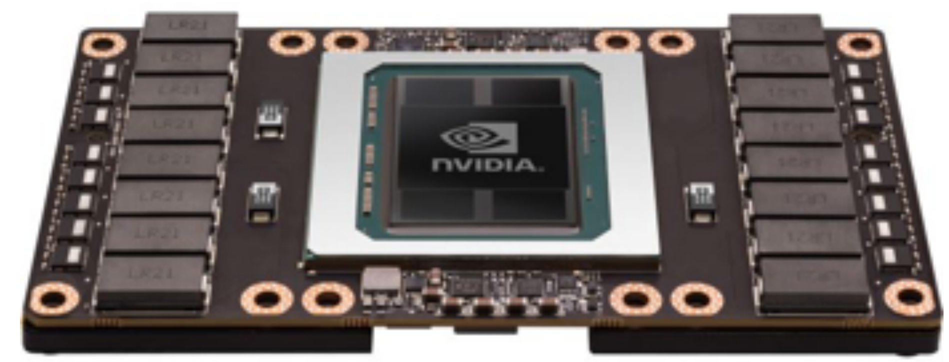
- 448 cores
- 1150 MHz clock
- 144 GB/sec mem. bandwidth
- 6 GB RAM
- 515 GFLOPS double precision



Used in this week's demos.

x4 on americano.amath

NVIDIA Tesla P100

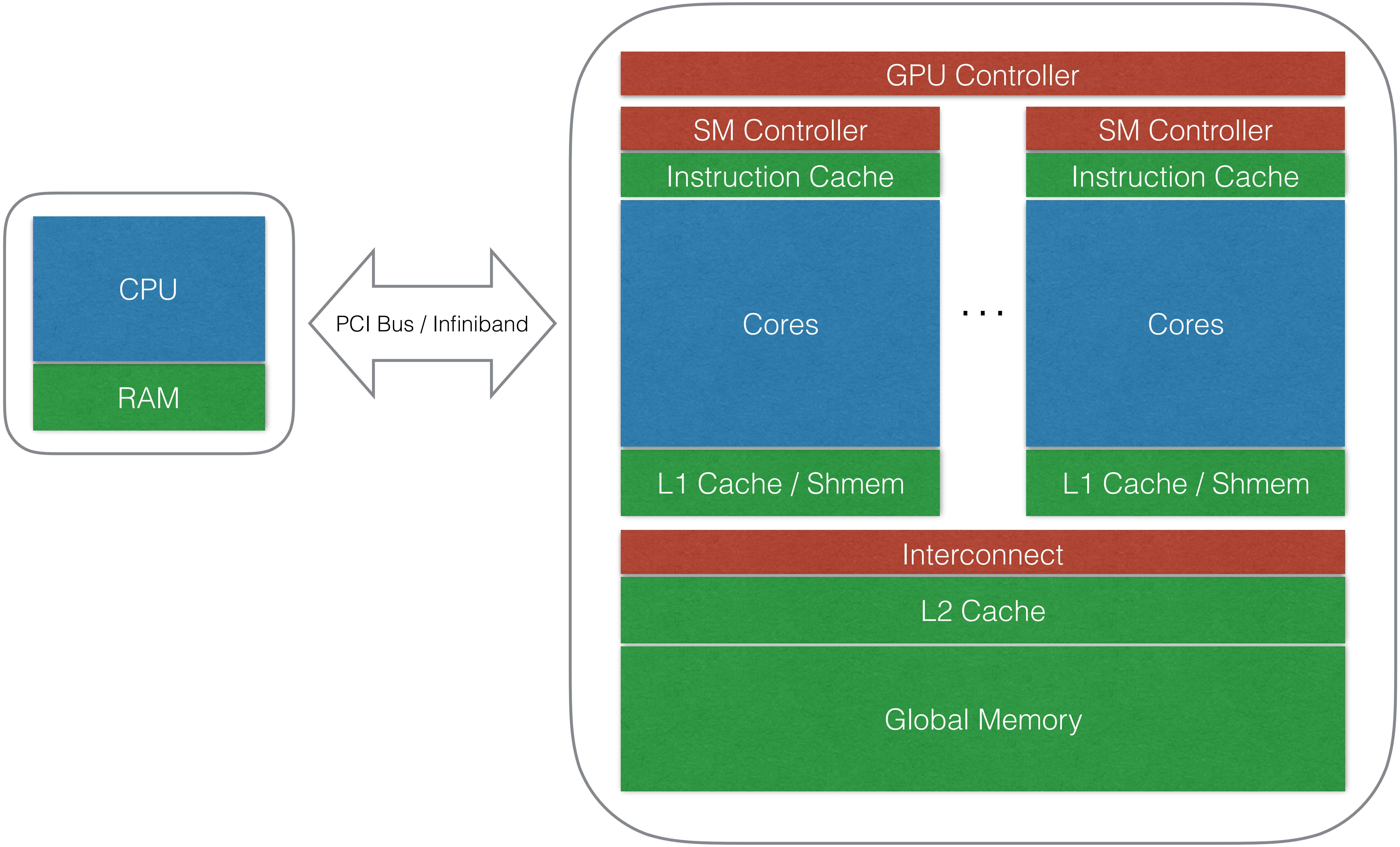


- 3584 cores
- 1328 MHz clock
- 720 GB/sec mem. bandwidth
- 16 GB RAM
- 5.3 TFLOPS double precision

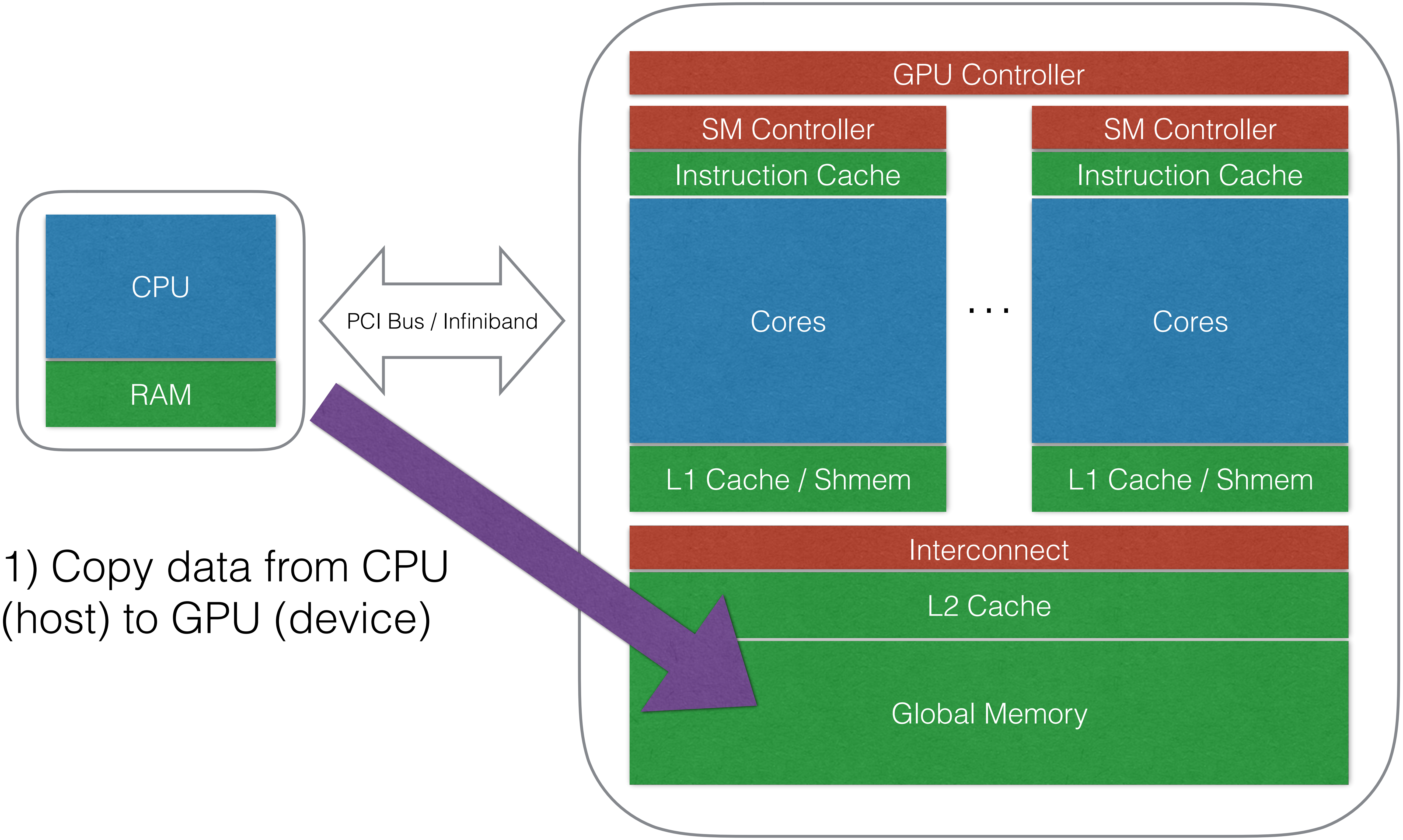
Computing power of top supercomputer in 2000

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

CPU vs. GPU

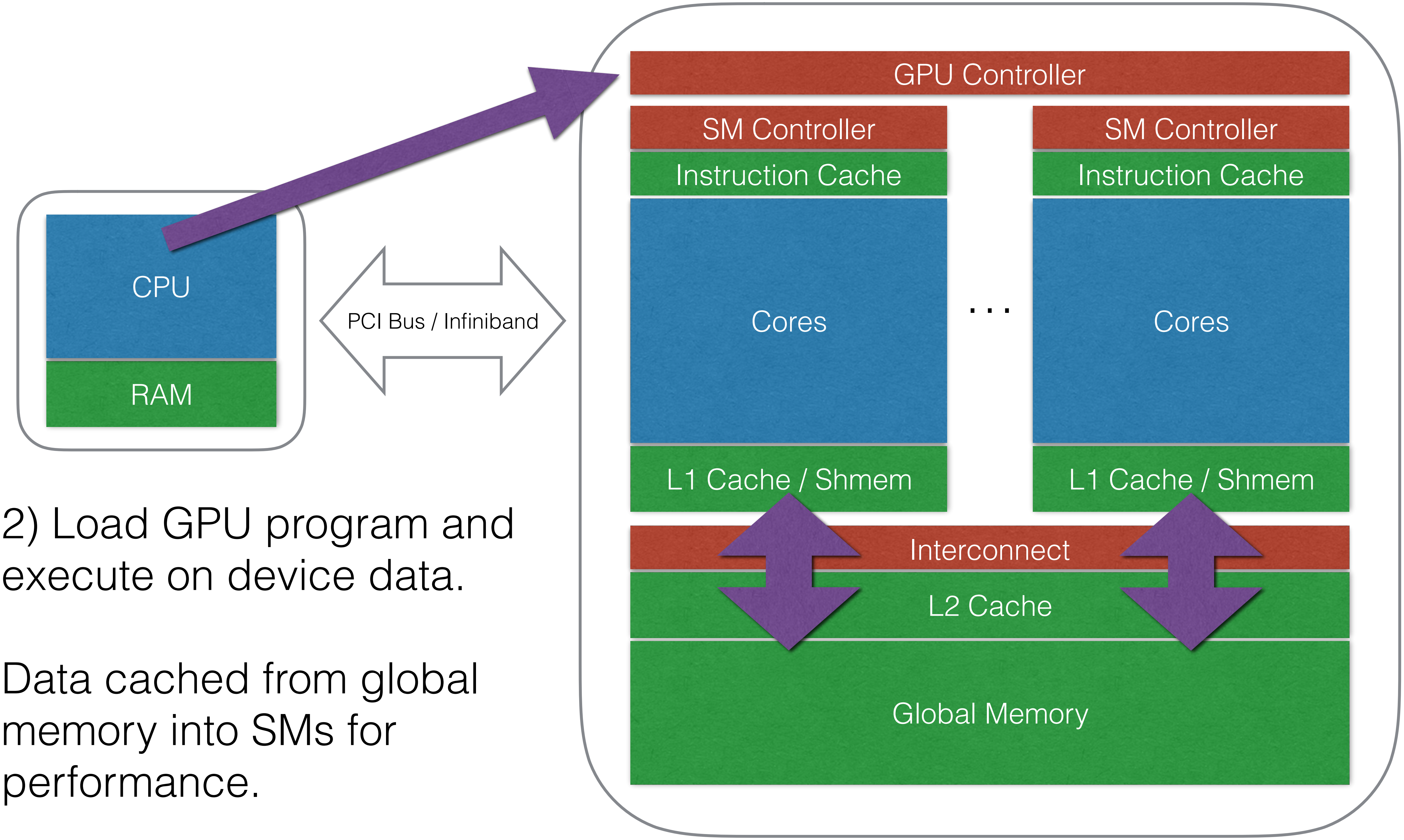

CPU
RAM
PCI Bus / Infiniband
GPU Controller
SM Controller
Instruction Cache
Cores
L1 Cache / Shmem
. . .
SM Controller
Instruction Cache
Cores
L1 Cache / Shmem
Interconnect
L2 Cache
Global Memory

Simple Processing Flow



1) Copy data from CPU (host) to GPU (device)

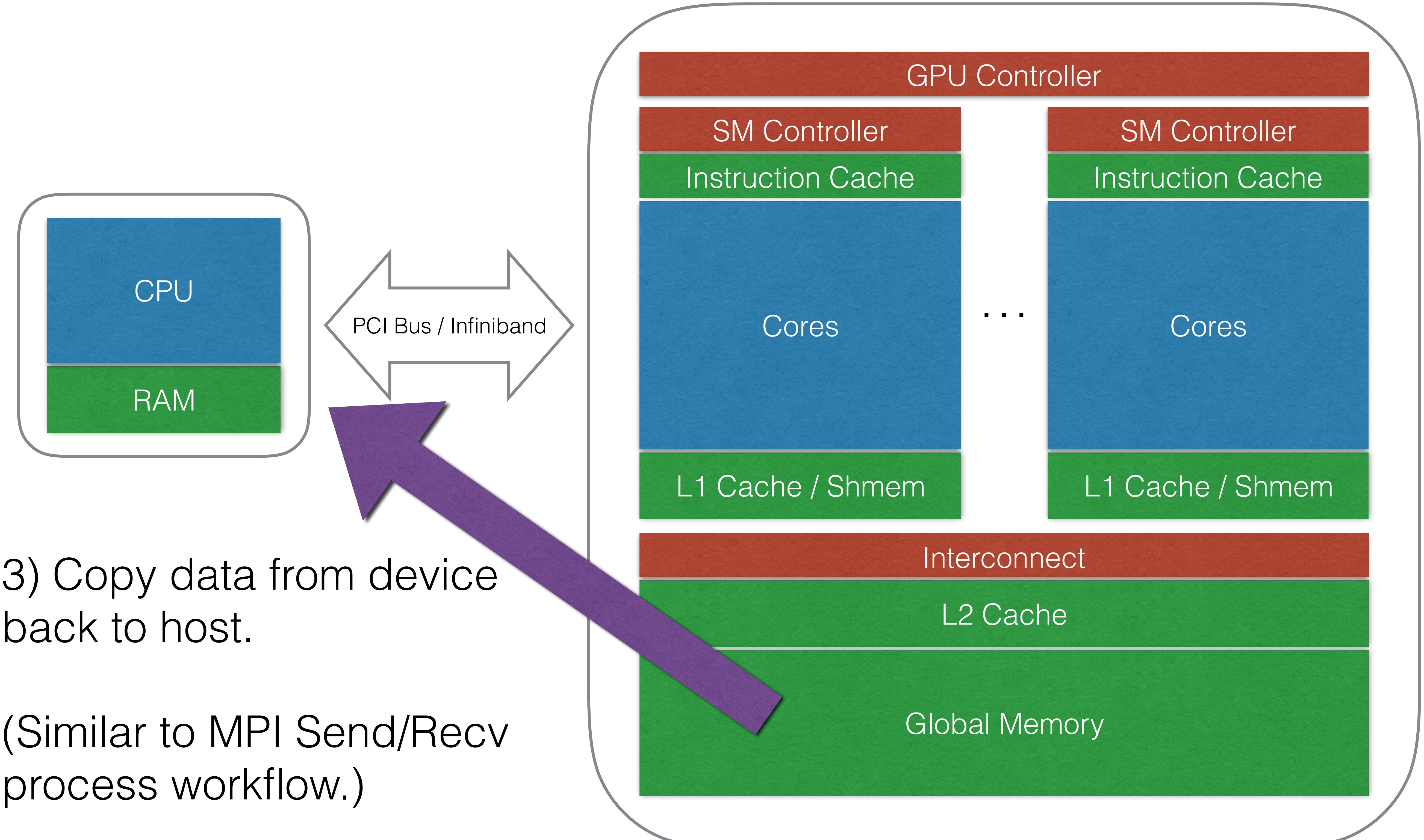
Simple Processing Flow



2) Load GPU program and execute on device data.

Data cached from global memory into SMs for performance.

Simple Processing Flow



3) Copy data from device back to host.

(Similar to MPI Send/Recv process workflow.)

Terminology

- *"Host-side"* — code running on CPU. Directs GPU calls and memory transfer.
- *"Device-side"* — code running on GPU. Only interacts with device-side memory.
- *"Kernel"* — a function that runs on GPU. Invoked by host-side call. (Sometimes, device-side call.)

Hello World

```
#include <cuda.h>

__global__ void cuda_hello(void) {
  // print a character buffer from the GPU!
  printf("Hello, world!");
}

int main(void) {
  // call the CUDA kernel from the GPU
  cuda_hello<<<1,1>>>();

  // wait for all CUDA kernels to finish
  cudaDeviceSynchronize();
  return 0;
}
```

95% of host-side functions defined here

Hello World

```
#include <cuda.h>

__global__ void cuda_hello(void) {
  // print a character buffer from the GPU!
  printf("Hello, world!");
}

int main(void) {
  // call the CUDA kernel from t
  cuda_hello<<<1,1>>>();

  // wait for all CUDA kernels to finish
  cudaDeviceSynchronize();
  return 0;
}
```

***Aside**: printing from GPU is bananas!*

<u>A CUDA "kernel"</u>

`__global__` indicates that this function is run on the GPU

Hello World

```
#include <cuda.h>

__global__ void cuda_he
  // print a character
  printf("Hello, world
}

int main(void) {
  // call the CUDA kernel from the GPU
  cuda_hello<<<1,1>>>();

  // wait for all CUDA kernels to finish
  cudaDeviceSynchronize();
  return 0;
}
```

Invocation of the Kernel

For now, the `<<<1,1>>>` just means to run the kernel on a single GPU thread.

Hello World

```
#include <cuda.h>

__global__ void cuda_hello(void) {
  // print a character buffer from the GPU!
  printf(“Hello, world!”)
}

int main(void) {
  // call the CUDA kernel from
  cuda_hello<<<1,1>>>();

  // wait for all CUDA kernels to finish
  cudaDeviceSynchronize();
  return 0;
}
```

Synchronize GPU Threads

With for GPU threads to finish. CPU can execute code in the meantime.

Compiling

- Use the Nvidia CUDA C compiler

  ```
  $ nvcc -arch=sm_20 hello.cu
  $ ./a.out
  “Hello, world!”
  ```

- Different “compute capabilities” on different GPUS. (Compile targets different hardware levels.)

  “`-arch=sm_53`” = Maxwell features

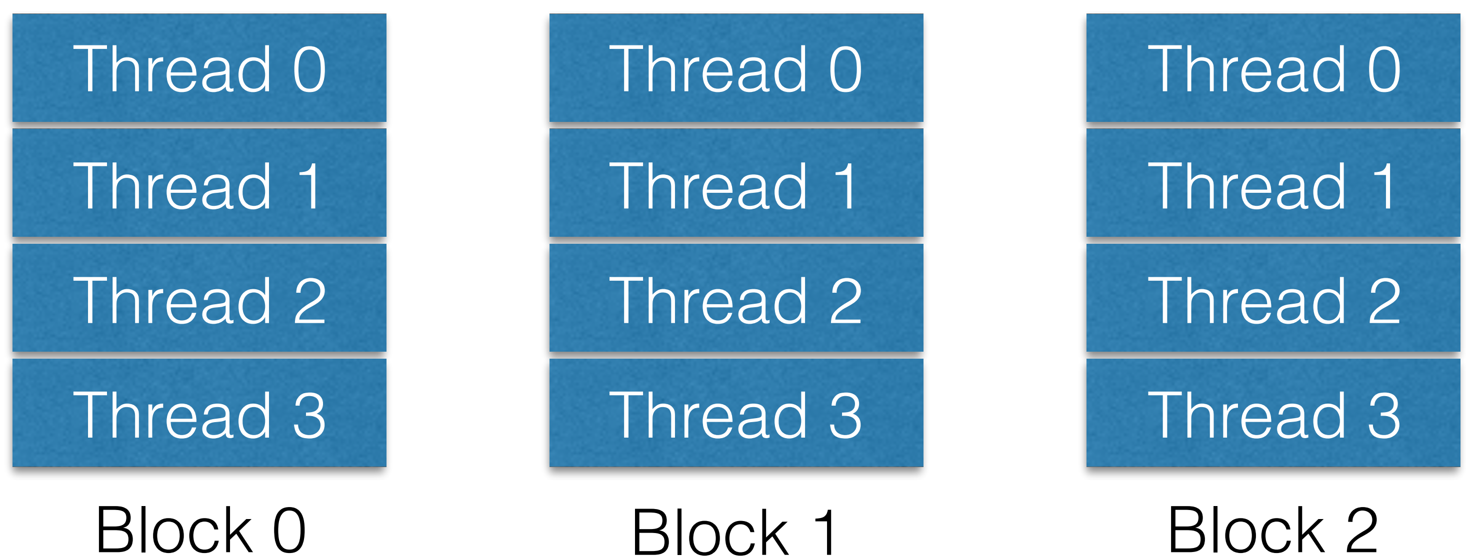
Demo

`hello.cu`

GPU Parallelism

- Shared Memory Environment (e.g. OpenMP)

- Thread organization:



- Set thread / block count using triple chevrons

```
my_kernel<<<nblocks, nthreads_per_block>>>(…)
```

Threads and Block

- Obtain current thread:

```
int tid = threadIdx.x;
```

- Obtain current block:

```
int bid = blockIdx.x;
```

Hello World - Revisited

```
#include <cuda.h>

__global__ void cuda_hello(void) {
  int tid = threadIdx.x;
  printf(“Hello, from thread %d\n”, tid);
}

int main(void) {
  cuda_hello<<<1,4>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Run with 1 block, 4 threads.

Demo

`hello.cu`

Run with 2 blocks and 4 threads per block. “*parallelism coordinate*”

Vector Addition

- Recall three-step memory flow:
  1. **Copy data from host to device**
  2. **Compute on device**
  3. **Copy results back to host**
- Host memory separated from device memory
  - host pointers point to CPU memory
  - device pointers point to GPU memory

vec_add - Device Side

- For now, assume vectors of length N and we spawned 1 block, N threads

```
__global__ void
vec_add(int* out, int* v, int* w)
{
  size_t index = threadIdx.x;
  out[index] = v[index] + w[index];
}
```

vec_add - Device Side

- For now, assume vectors of length N and we spawned 1 block, N threads

Pointers to device-side memory.

```
__global__ void
vec_add(int* out, int* v, int* w)
{
  size_t index = threadIdx.x;
  out[index] = v[index] + w[index];
}
```

Determine vector index from thread / block indices.

`vec_add` - Allocating Device Memory

- Device-side analogues of common C memory funs

```
cudaMalloc(&dev_ptr, size);
cudaMemcpy(dev_ptr, host_ptr, size,
           direction);
cudaFree(dev_ptr);
```

Copies data to/from host from/to device. (Depending on `dir`.)

vec_add - Allocating Device Memory

1) Allocate some host-side data:

Storing the array size (not just length) will be useful later

```
int N = 16;
size_t size = N*sizeof(int);
int* host_v = (int*) malloc(size);
int* host_w = (int*) malloc(size);

// populate host_v and host_w with data
// …
```

`vec_add` - Allocating Device Memory

2) Allocate some corresponding device-side data:

```
int* dev_v;
int* dev_w;
int* dev_sum;

// note: cudaMalloc wants ptr to ptr
cudaMalloc((void**) &dev_v, size);
cudaMalloc((void**) &dev_w, size);
cudaMalloc((void**) &dev_out, size);
```

`vec_add` - Allocating Device Memory

2) Allocate some corresponding device-side data:

```
int* dev_v;
int* dev_w;
int* dev_sum;
```

These don't actually point to anywhere in RAM or GPU memory. Just for reference.

```
// note: cudaMalloc wants ptr to ptr
cudaMalloc((void**) &dev_v, size);
cudaMalloc((void**) &dev_w, size);
cudaMalloc((void**) &dev_out, size);
```

`vec_add` — Allocating Device Memory

- 3) Copy host-side data to device using dev ptrs.

```
cudaMemcpy(dev_v, host_v, size,
           cudaMemcpyHostToDevice);
cudaMemcpy(dev_w, host_w, size,
           cudaMemcpyHostToDevice);
```

- **Remember**: data communication to GPU is explicit (like MPI)

`vec_add` — Execute GPU Kernel on Device Data

- 4) Launch Kernel on N threads and wait to finish.

```
vec_add<<<1,N>>>(dev_sum,dev_v,dev_w);

cudaDeviceSychronize();
```

- **Note**: number of threads spawned = problem size
  - (no need to pass vector length)

`vec_add` — Copy Results Back to Host

- Remember to free device-side data:

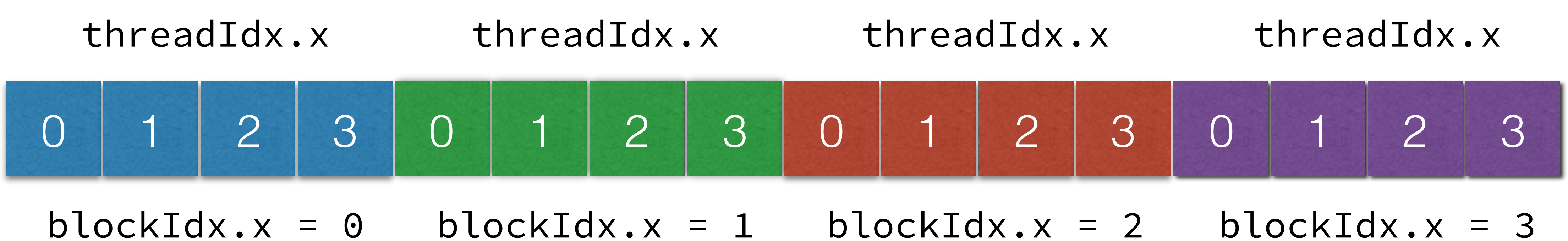
```
cudaMemcpy(host_sum, dev_sum, size,
           cudaMemcpyDeviceToHost);

cudaFree(dev_v);
cudaFree(dev_w);
cudaFree(dev_sum);

// … use results in host_sum …
```

Demo

`vec_add.cu`

Combining Blocks and Threads

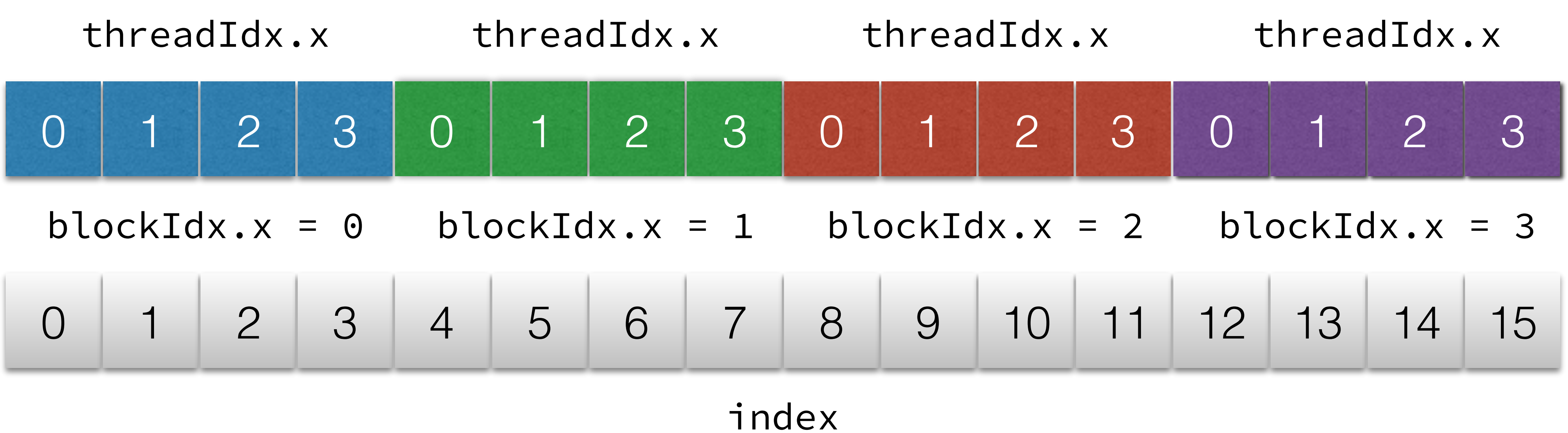
- Each block works on a “chunk” of the vector
- Each block thread adds across an index



Combining Blocks and Threads

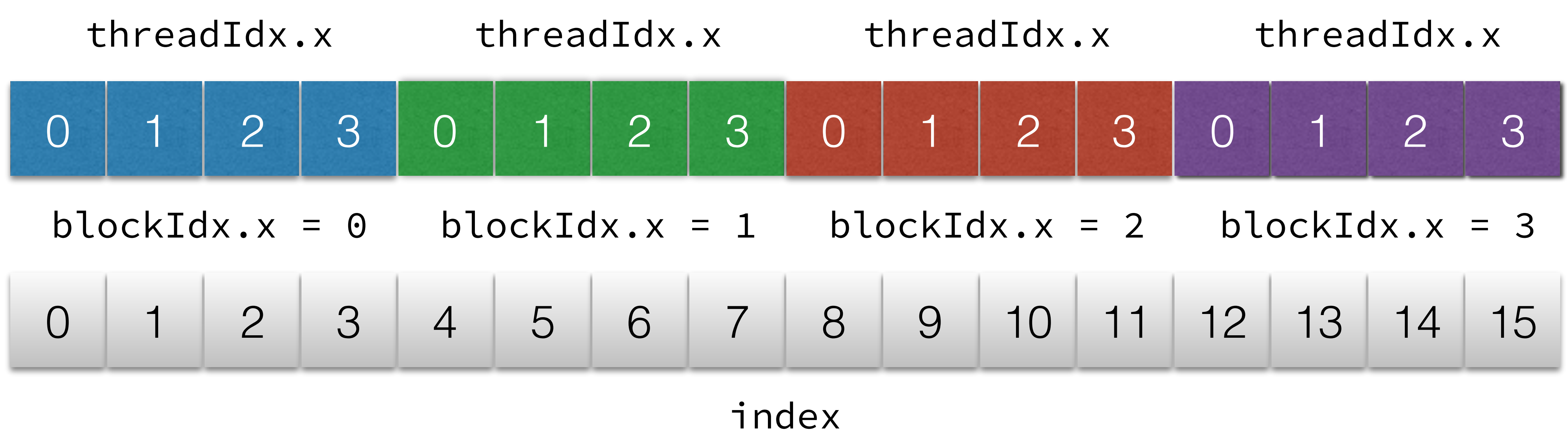
- Compute "global index" = coordinate of vector as a function of `threadIdx.x` and `blockIdx.x`

- `blockDim.x` = number of allocated blocks



Combining Blocks and Threads

```
size_t index;
index = threadIdx.x +
        blockIdx.x*blockDim.x;
```



vec_add — Flexible Kernel

- Works for any combination of threads / blocks:

```
__global__ void
vec_add(int* out, int* v, int* w)
{
  size_t index = threadIdx.x +
                 blockIdx.x * blockDim.x;
  out[index] = v[index] + w[index];
}
```

Demo

`vec_add.cu` with variable block/kernel

Why Bother?

- Within each block threads can
  - **communicate** —
    - *fast "shared memory" between threads within a block (lives in L1 memory)*
    - *otherwise, slow global memory access*
  - **synchronize** — *race cond. and branches*

Why Bother?

- **Latency hiding** — while one block makes mem. request another block can run on SM
  - “block swapping” — hide transfer times
- **Hardware Restrictions** (C2075)
  - 1536 threads / SM
  - 8 blocks / SM
  - total 48 KB shmem / SM (shared among blocks)
  - 1024 threads / block

Why Bother?

- **Latency hiding** — while one block makes mem. request another block can run on SM
    - “block swapping” — hide tran…
- **Hardware Restrictions** (…
    - 1536 threads / SM
    - 8 blocks / SM
    - total 48 KB shmem / SM …ared among blocks)
    - 1024 threads / block

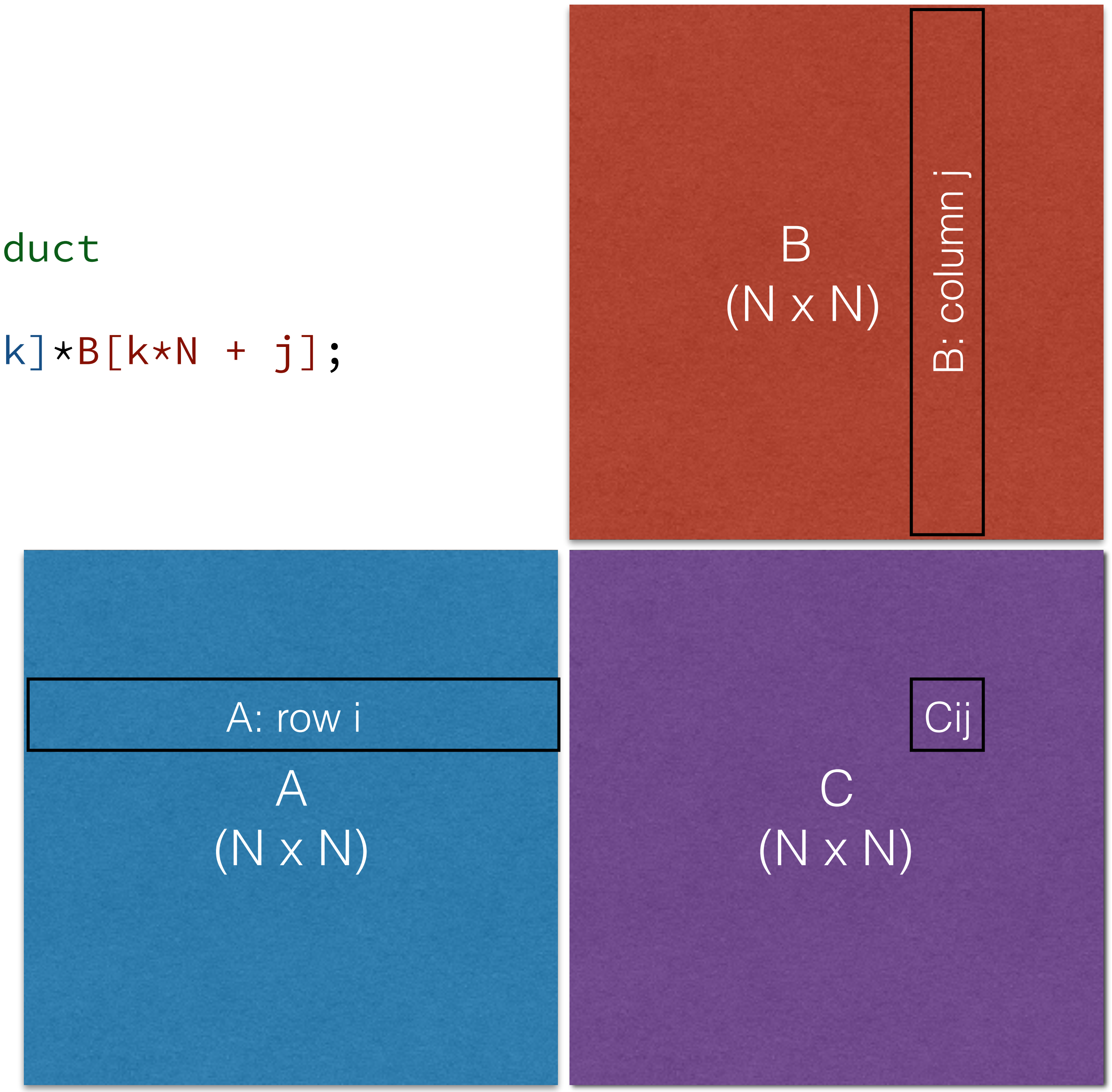
Recommend 128, 160, 192, 256 threads / block. (Multiples of 32)

Warps

- **Key GPGPU Difference** — each SM splits blocks into "warps" of 32 threads. <u>*All threads in a warp execute concurrently.*</u>
  - warp waits until 32 cores are available on SM
  - *if a block contains 48 threads*: 32 in one warp 16 in another (+16 no-op threads)
  - performance issues: contiguity, sequential access, aligned access

mat_mul

```
// for each row of C
for (int i=0; i<N; ++i) {
  // for each column of C
  for (int j=0; j<N; ++j) {
    // compute the inner product
    for (int k=0; k<N; ++k)
      C[i*N + j] += A[i*N + k]*B[k*N + j];
  }
}
```



Each thread is assigned a `Cij` to compute.

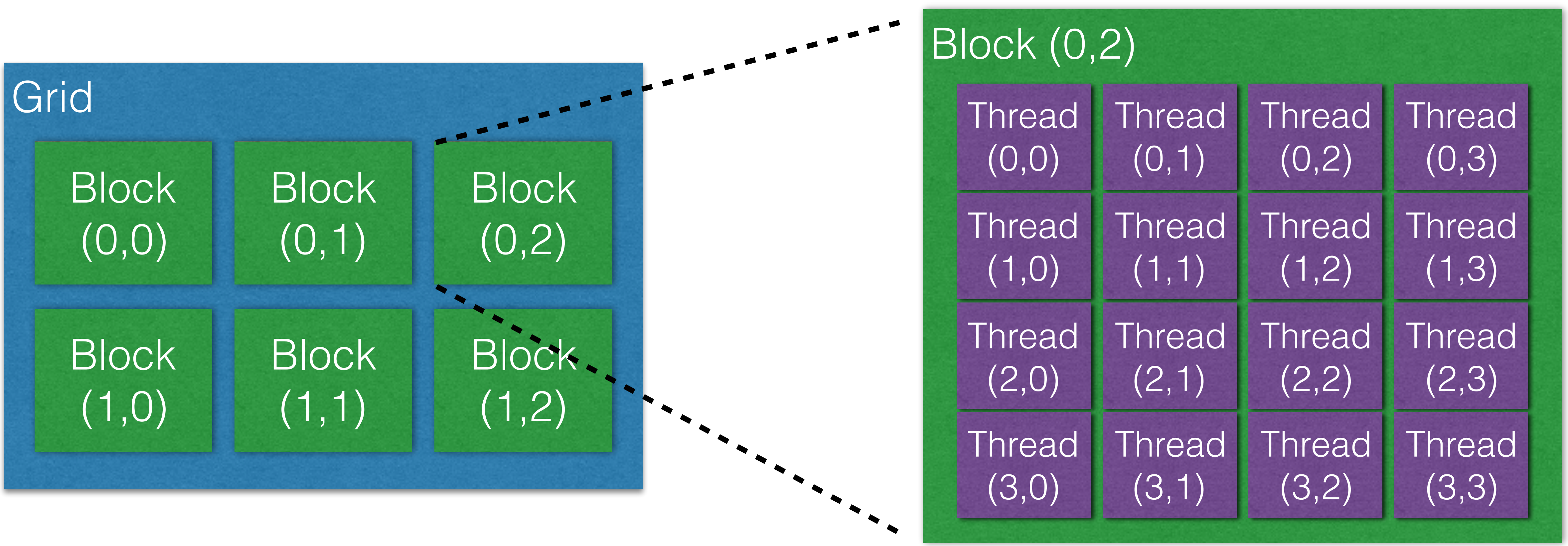
Take advantage of 2D thread / block arrangement notation!

Parallel `mat_mul`

- Each thread computes `Cij` — 2D organization of blocks and threads

```
size_t i = threadIdx.y + blockIdx.y*blockDim.y;
size_t j = threadIdx.x + blockIdx.x*blockDim.x;
```

- Thread at "global index" `(i,j)` computes `C[i*N+j]`



Creating 2D Grids

- Normal Kernel Call

```
mykernel<<<nblocks, thperblk>>>(…args…);
```

- 2D Grids — if each block contained `thpblk_x * thpblk_y` threads total

```
// specify block size
dim3 dim_block(thpblk_x, thpblk_y);
// specify number of blocks
dim3 dim_grid(nblocks_x, nblocks_y);
mykernel<<<dim_grid, dim_block>>>(…args…);
```

mat_mul — CUDA Kernel

```
__global__ void
mat_mul(double* C, double* A, double* B)
{
  // row#, col#, and row/col length
  size_t i = threadIdx.y + blockIdx.y*blockDim.y;
  size_t j = threadIdx.x + blockIdx.x*blockDim.x;
  size_t N = blockDim.x * gridDim.x;

  // inner product
  for (size_t k=0; k<N; ++k)
    C[i*N+j] = A[i*N+k] * B[k*N+j];
}
```