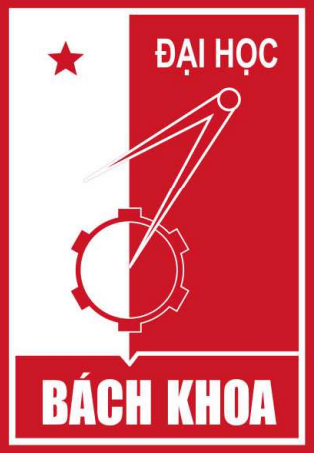


HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

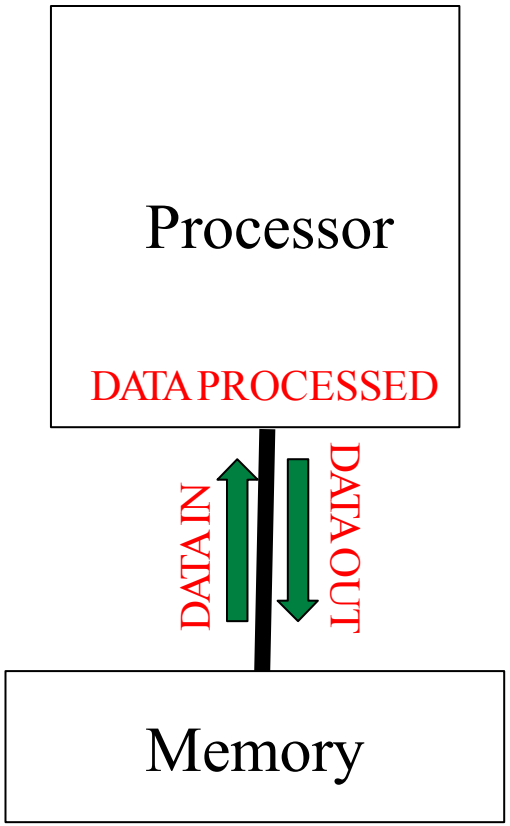
General purpose PGPU, CUDA

References

- Michael J. Quinn. **Parallel Computing. Theory and Practice**. McGraw-Hill
- Albert Y. Zomaya. **Parallel and Distributed Computing Handbook**. McGraw-Hill
- Ian Foster. **Designing and Building Parallel Programs**. Addison-Wesley.
- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar **. Introduction to Parallel Computing, Second Edition.** Addison Wesley.
- Joseph Jaja**. An Introduction to Parallel Algorithm.** Addison Wesley.
- Nguyễn Đức Nghĩa**. Tính toán song song.** Hà Nội 2003.

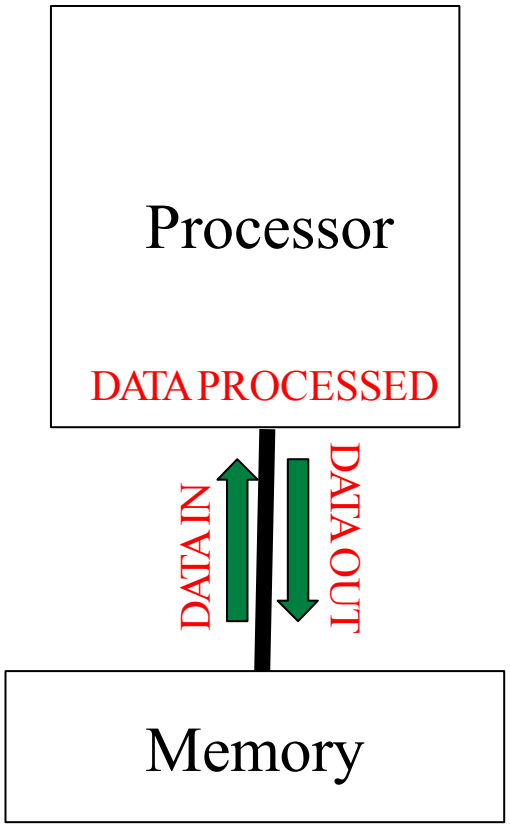
5.1 GPU Architecture

Performance factors (1)



- Amount of data processed at one time (Parallel processing)
- Processing speed on each data element (Clock frequency)
- Amount of data transferred at one time (Memory bandwidth)
- Time for each data element to be transferred (Memory latency)

Performance factors (2)



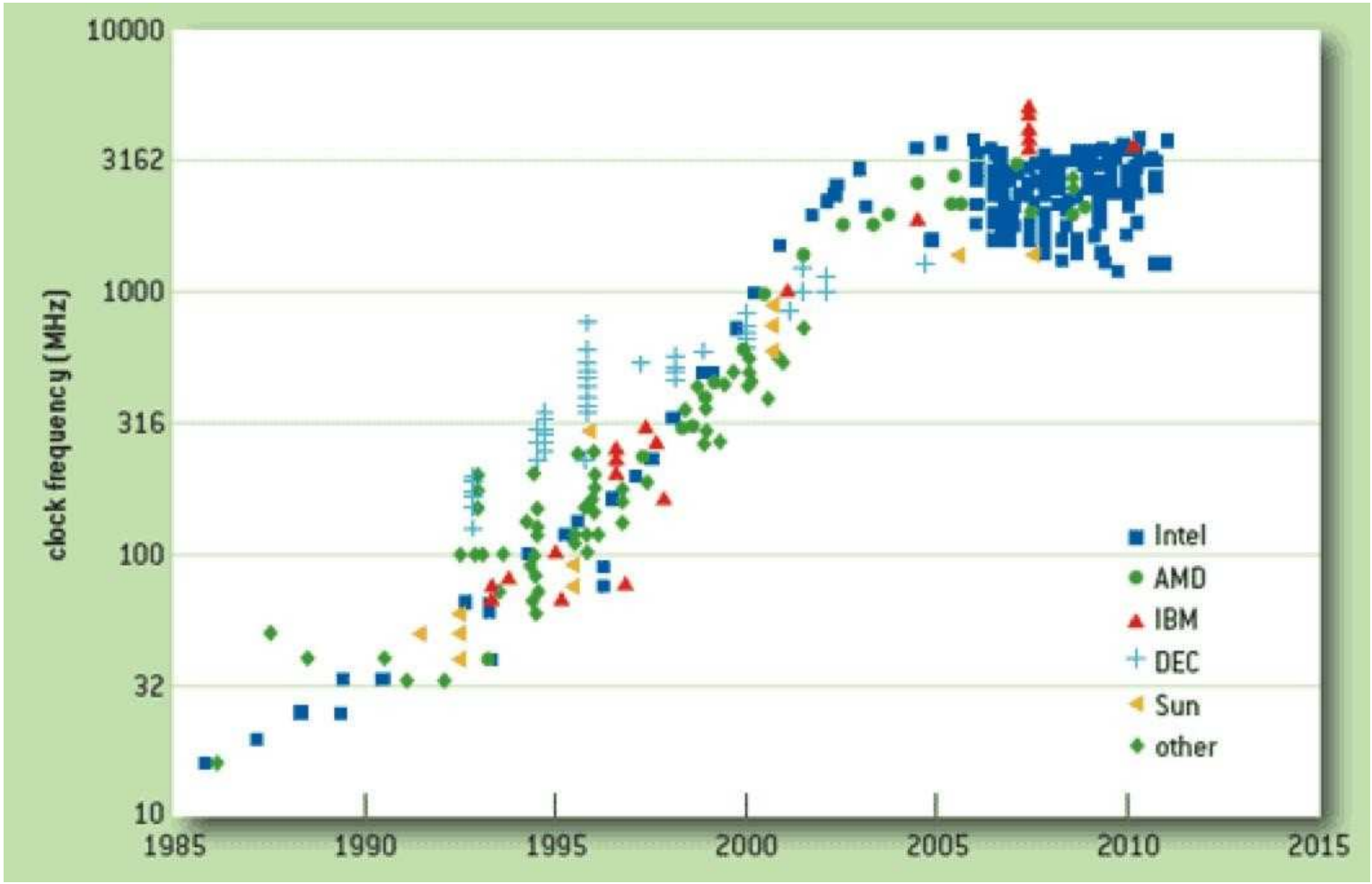
- Different computational problems are sensitive to these in different ways from one another
- Different architectures address these factors in different ways

CPUs: 4 key factors

- *Parallel processing*
  - Until relatively recently, each CPU only had a single core. Now CPUs have multiple cores, where each can process multiple instructions per cycle
- *Clock frequency*
  - CPUs aim to maximise clock frequency, but this has now hit a limit due to power restrictions (more later)
- *Memory bandwidth*
  - CPUs use regular DDR memory, which has limited bandwidth
- *Memory latency*
  - Latency from DDR is high, but CPUs strive to **hide** the latency through:
    - Large on-chip low-latency caches to stage data
    - Multithreading
    - Out-of-order execution

The Problem with CPUs

- The power used by a CPU core is proportional to Clock Frequency x Voltage^2
- In the past, computers got faster by increasing the frequency
  - Voltage was decreased to keep power reasonable.
- Now, voltage cannot be decreased any further
  - 1s and 0s in a system are represented by different voltages
  - Reducing overall voltage further would reduce this difference to a point where 0s and 1s cannot be properly distinguished



Reproduced from http://queue.acm.org/detail.cfm?id=2181798

The Problem with CPUs

- Instead, performance increases can be achieved through exploiting parallelism
- Need a chip which can perform many parallel operations every clock cycle
  - Many cores and/or many operations per core
- Want to keep power/core as low as possible
- Much of the power expended by CPU cores is on functionality not generally that useful for HPC
  - e.g. branch prediction

Accelerators

- So, for HPC, we want chips with simple, low power, number-crunching cores
- But we need our machine to do other things as well as the number crunching
  - Run an operating system, perform I/O, set up calculation etc
- Solution: “Hybrid” system containing both CPU and “accelerator” chips

Accelerators

- It costs a huge amount of money to design and fabricate new chips
  - Not feasible for relatively small HPC market
- Luckily, over the last few years, Graphics Processing Units (GPUs) have evolved for the highly lucrative gaming market
  - And largely possess the right characteristics for HPC
    - Many number-crunching cores
- GPUs now firmly established in HPC industry

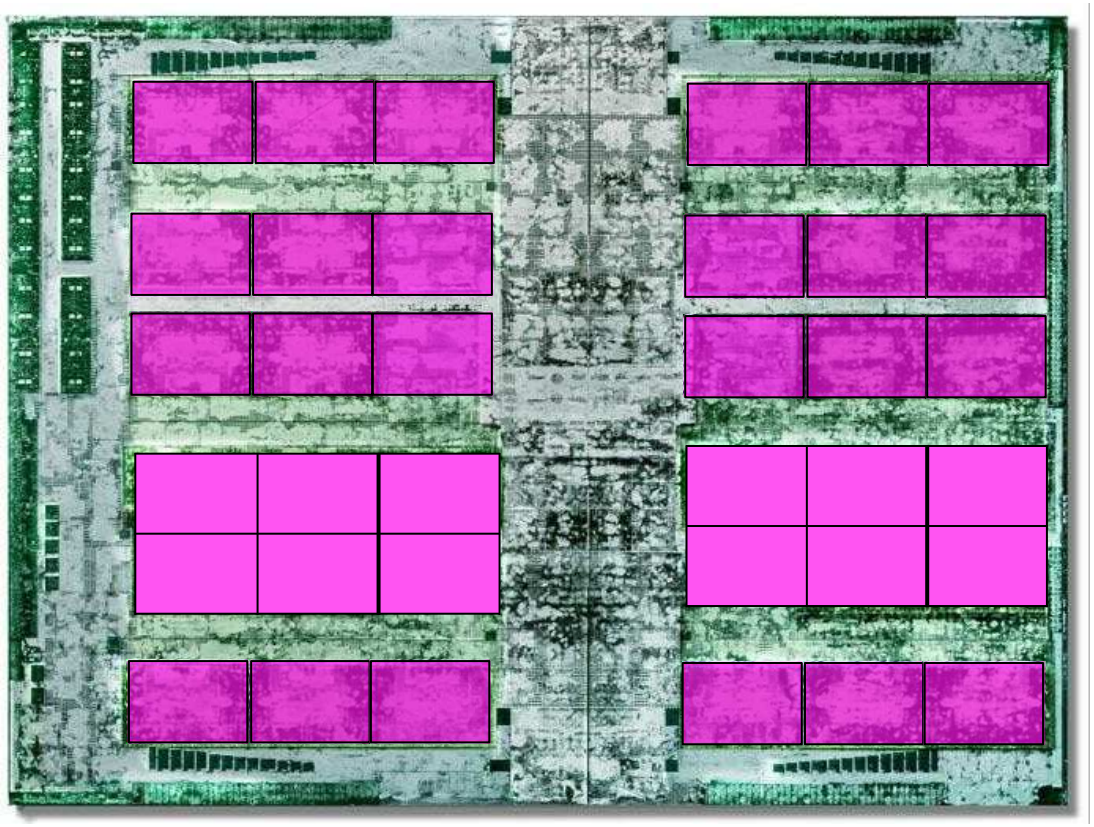
AMD 12-core CPU

- Not much space on CPU is dedicated to compute

= compute unit
(= core)

NVIDIA Pascal GPU

- GPU dedicates much more space to compute
  - At expense of caches, controllers, sophistication etc

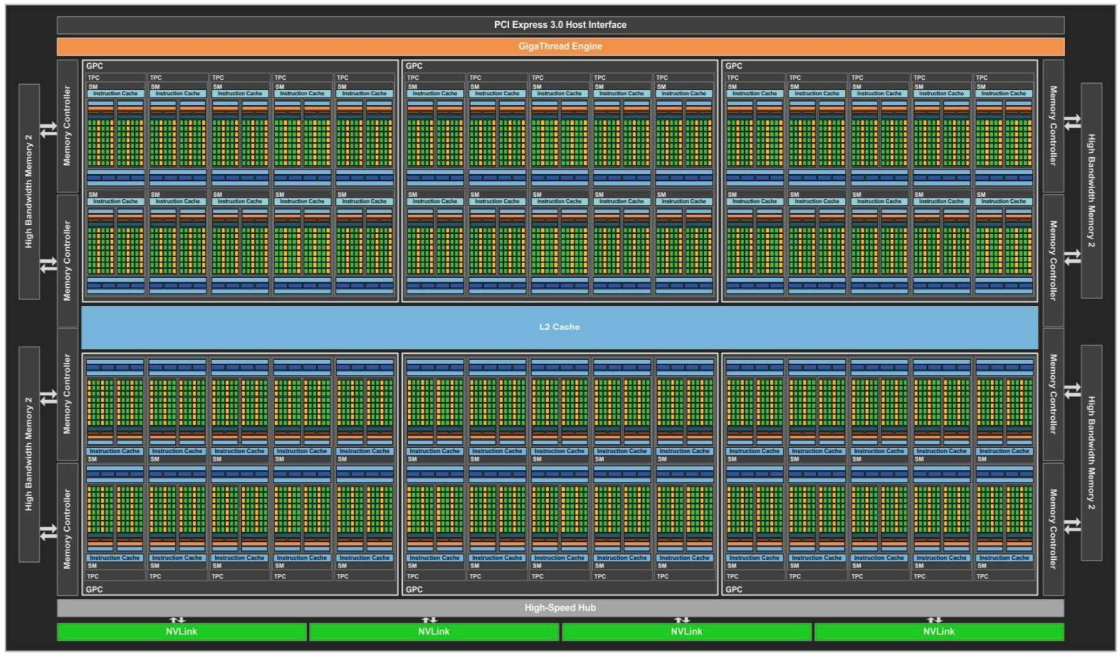


= compute unit
(= SM
= 64 CUDA cores)

GPUs: 4 key factors

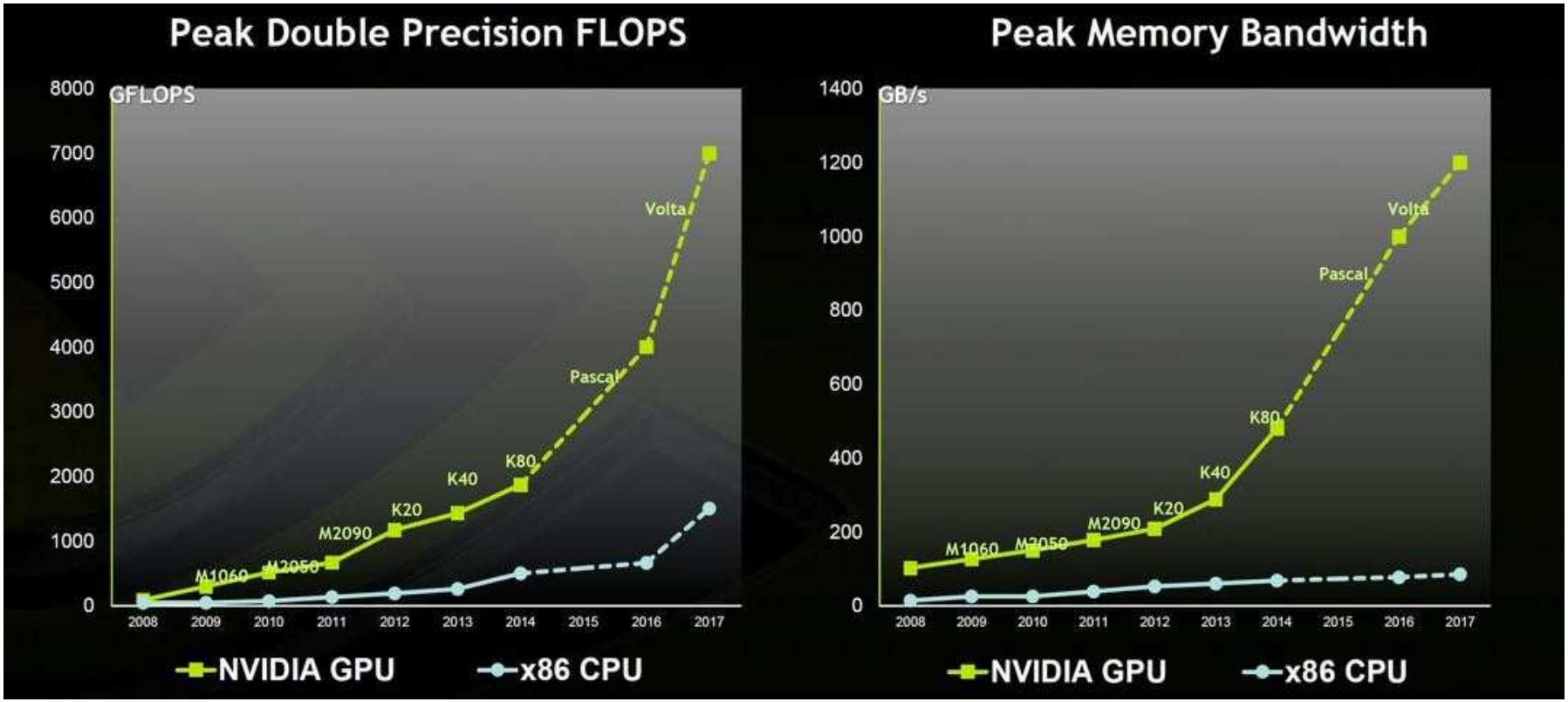
- *Parallel processing*
  - GPUs have a much higher extent of parallelism than CPUs: many more cores (high-end GPUs have thousands of cores).
- *Clock frequency*
  - GPUs typically have lower clock-frequency than CPUs, and instead get performance through parallelism.
- *Memory bandwidth*
  - GPUs use high bandwidth GDDR or HBM2 memory.
- *Memory latency*
  - Memory latency from is similar to DDR.
  - GPUs hide latency through very high levels of multithreading.

NVIDIA Tesla Series GPU



- Chip partitioned into *Streaming Multiprocessors* (SMs) that act independently of each other
- Multiple cores per SM. Groups of cores act in “lock-step”: they perform the same instruction on different data elements
- Number of SMs, and cores per SM, varies across products. High-end GPUs have thousands of cores

Performance trends



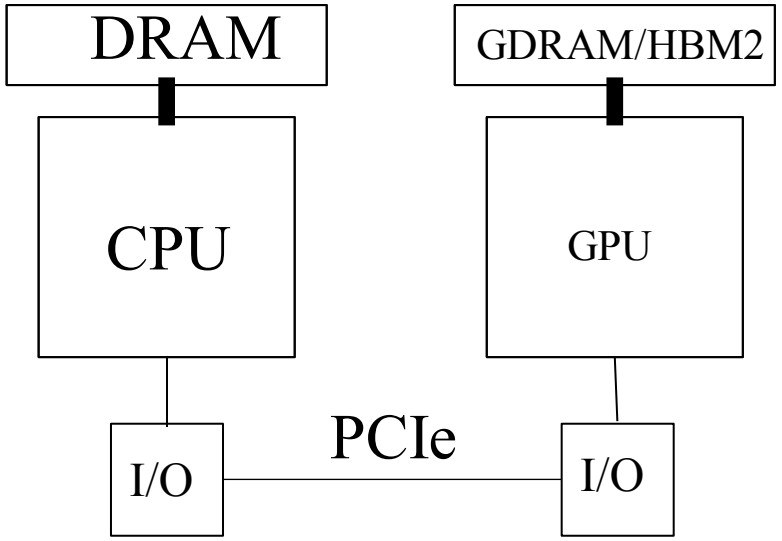
- GPU performance has been increasing much more rapidly than CPU

Programming GPU

- GPUs cannot be used *instead* of CPUs
  - They must be used together
  - GPUs act as accelerators
    - Responsible for the computationally expensive parts of the code
- CUDA: Extensions to the C language which allow interfacing to the hardware (NVIDIA specific)
- OpenCL: Similar to CUDA but cross-platform (including AMD and NVIDIA)

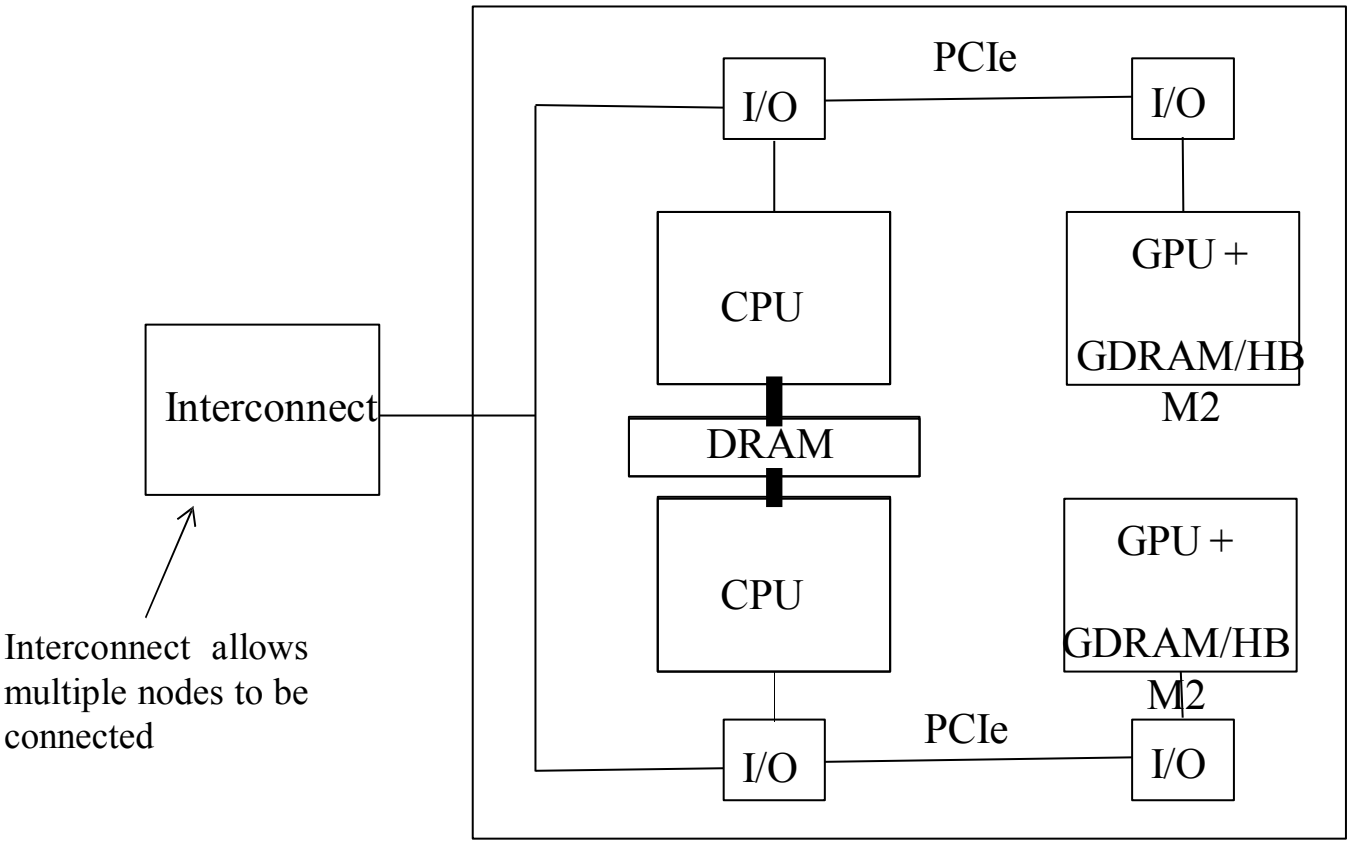
GPU Accelerated Systems

- CPUs and GPUs are used together
  - Communicate over PCIe bus
    - Or, in case of newest Pascal P100 GPUs, NVLINK (more later)

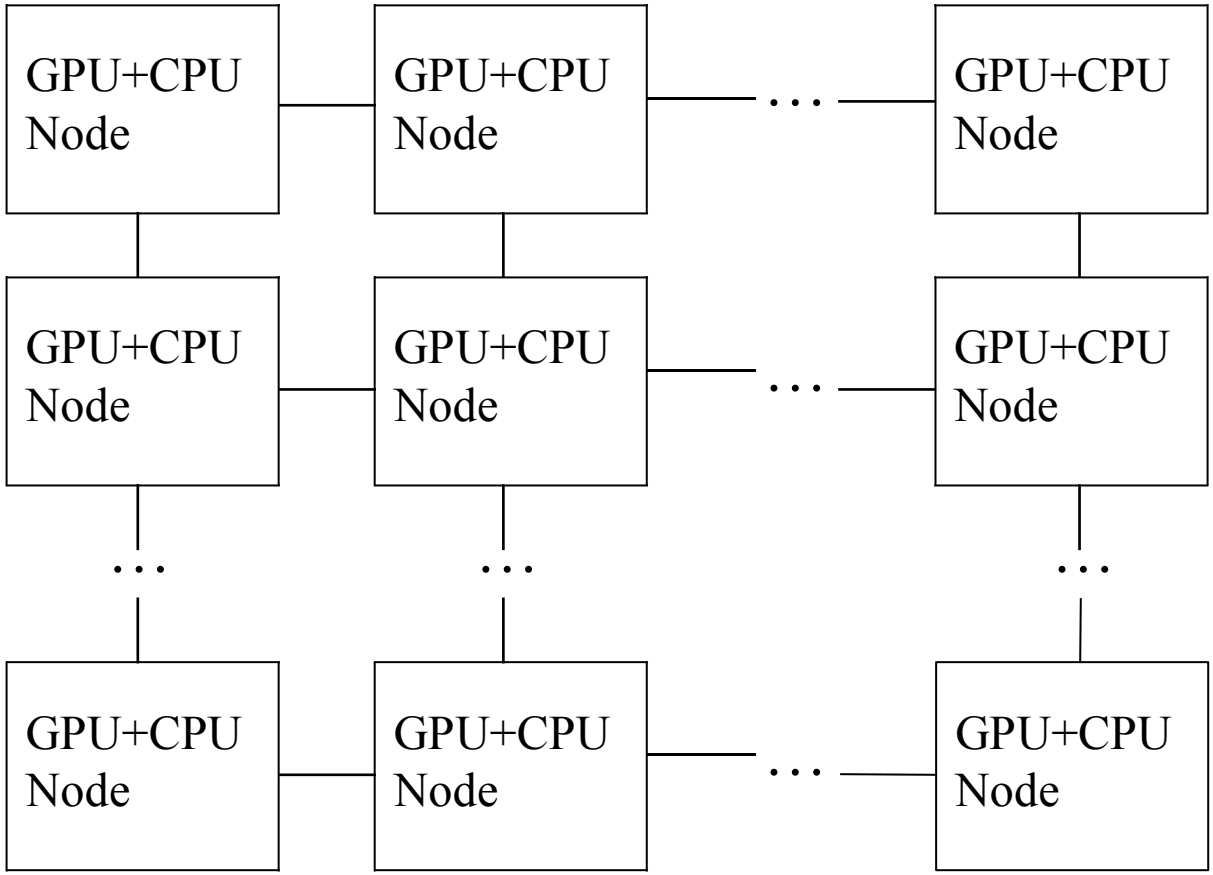


Scaling to larger systems

- Can have multiple CPUs and GPUs within each “workstation” or “shared memory node”
  - E.g. 2 CPUs +2 GPUs (below)
  - CPUs share memory, but GPUs do not



GPU Accelerated Supercomputer



Summary

- GPUs have higher compute and memory bandwidth capabilities than CPUs
  - Silicon dedicated to many simplistic cores
  - Use of high bandwidth graphics or HBM2 memory
- Accelerators are typically not used alone, but work in tandem with CPUs
- Most common are NVIDIA GPUs
  - AMD also have high performance GPUs, but not so widely used due to programming support
- GPU accelerated systems scale from simple workstations to large-scale supercomputers

5.2 Introduction to CUDA Programming

What is CUDA?

- Programing system for machines with GPUs
  - Programming Language
  - Compilers
  - Runtime Environments
  - Drivers
  - Hardware

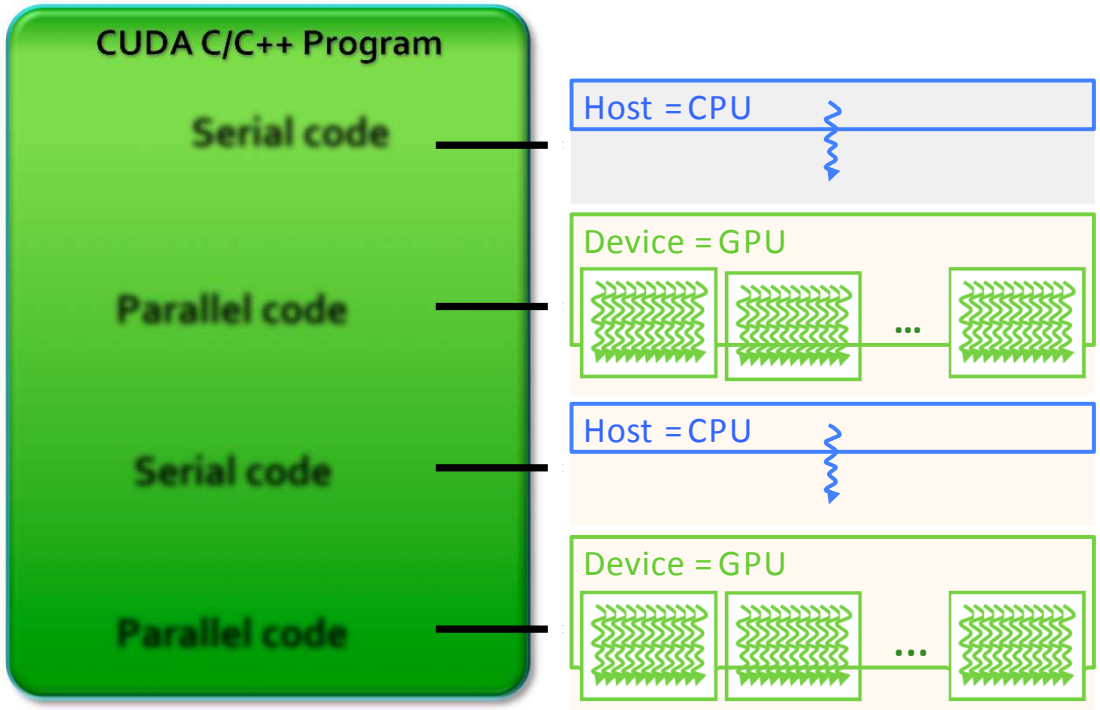
CUDA : Heterogeneous Parallel Computing

- **CPU optimized for fast single-thread execution**
  - Cores designed to execute 1 thread or 2 threads concurrently
  - Large caches attempt to hide DRAM access times
  - Cores optimized for low-latency cache accesses
  - Complex control-logic for speculation and out-of-orderexecution

- **GPU optimized for high multi-thread throughput**
  - Cores designed to execute many parallel threads concurrently
  - Cores optimized for data-parallel, throughput computation
  - Chips use extensive multithreading to tolerate DRAM access times

CUDA C/C++ Program

- **Serial code executes in a Host (CPU) thread**
- **Parallel code executes in many concurrent Device (GPU) threads across multiple parallel processing elements**

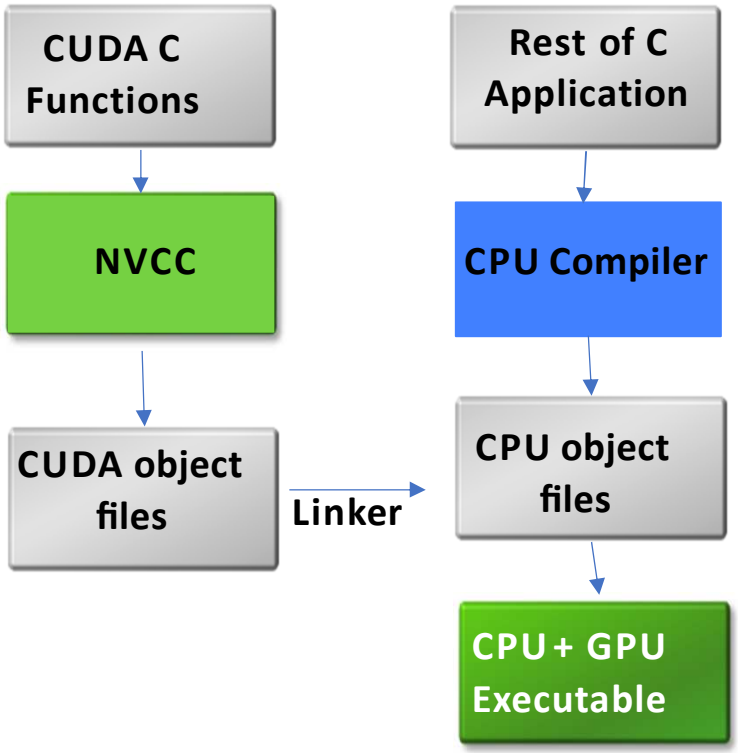


Compiling CUDA C/C++ Programs

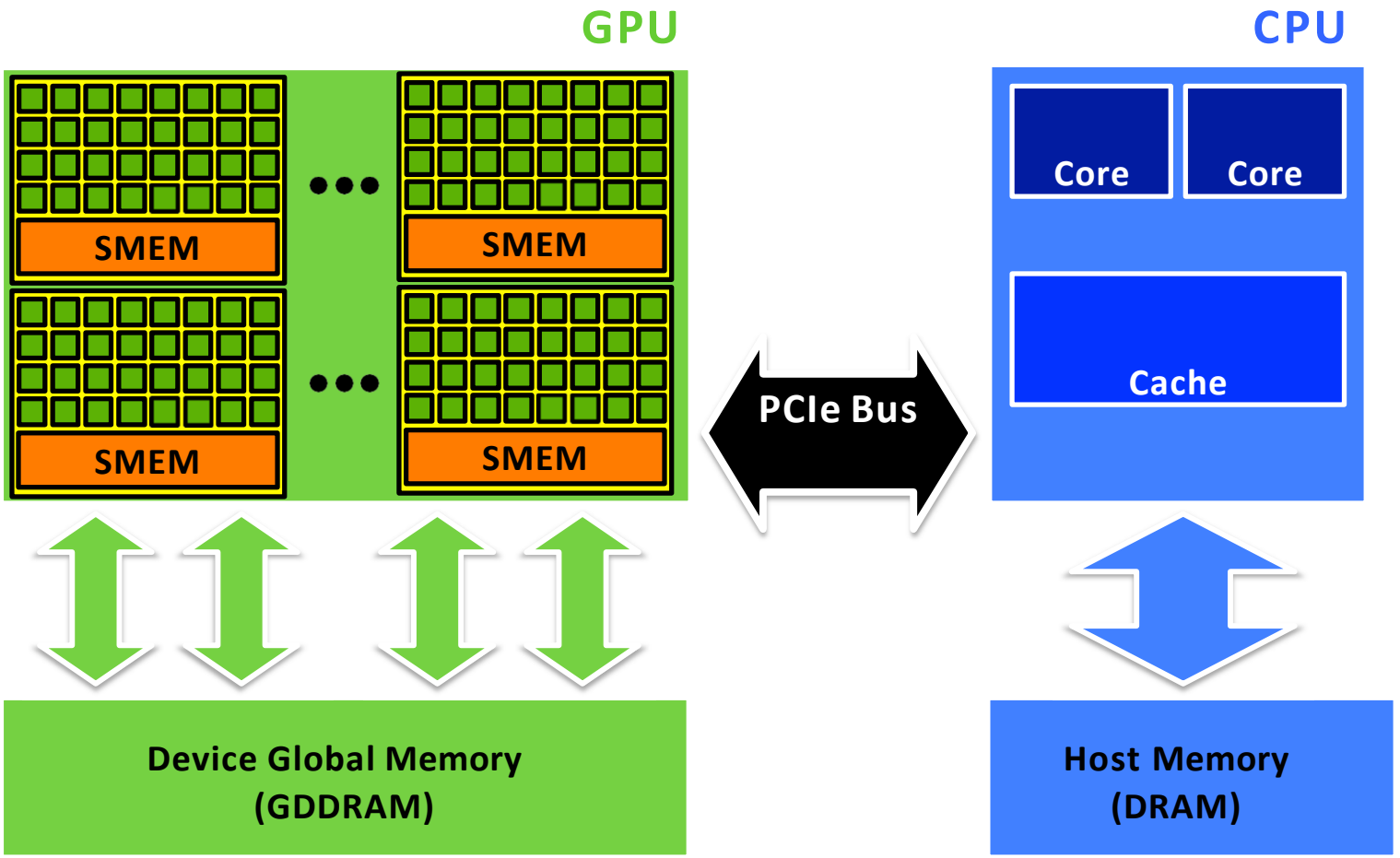
```
// foo.cpp
int foo(int x)
{
  ...
}
float bar(float x)
{
  ...
}
```

```
// saxpy.cu
__global__void saxpy(int n, float ... )
{
  int i =threadIdx.x +... ;
  if (i <n) y[i] =a*x[i] + y[i];
}
```

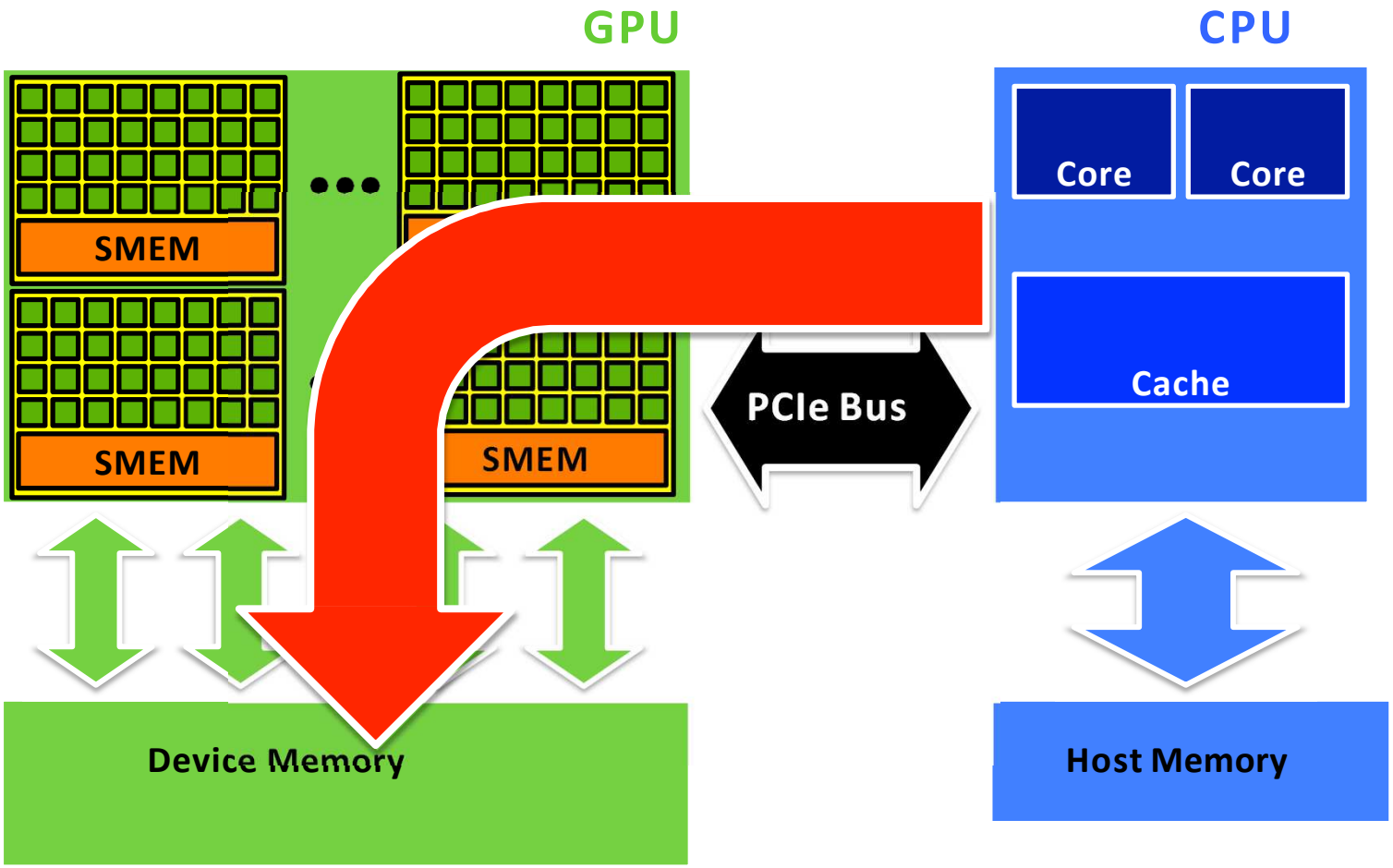
```
// main.cpp
void main( ) {
  float x=bar(1.0)
  if (x<2.0f)
    saxpy<<<...>>>(foo(1), ...);
  ...
}
```



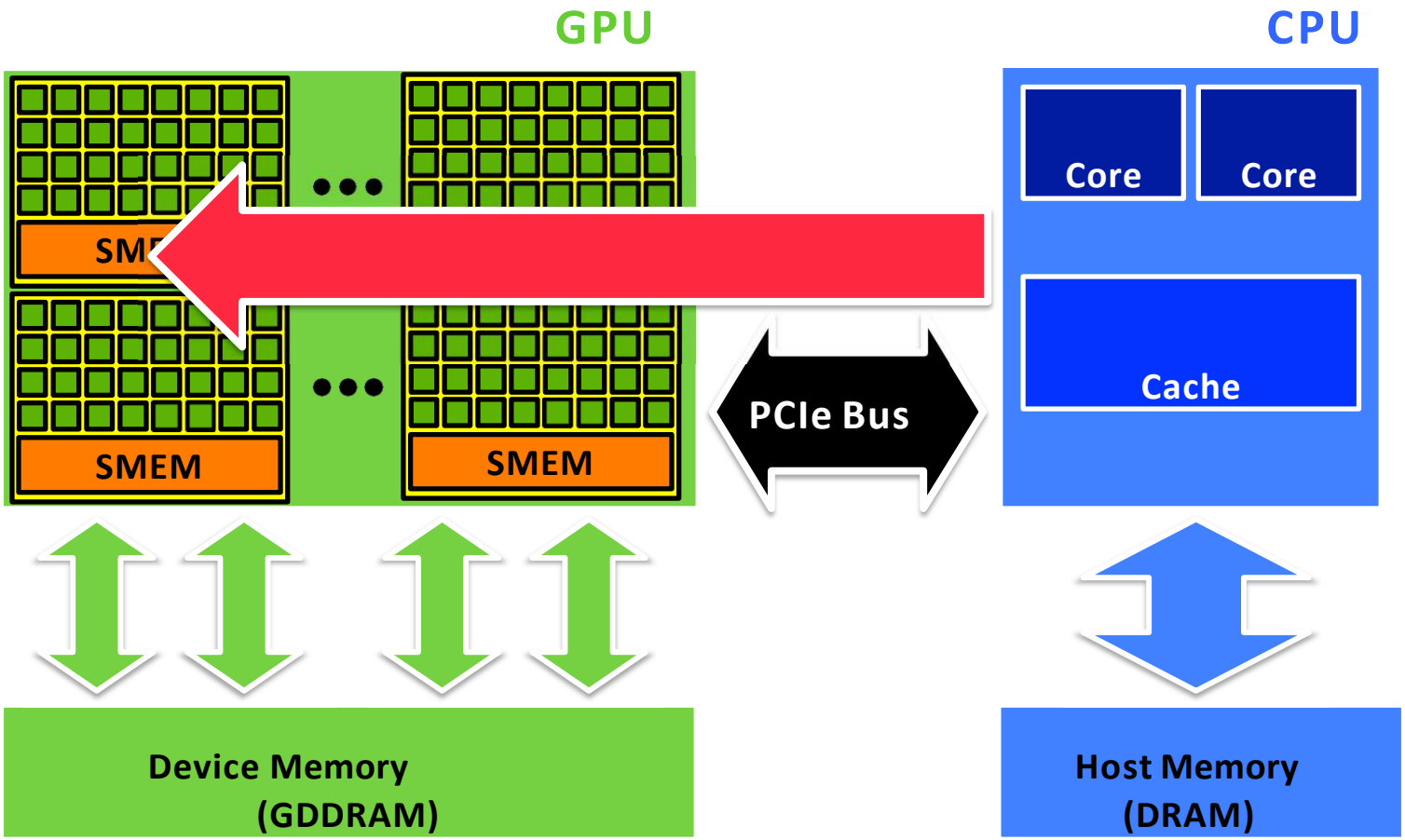
Canonical execution flow



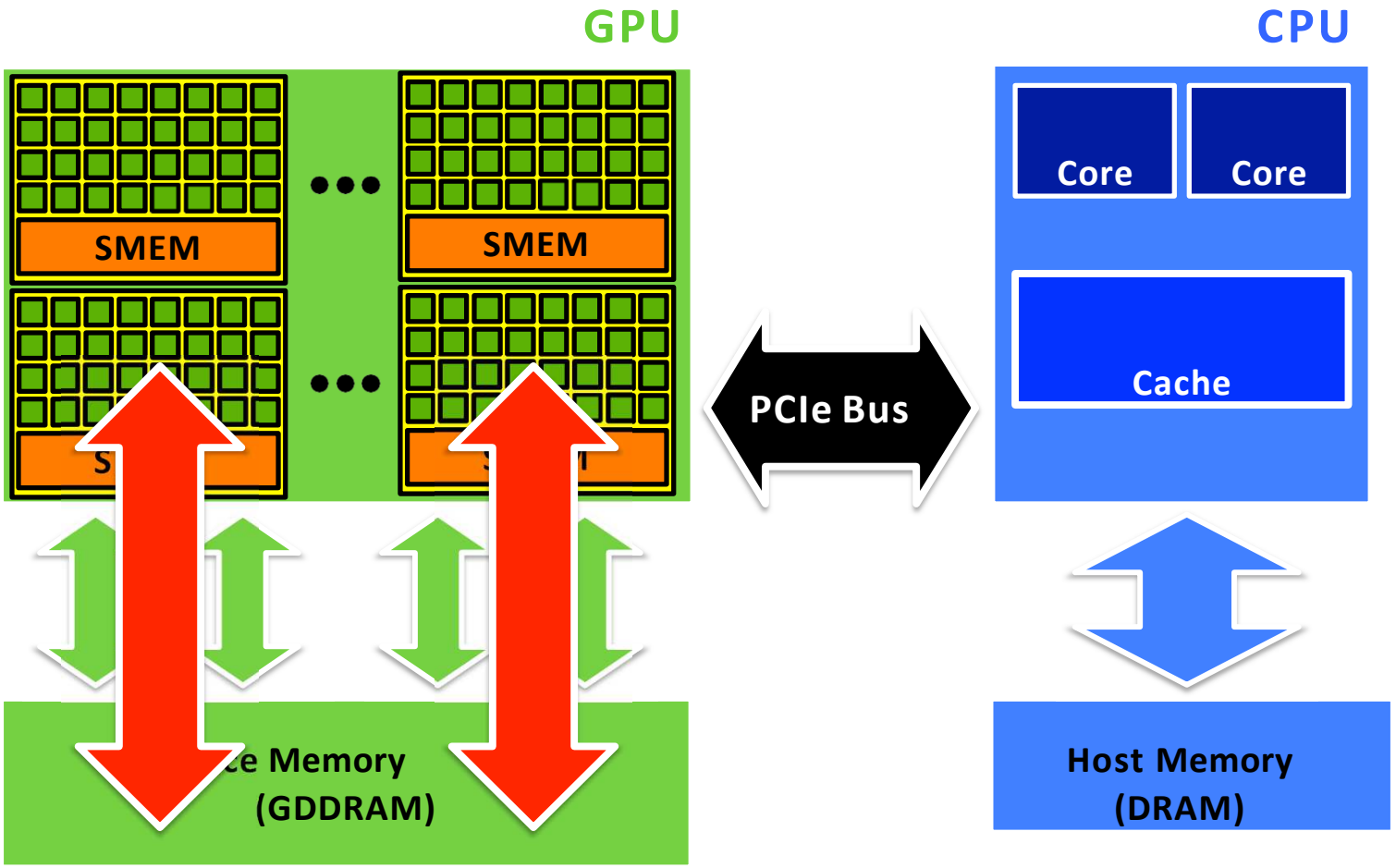
Step 1 – copy data to GPU memory



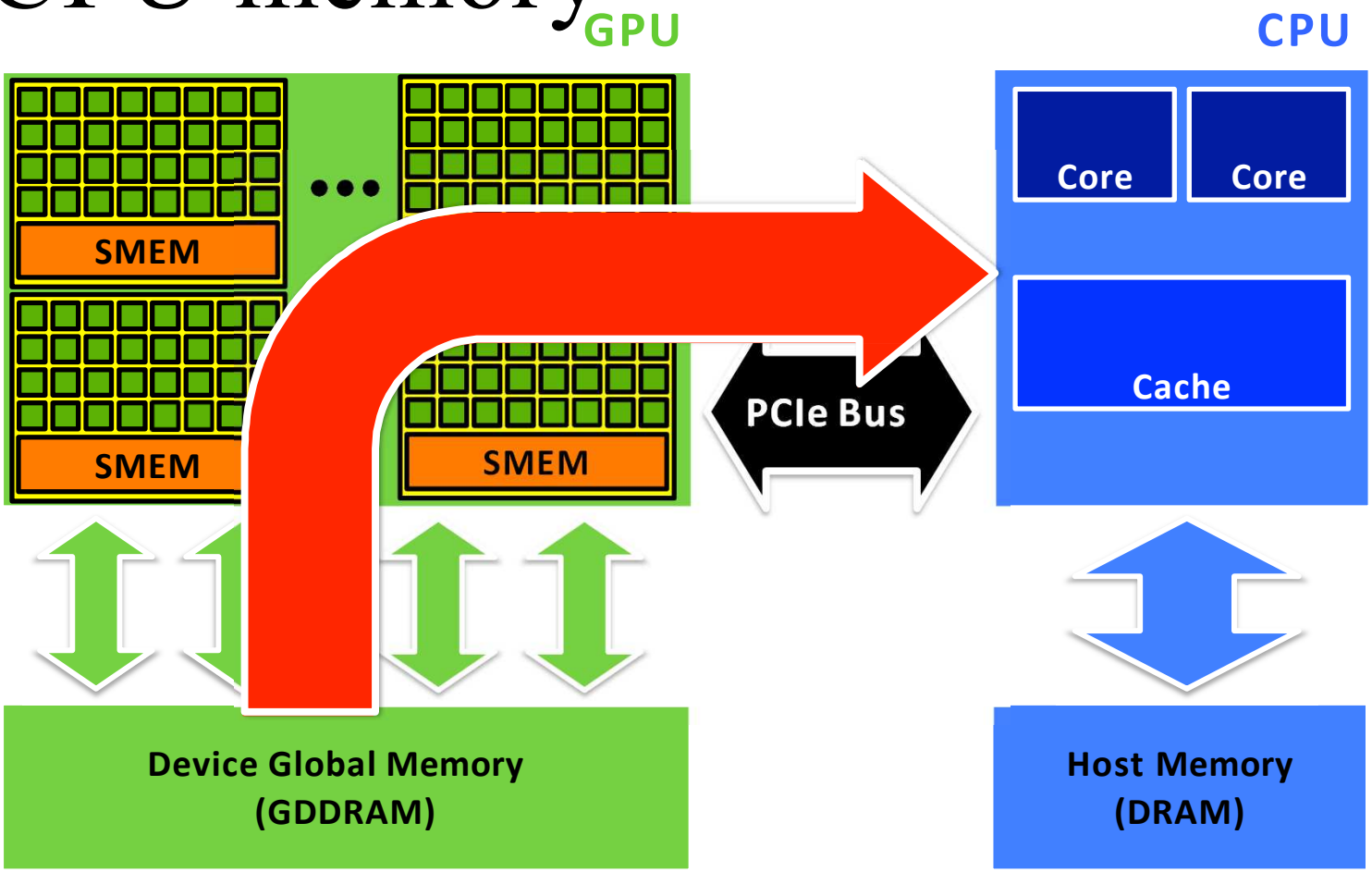
Step 2 – launch kernel on GPU



Step 3 – execute kernel on GPU

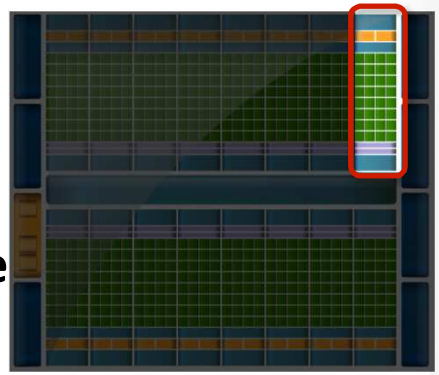


Step 4 – copy data to CPU memory



Example: Fermi's GPU Architecture

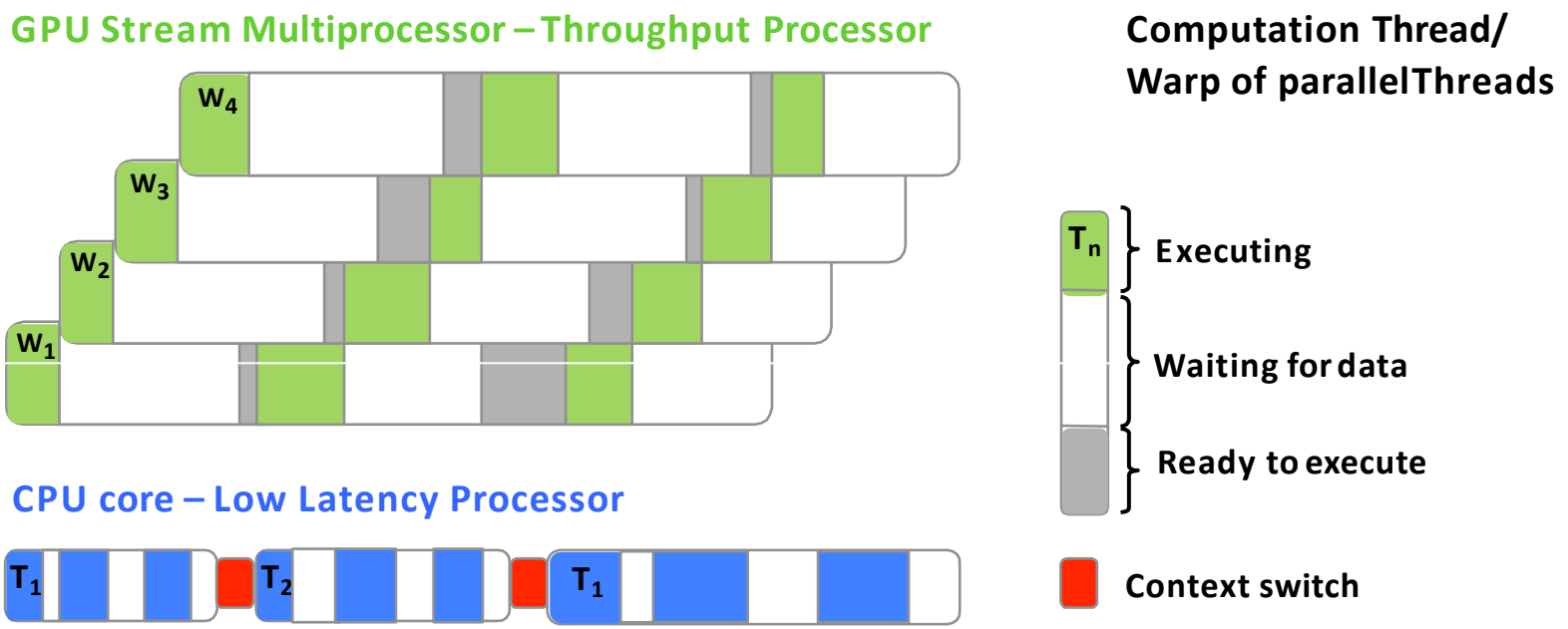
- 32 CUDA Cores per Streaming Multiprocessor (SM)
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 Warp schedulers per SM
  - 1,536 concurrent threads
- 4 special--function units
- 64KB shared memory + L1 cache
- 32K 32-bit registers



- Fermi GPUs have as many as 16 SMs
  - 24,576 concurrent threads

Multithreading

- CPU architecture attempts to minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps



CUDA Kernels

- **Parallel portion of application: execute as a kernel**
  - **Entire GPU executes kernel**
  - **Kernel launches create thousands of CUDA threads efficiently**

CPU	Host	Executes functions
GPU	Device	Executes kernels

- **CUDA threads**
  - **Lightweight**
  - **Fast switching**
  - **1000s execute simultaneously**
- **Kernel launches create hierarchical groups of threads**
  - **Threads are grouped into Blocks, and Blocks into Grids**
  - **Threads and Blocks represent different levels of parallelism**

CUDA C : C with a few keywords

- **Kernel : function that executes on device (GPU) and can be called from host (CPU)**
  - **Can only access GPU memory**
  - **No variable number of arguments**
  - **No static variables**

- **Functions must be declared with a qualifier**

  __global__ : GPU kernel function launched by CPU, must return void

  __device__ : can be called from GPU functions

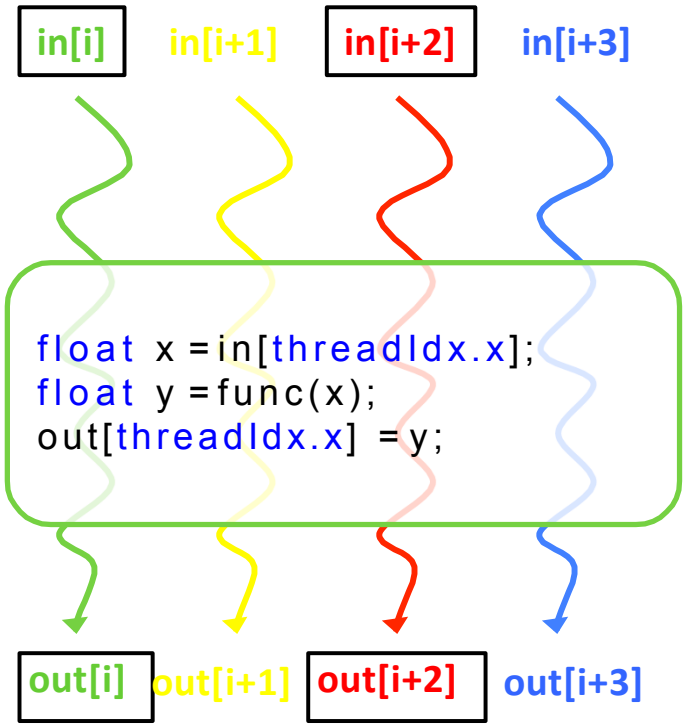
  __host__ : can be called from CPU functions (default)

  __host__ and __device__ qualifiers can be combined

- **Qualifiers determines how functions are compiled**
  - **Controls which compilers are used to compile functions**

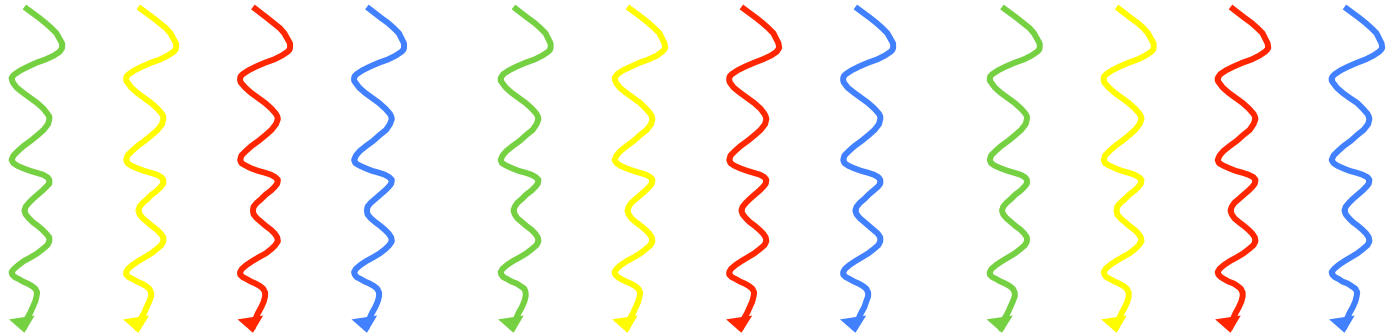
CUDA Kernels : Parallel Threads

- A kernel is a function executed on the GPU as an array of parallel threads
- All threads execute the same kernel code, but can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions



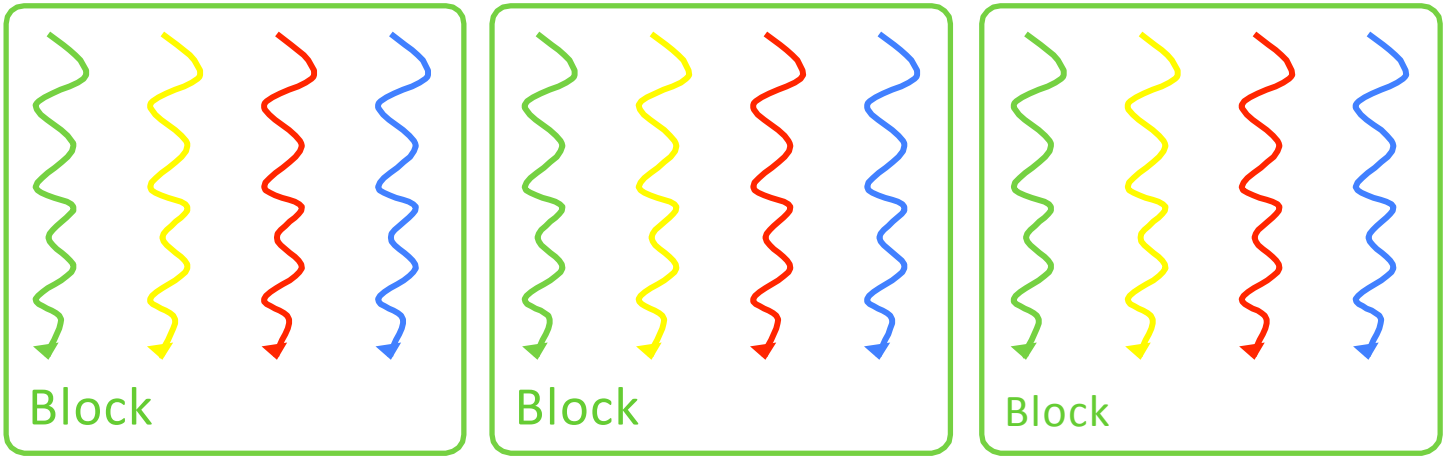
5.3 Synchronization/Communication

CUDA Thread Organization



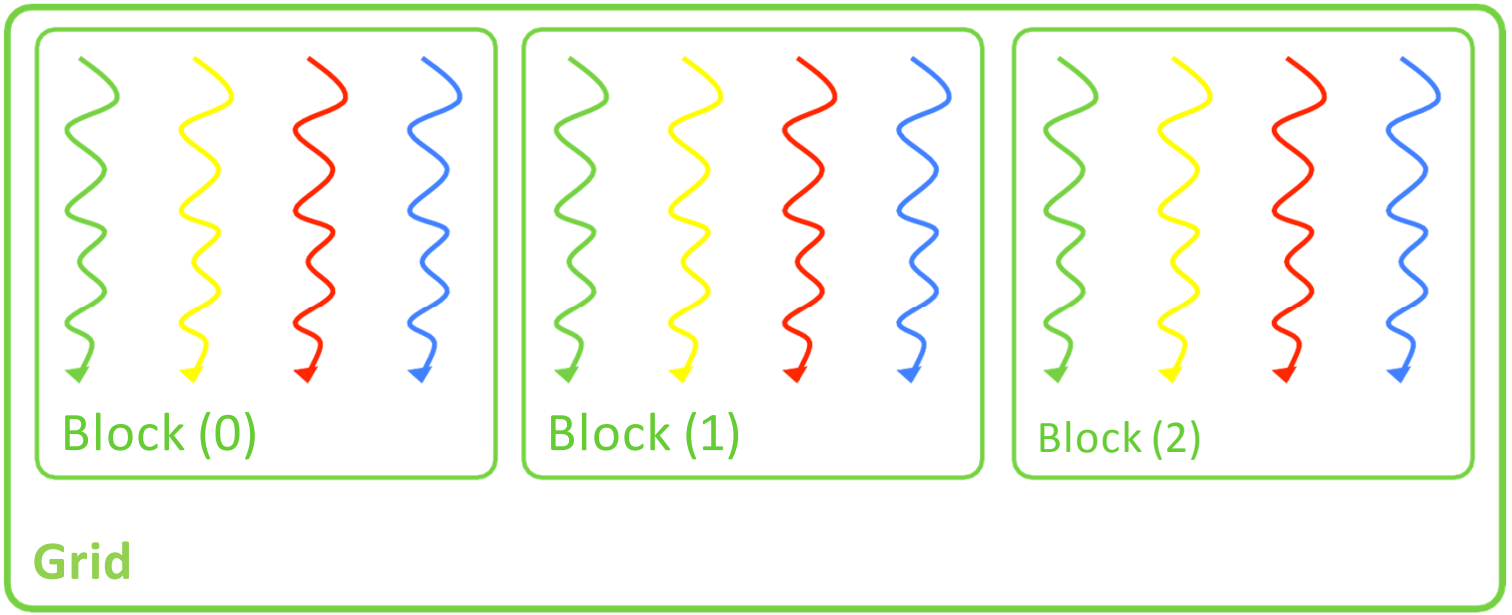
- GPUs can handle thousands of concurrent threads
- CUDA programming model supports even more
  - Allows a kernel launch to specify more threads than the GPU can execute concurrently
  - Helps to amortize kernel launch times

Blocks of threads



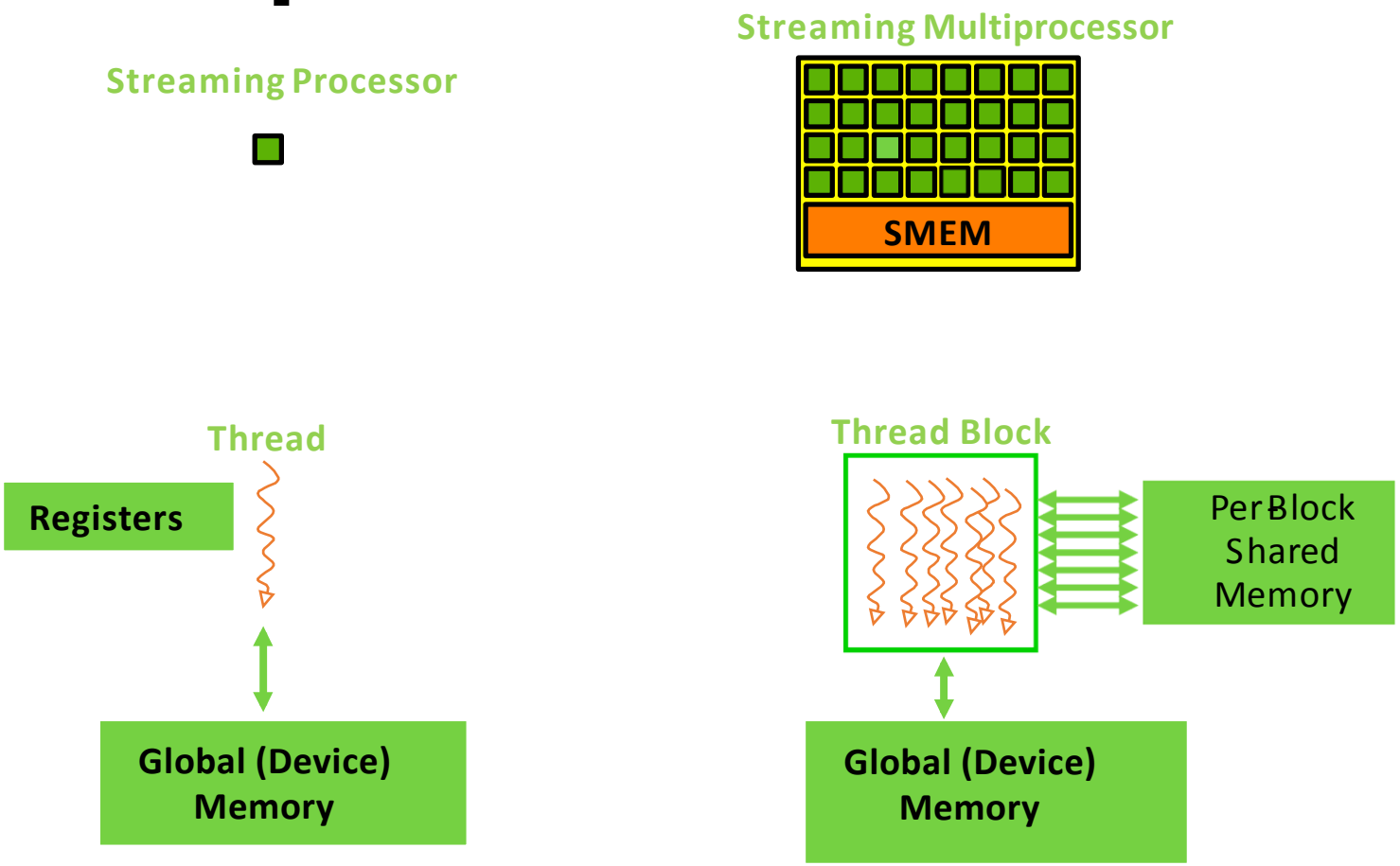
- **Threads are grouped into blocks**

Grids of blocks

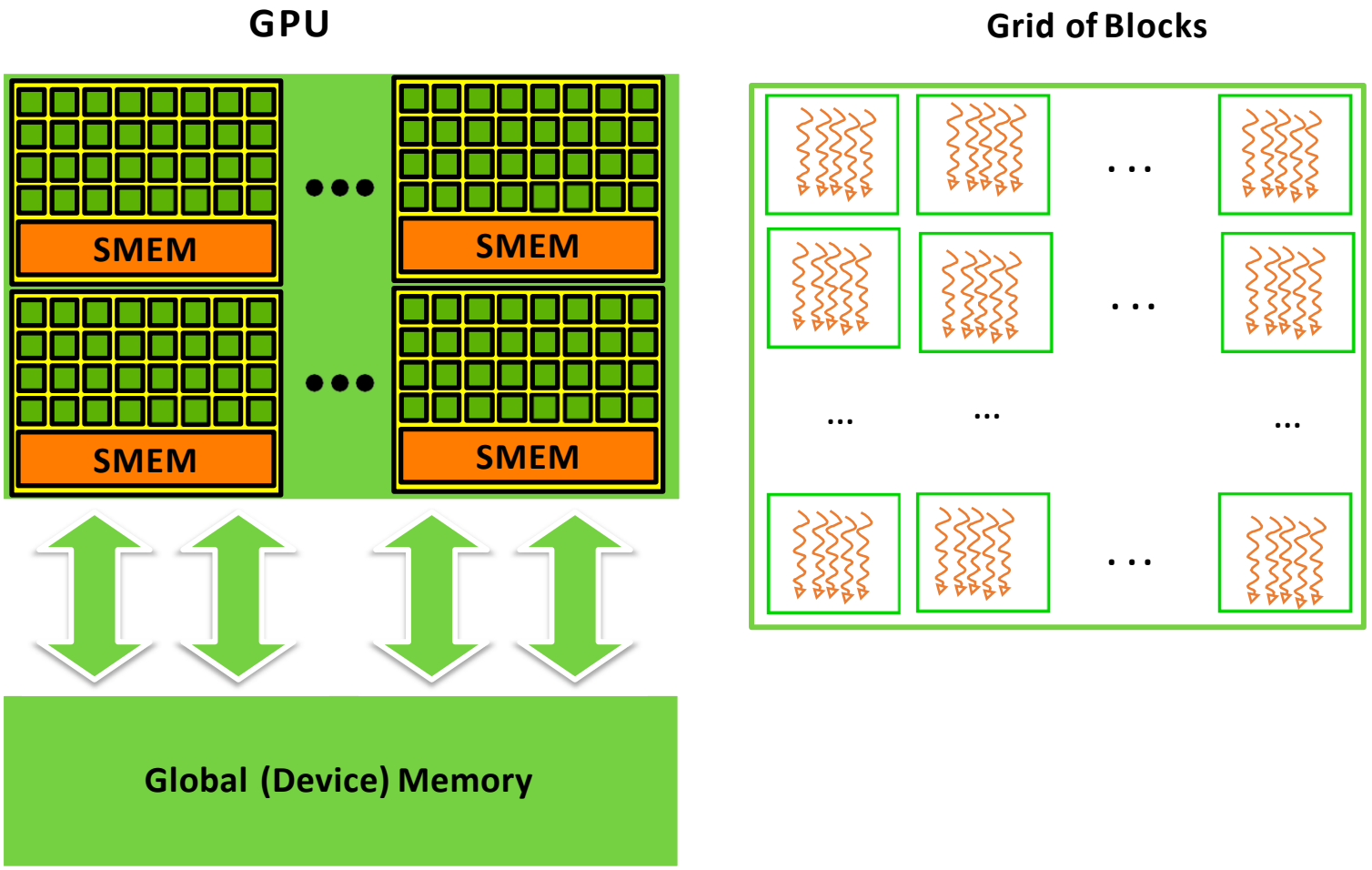


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Blocks execute on Streaming Multiprocessors



Grids of blocks executes across GPU



Kernel Execution Recap

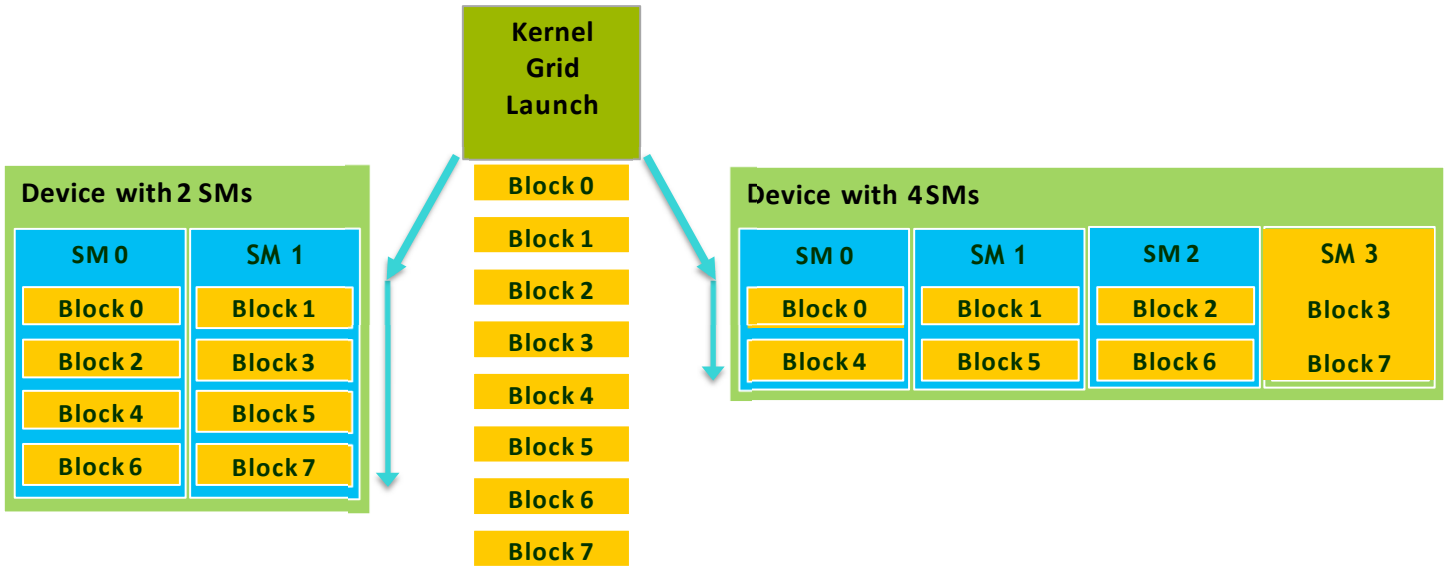
- A thread executes on a single streaming processor
  - Allows use of familiar scalar code within kernel

- A block executes on a single streaming multiprocessor
  - Threads and blocks do not migrate to different SMs
  - All threads within block execute in concurrently, in parallel
    - A streaming multiprocessor may execute multiple blocks
      - Must be able to satisfy aggregate register and memory demands

- A grid executes on a single device (GPU)
  - Blocks from the same grid may execute concurrently or serially
  - Blocks from multiple grids may execute concurrently
  - A device can execute multiple kernels concurrently

Block abstraction provides scalability

- Blocks may execute in arbitrary order, concurrently or sequentially, and parallelism increases with resources
  - Depends on when execution resources become available
- Independent execution of blocks provides scalability
  - Blocks can be distributed across any number of SMs



Blocks enable efficient collaboration

- **Threads often need to collaborate**
  - Cooperatively load/store common data sets
  - Share results or cooperate to produce a single result
  - Synchronize with each other

- **Threads in the same block**
  - Can communicate through shared and global memory
  - Can synchronize using fast synchronization hardware

- **Threads in different blocks of the same grid**
  - Cannot synchronize reliably

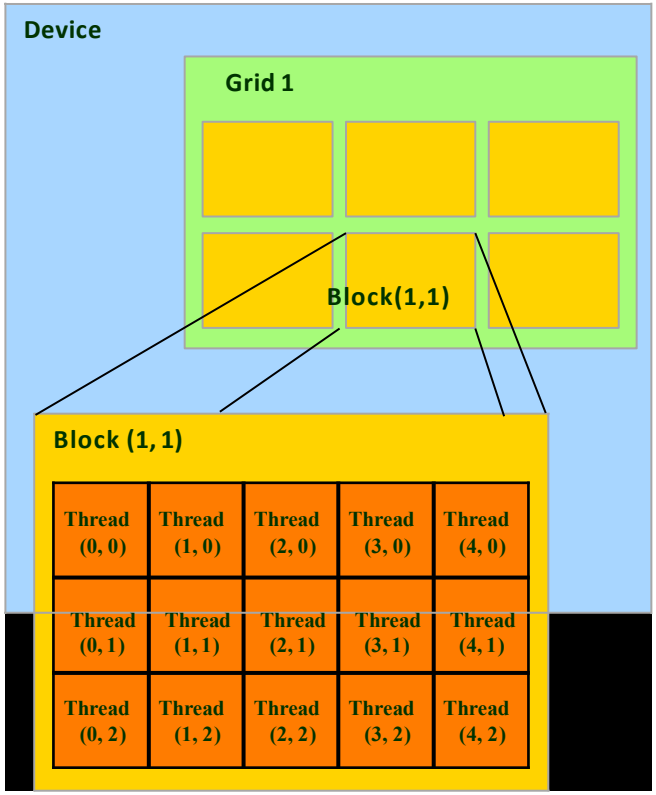
  No guarantee that both threads are alive at the same time

Blocks must be independent

- **Any possible interleaving of blocks is allowed**
  - **Blocks presumed to run to completion without pre-emption**
  - **May run in any order, concurrently or sequentially**
- **Programs that depend on block execution order within grid for correctness are not well formed**
  - **May deadlock or return incorrect results**
- **Blocks may coordinate but not synchronize**
  - **shared queue pointer: OK**
  - **shared lock: BAD … can easily deadlock**

Thread and Block ID and Dimensions

- **Threads**: 3D IDs, unique within a block
- **Thread Blocks**: 2D IDs, unique within a grid
- **Dimensions set at launch**: Can be unique for each grid
- **Build-in variables**
  - threadIdx, blockIdx
  - blockDim, gridDim
- **Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads**



Examples of Indexes and Indexing

```
__global__void kernel( int *a )
{
    int idx =blockIdx.x*blockDim.x +threadIdx.x;
    a[idx] = 7;
    }
```

7777777777777777

```
__global__void kernel( int *a )
{
    int idx =blockIdx.x*blockDim.x +threadIdx.x;
    a[idx] = blockIdx.x;
    }
```

0000111122223333

```
__global__void kernel( int *a )
{
    int idx =blockIdx.x*blockDim.x +threadIdx.x;
    a[idx] = threadIdx.x;
    }
```

0123012301230123

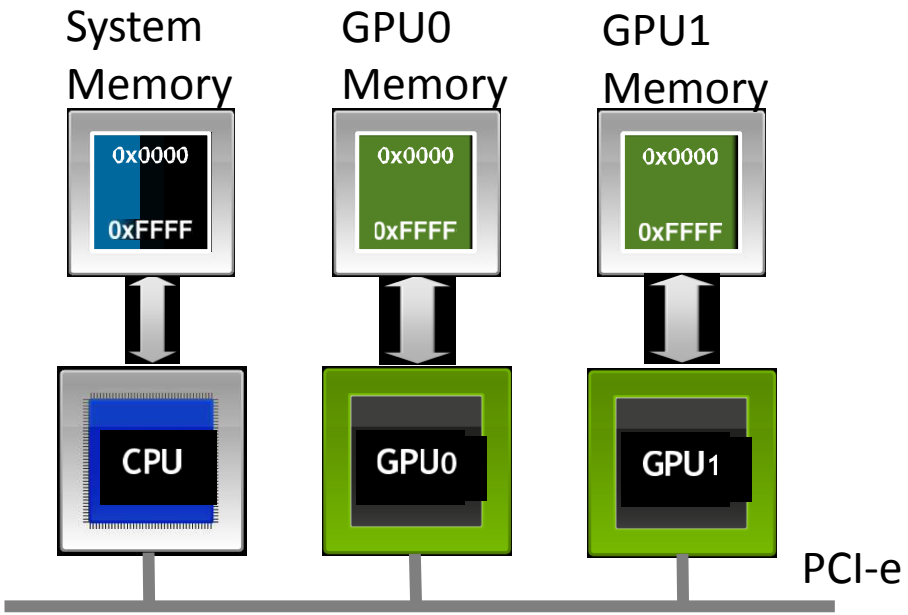
Example of 2D indexing

```
__global__ void kernel(int *a, int dimx, int dimy)
{
  int ix  = blockIdx.x*blockDim.x + threadIdx.x;
  int iy  = blockIdx.y*blockDim.y + threadIdx.y;
  int idx = iy * dimx + ix;

  a[idx]  = a[idx]+1;
}
```

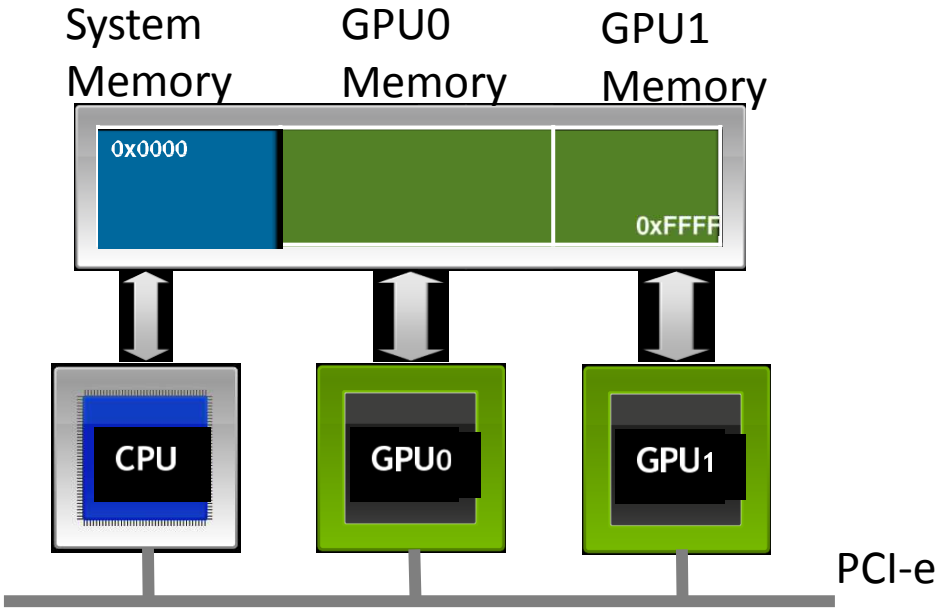
Independent address spaces

- **CPU and GPU have independent memory systems**
  - **PCIe bus transfers data between CPU and GPU memory systems**
- **Typically, CPU thread and GPU threads access what are logically different, independent virtual address spaces**



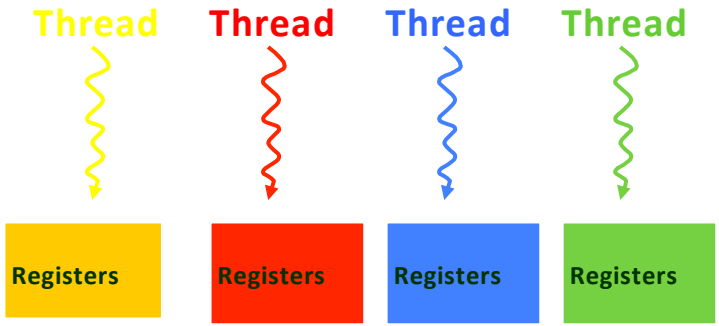
Independent address spaces: consequences

- **Cannot reliably determine whether a pointer references a host (CPU) or device (GPU) address from the pointer value**
  - **Dereferencing CPU/GPU pointer on GPU/CPU will likely cause crash**
- **Unified virtual addressing (UVA) in CUDA 4.0**
  - **One virtual address space shared by CPU thread and GPU threads**



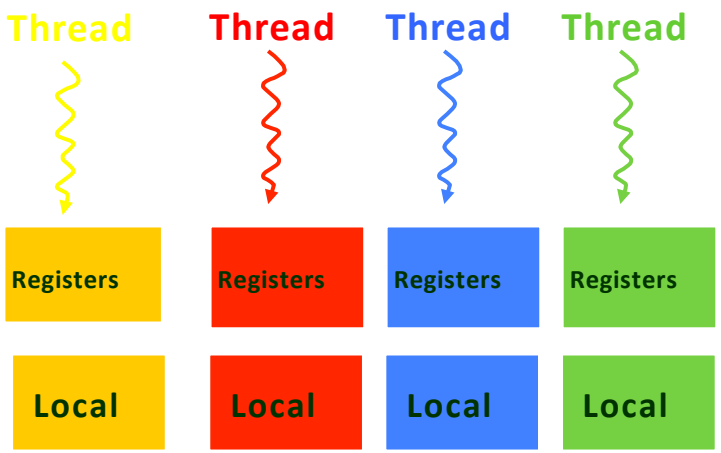
CUDA Memory Hierarchy

- **Thread**
  - **Registers**



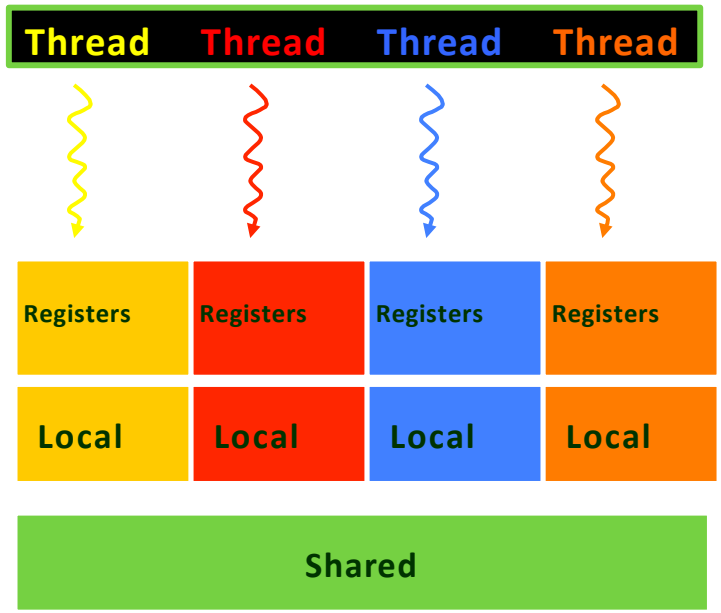
CUDA Memory Hierarchy

- **Thread**
  - **Registers**
  - **Local memory**



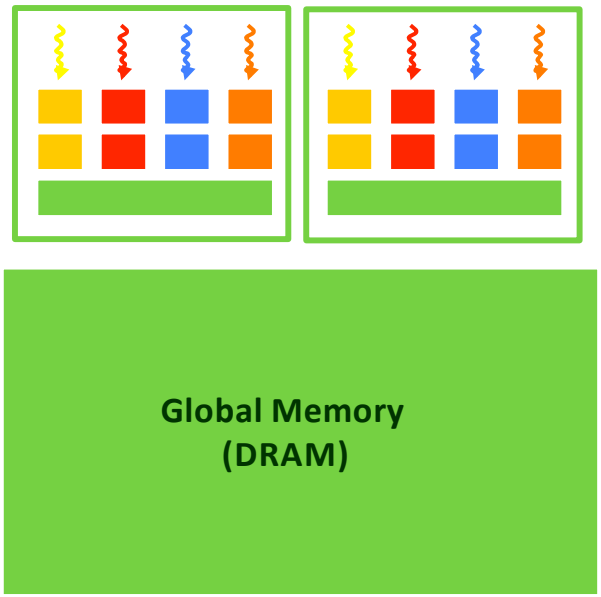
CUDA Memory Hierarchy

- **Thread**
  - Registers
  - Local memory
- **Thread Block**
  - Shared memory



CUDA Memory Hierarchy

- **Thread**
  - **Registers**
  - **Local memory**
- **Thread Block**
  - **Shared memory**
- **All Thread Blocks**
  - **Global Memory**



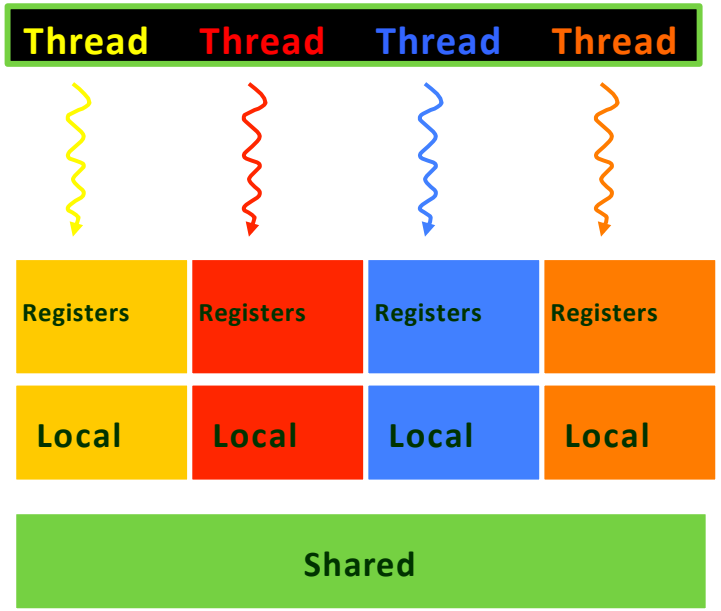
Shared Memory

__shared__ <type> x [<elements>];

- Allocated per thread block
- Scope: threads in block
- Data lifetime: same as block
- Capacity: small (about 48kB)
- Latency: a few cycles
- Bandwidth: very high

  SM: 32 * 4 B * 1.15 GHz /2 = 73.6 GB/s

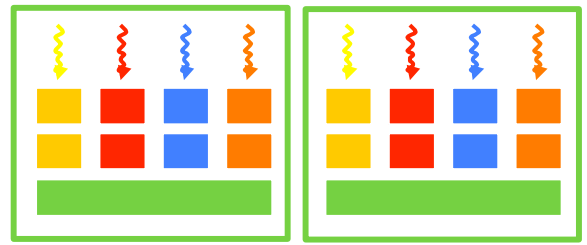
  GPU: 14 * 32 * 4 B * 1.15 GHz /2 = 1.03 TB/s
- Common uses



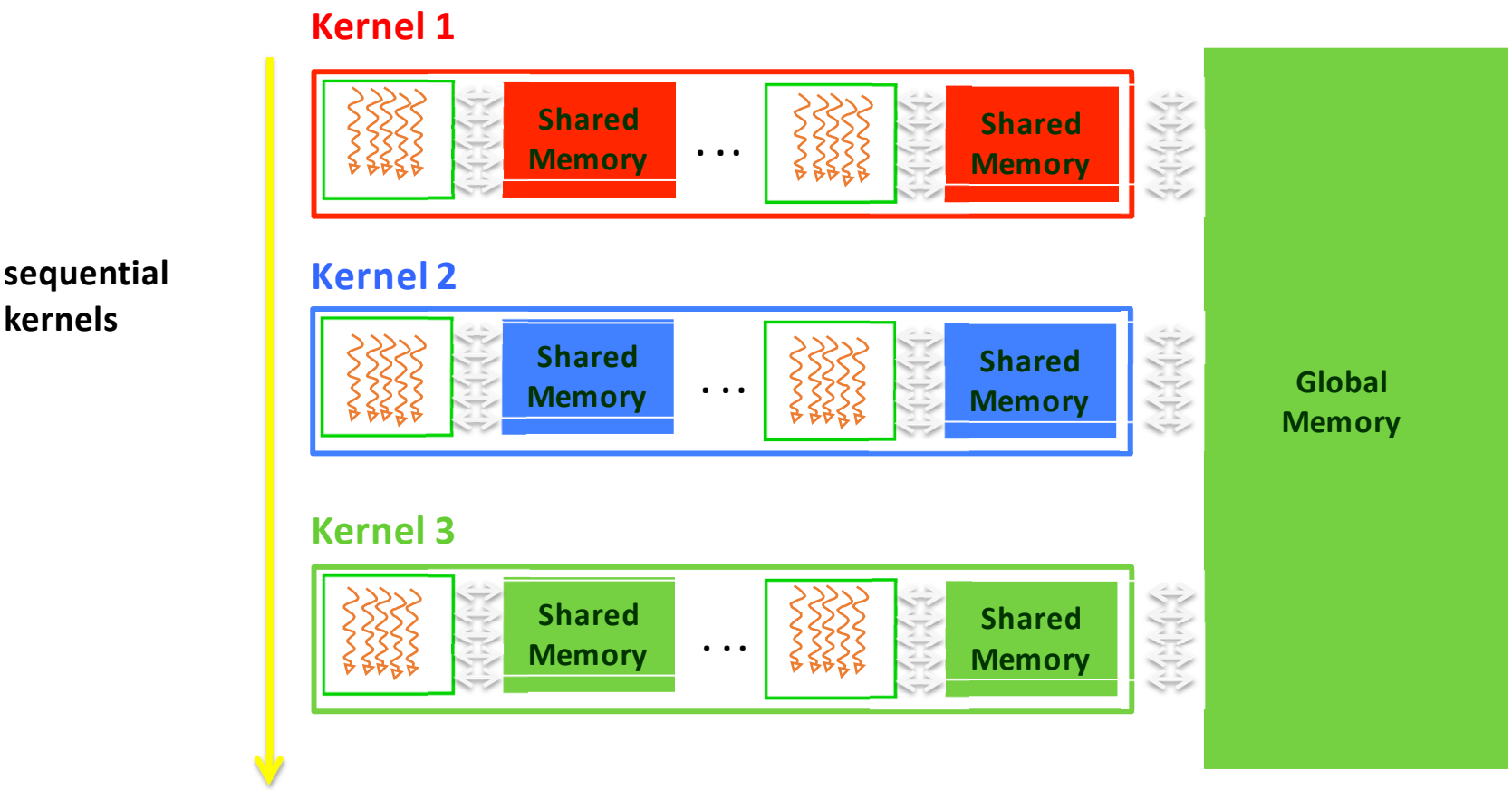
- Sharing data among threads in a block
- User–managed cache (to reduce global memory accesses)

Global Memory

- Allocated explicitly by host (CPU) thread
- Scope: all threads of all kernels
- Data lifetime: determine by host (CPU) thread
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaFree (void* pointer)
- Capacity: large (1-6GB)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s
  - Data access patterns will limit bandwidth achieved in practice
- Common uses
  - Staging data transfers to/from CPU
  - Staging data between kernel launches



Communication and Data Persistence



Managing Device (GPU) Memory

- **Host (CPU) manages device (GPU) memory**

  cudaMalloc (void ** pointer, size_t num_bytes)

  cudaMemset (void* pointer, int value, size_t count)

  cudaFree(void* pointer)

- **Example: allocate and initialize array of 1024 ints on device**

```
// allocate and initialize int x[1024] on device
int n = 1024;
int num_bytes = 1024*sizeof(int);
int* d_x = 0;  // holds device pointer
cudaMalloc((void**)&d_x,  num_bytes);
cudaMemset(d_x,  0,  num_bytes);
cudaFree(d_x);
```

Transferring Data

- cudaMemcpy(void* dst, void* src, size_t num_bytes, enum cudaMemcpyKind direction);
  - returns to host thread after the copy completes
  - blocks CPU thread until all bytes have been copied
  - doesn't start copying until previous CUDA calls complete
- Direction controlled by enum cudaMemcpyKind

  cudaMemcpyHostToDevice

  cudaMemcpyDeviceToHost

  cudaMemcpyDeviceToDevice
- CUDA also provides non-blocking
  - Allows program to overlap data transfer with concurrent computation on host and device
  - Need to ensure that source locations are stable and destination locations are not accessed

Example: SAXPY Kernel [1/4]

```
// [compute] for (i=0; i <n; i++) y[i] =a* x[i] +y[i];
//    Each thread processes one element
__global__void saxpy(int n, float a, float* x, float* y)
{
  int i =threadIdx.x +blockDim.x * blockIdx.x;
  if (i <n) y[i] =a*x[i] +y[i];
}

int main()
{
  ...
  // invoke parallel SAXPYkernel with 256 threads / block
  int nblocks =(n +255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Example:SAXPY Kernel [1/4]

```
// [computes] for (i=0; i <n; i++) y[i] =a* x[i] +y[i];
//    Each thread processes one element
__global__void saxpy(int n, float a, float* x, float*  y)
{
  int   i = threadIdx.x +blockDim.x * blockIdx.x;
  if  (i < n) y[i] = a*x[i] + y[i];
}

int main()
{
  ...
  // invoke parallel SAXPYkernel with 256 threads /  block
  int nblocks =(n +255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Device Code

Example: SAXPY Kernel [1/4]

```
// [computes] for (i=0; i <n; i++) y[i] =a* x[i] +y[i];
//    Each thread processes one element
__global__void saxpy(int n, float a, float* x, float* y)
{
  int i =threadIdx.x +blockDim.x * blockIdx.x;
  if (i <n) y[i] =a*x[i] +y[i];
}

int main()
{
  ...
  // invoke parallel SAXPYkernel with 256 threads / block
  int nblocks =(n +255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  ...
}
```

Host Code

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [2/4]

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n * sizeof(float));
  cudaMalloc((void**) &d_y, n * sizeof(float));

  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyHostToDevice);

  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel [3/4]

```
    // invoke parallel SAXPY kernel with 256 threads / block
    int nblocks = (n + 255)/256;
    saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

    // copy y from device (GPU) memory to host (CPU) memory
    cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

    // do something with the result...


    // free device (GPU) memory
    cudaFree(d_x);
    cudaFree(d_y;

    return 0;
}
```

Example: SAXPY Kernel [3/4]

```
    // invoke parallel SAXPY kernel with 256 threads / block
    int nblocks = (n + 255)/256;
    saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

    // copy y from device (GPU) memory to host (CPU) memory
    cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

    // do something with the result...

    // free device (GPU) memory
    cudaFree(d_x);
    cudaFree(d_y;

    return 0;
}
```

Example: SAXPY Kernel [4/4]

```
void saxpy_serial(int n, float a, float* x, float* y)
{
        for (int i =0; i <n; ++i)
          y[i] =a*x[i] +y[i];
}
// invoke host SAXPY function
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__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];
}
// invoke parallel SAXPY kernel with 256 threads/block
int nblocks =(n +255) / 256;
saxpy<<<nblocks, 256>>>(n, 2.0, x, y);
```

CUDA C Code

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions !

soict.hust.edu.vn/ fb.com/groups/soict