Introduction to Parallel GPU Computing

John Owens
Department of Electrical and Computer Engineering
Institute for Data Analysis and Visualization
University of California, Davis

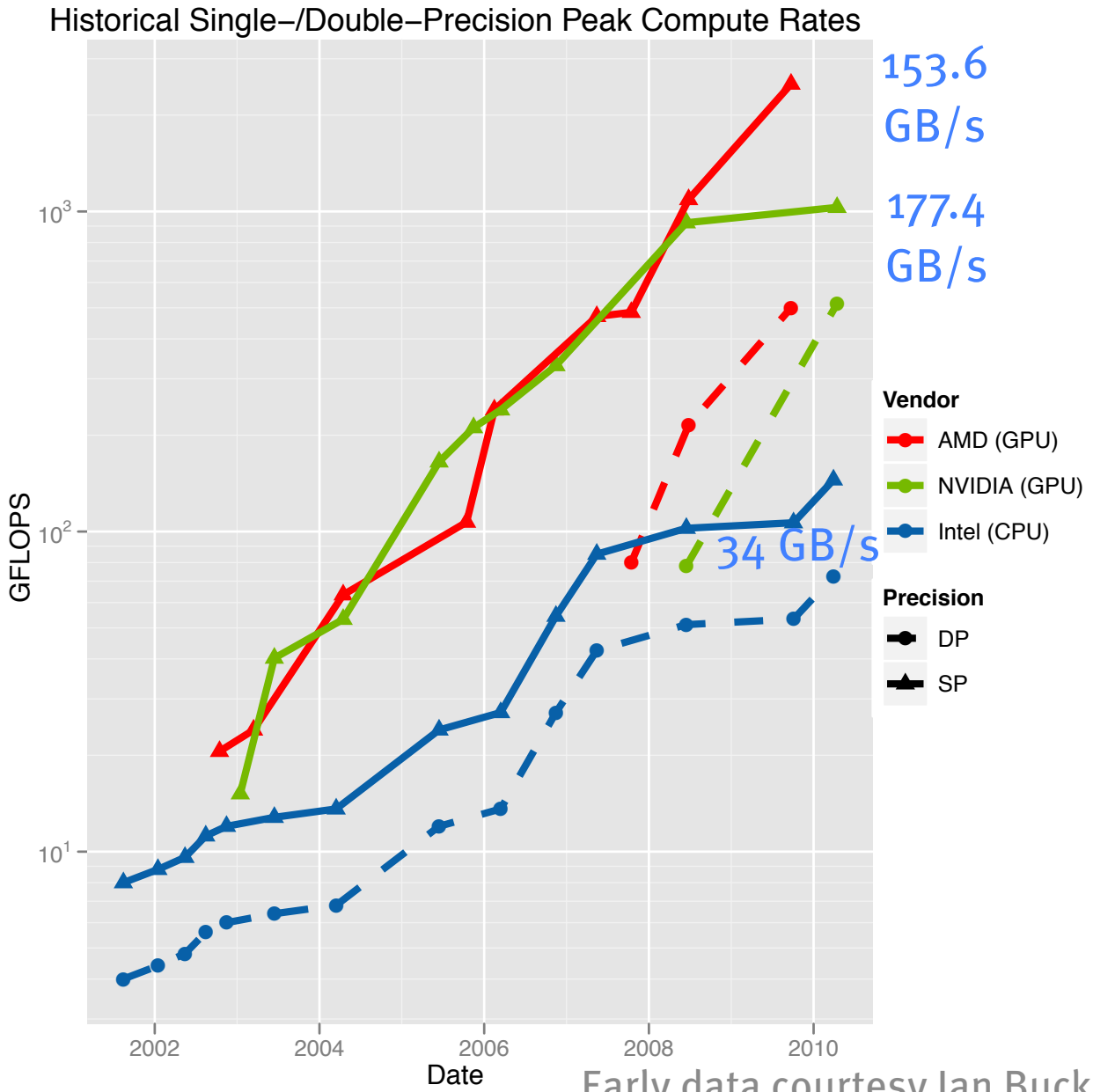
Goals for this Hour

- *Why* GPU computing?
- Multi-GPU computing
- Single-GPU computing

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

—Seymour Cray

Recent GPU Performance Trends



$390
R5870

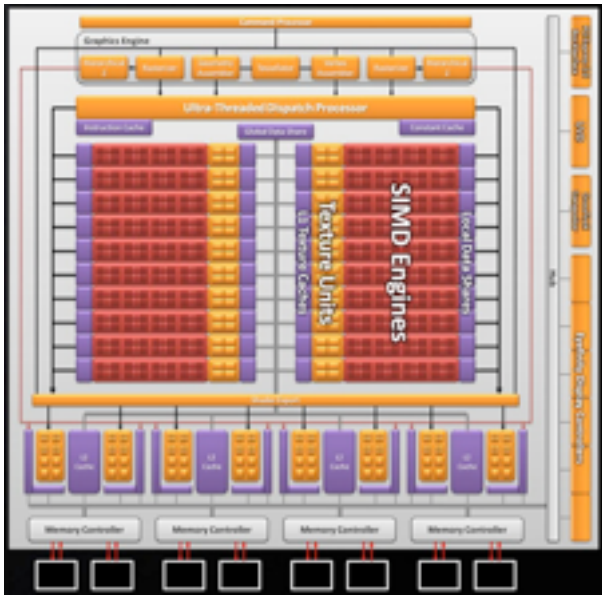
$450
GTX480

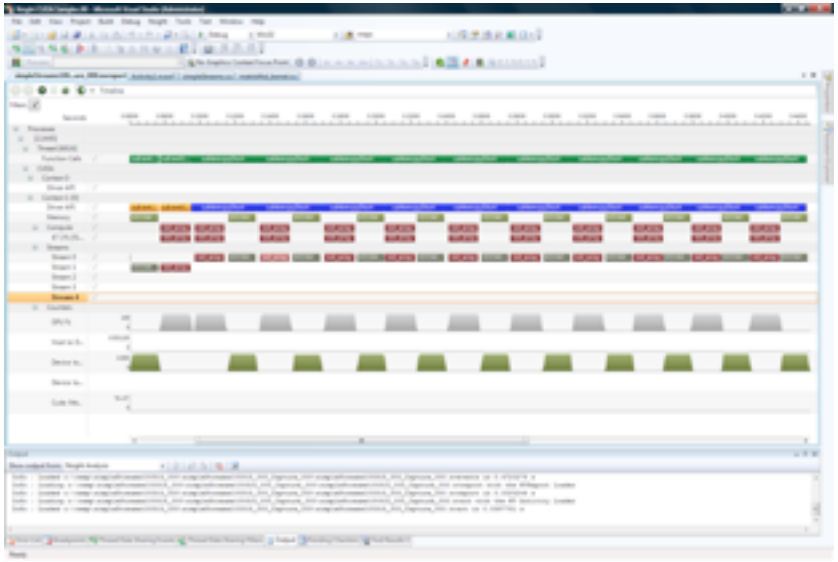
$3692
X7560

Early data courtesy Ian Buck; from Owens et al. 2007 [CGF]

What's new?

- Double precision
- Fast atomics
- Hardware cache & ECC
- (CUDA) debuggers & profilers





Intel ISCA Paper (June 2010)

Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee[†], Changkyu Kim[†], Jatin Chhugani[†], Michael Deisher[†], Daehyun Kim[†], Anthony D. Nguyen[†], Nadathur Satish[†], Mikhail Smelyanskiy[†], Srinivas Chennupaty[⋆], Per Hammarlund[⋆], Ronak Singhal[⋆] and Pradeep Dubey[†]

victor.w.lee@intel.com

[†]Throughput Computing Lab, Intel Corporation

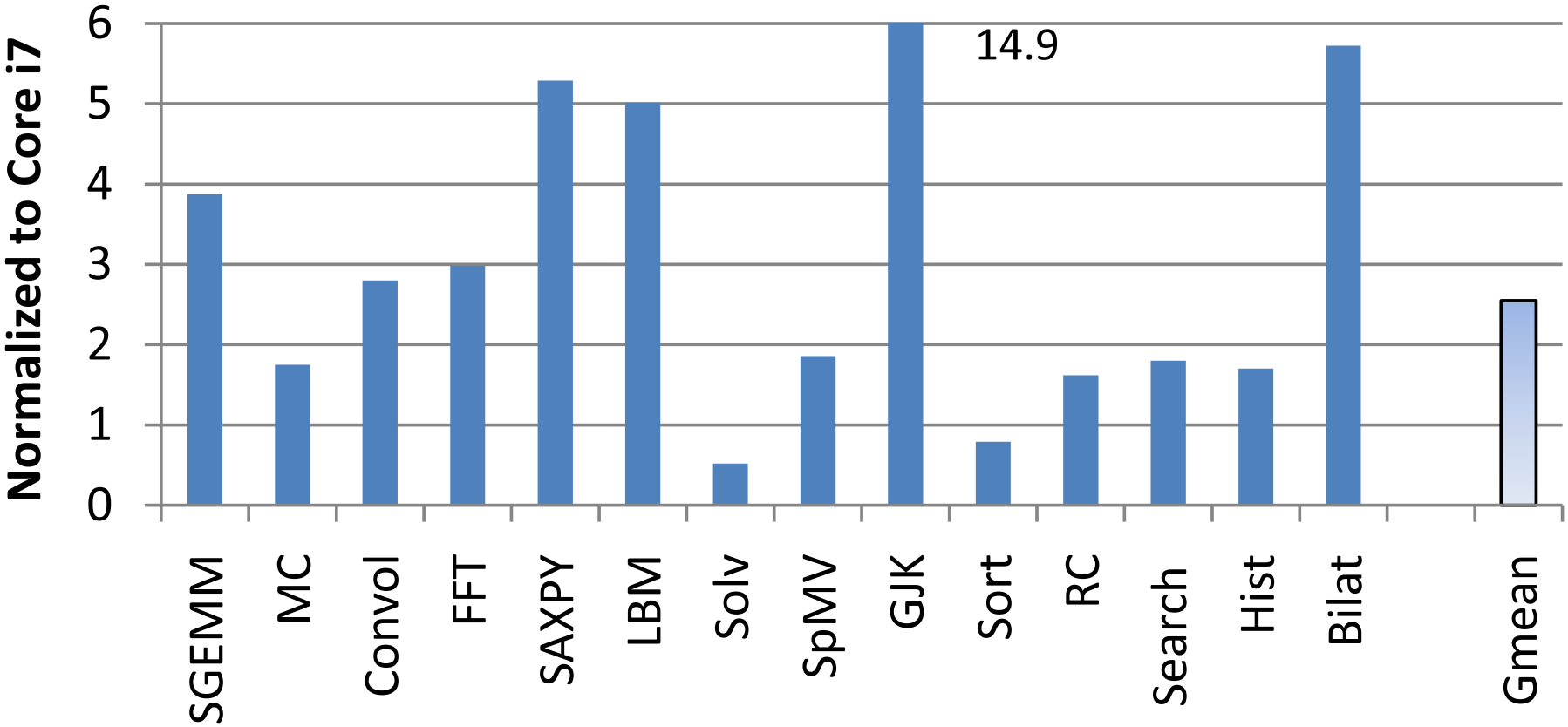
[⋆]Intel Architecture Group, Intel Corporation

ABSTRACT

Recent advances in computing have led to an explosion in the amount of data being generated. Processing the ever-growing data in a timely manner has made throughput computing an important aspect for emerging applications. Our analysis of a set of important throughput computing kernels shows that there is an ample amount of parallelism in these kernels which makes them suitable for to-

The past decade has seen a huge increase in digital content as more documents are being created in digital form than ever before. Moreover, the web has become the medium of choice for storing and delivering information such as stock market data, personal records, and news. Soon, the amount of digital data will exceed exabytes (10^{18}) [31]. The massive amount of data makes storing, cataloging, processing, and retrieving information challenging.

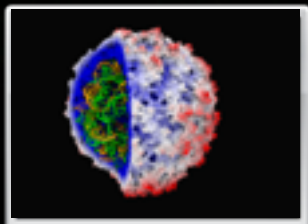
Top-Level Results



CUDA Successes

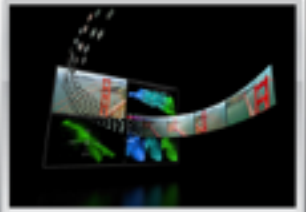
146X

Medical Imaging
U of Utah



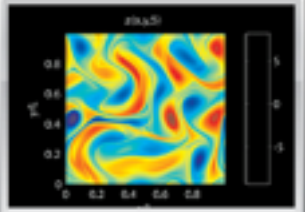
36X

Molecular Dynamics
U of Illinois



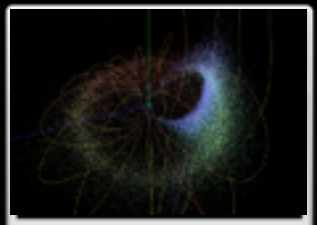
18X

Video Transcoding
Elemental Tech



50X

Matlab Computing
AccelerEyes

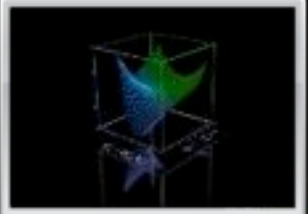


100X

Astrophysics
RIKEN

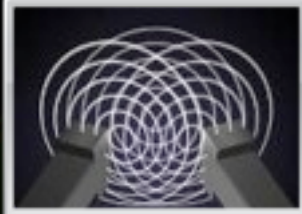
149X

Financial simulation
Oxford



47X

Linear Algebra
Universidad Jaime

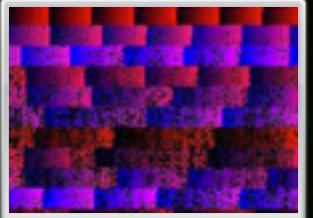


20X

3D Ultrasound
Techniscan

130X

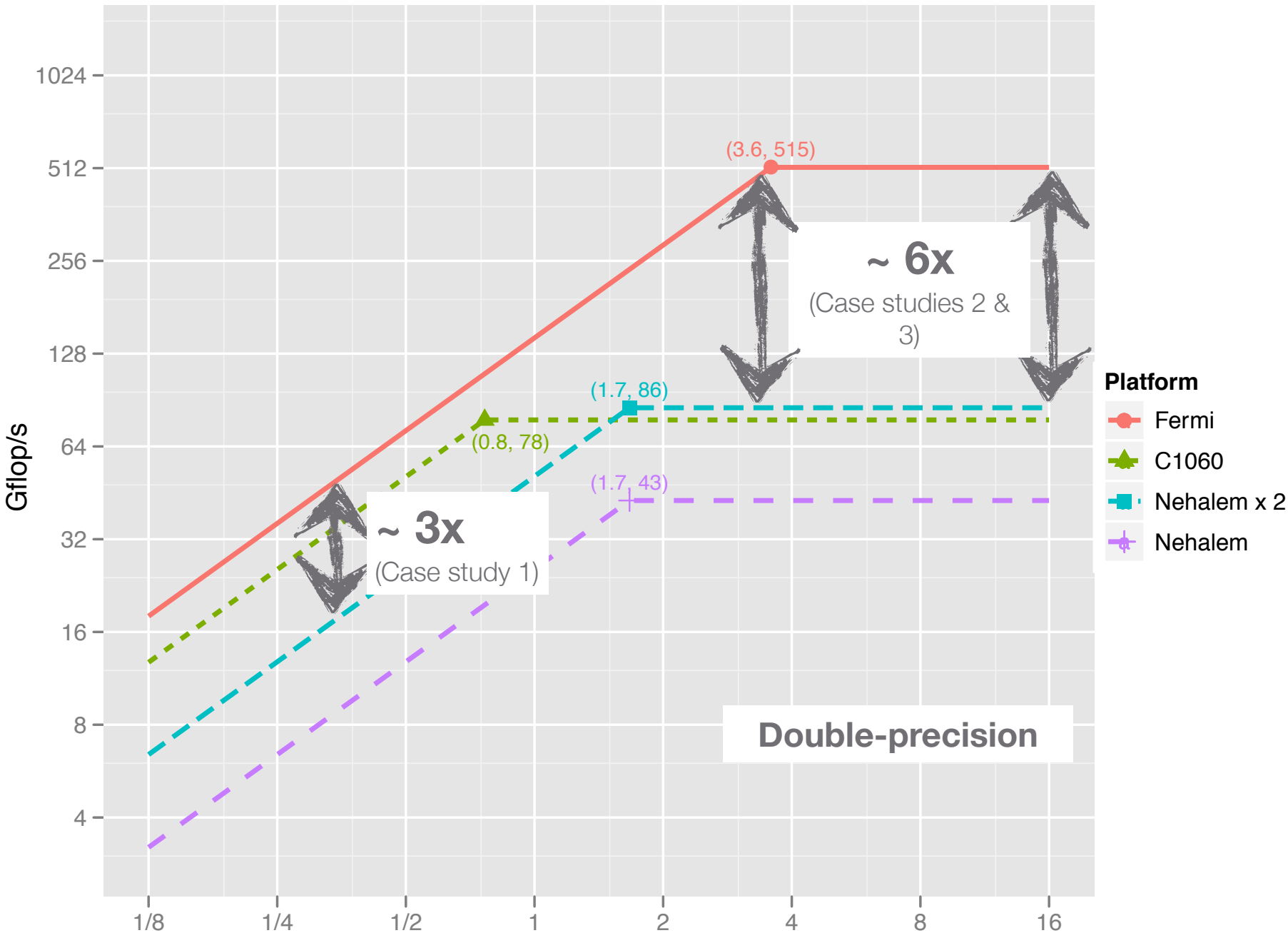
Quantum Chemistry
U of Illinois



30X

Gene Sequencing
U of Maryland

[courtesy David Luebke, NVIDIA]



Courtesy Rich Vuduc, Georgia Tech

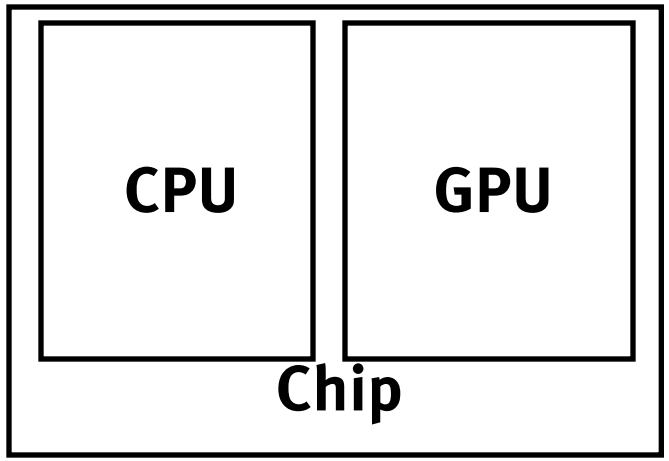
13 Dwarfs

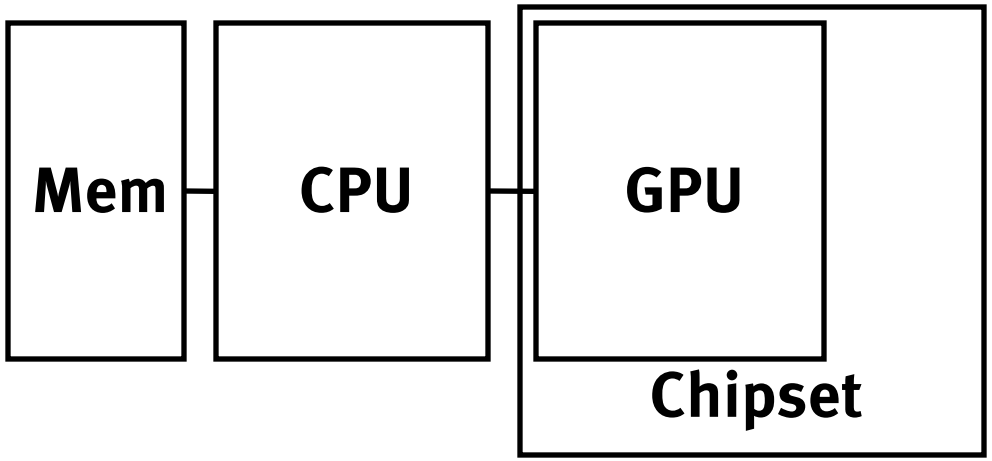
- 1. Dense Linear Algebra
- 2. Sparse Linear Algebra
- 3. Spectral Methods
- 4. N-Body Methods
- 5. Structured Grids
- 6. Unstructured Grids
- 7. MapReduce
- 8. Combinational Logic
- 9. Graph Traversal
- 10. Dynamic Programming
- 11. Backtrack and Branch-and-Bound
- 12. Graphical Models
- 13. Finite State Machines

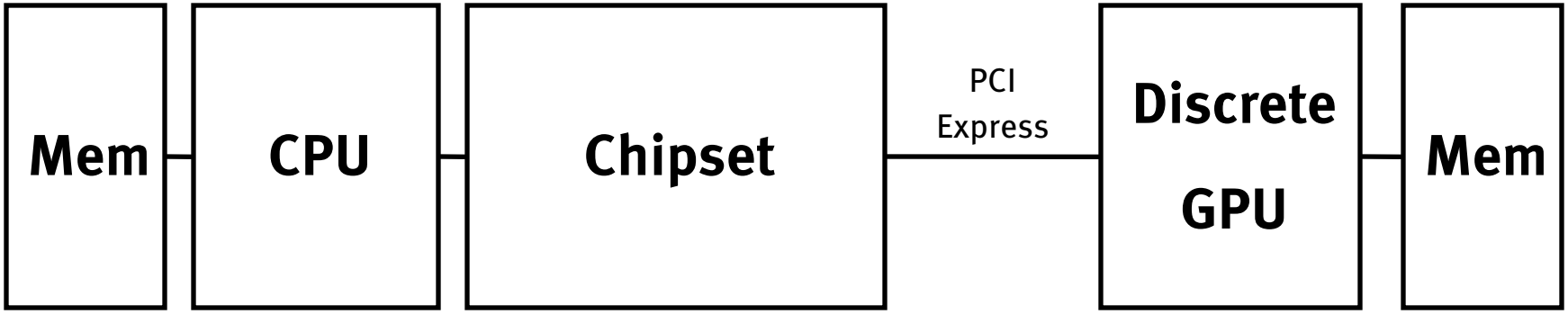
13 Dwarfs

- 1. Dense Linear Algebra
- 2. Sparse Linear Algebra
- 3. Spectral Methods
- 4. N-Body Methods
- 5. Structured Grids
- 6. Unstructured Grids
- 7. MapReduce
- 8. Combinational Logic
- 9. Graph Traversal
- 10. Dynamic Programming
- 11. Backtrack and Branch-and-Bound
- 12. Graphical Models
- 13. Finite State Machines

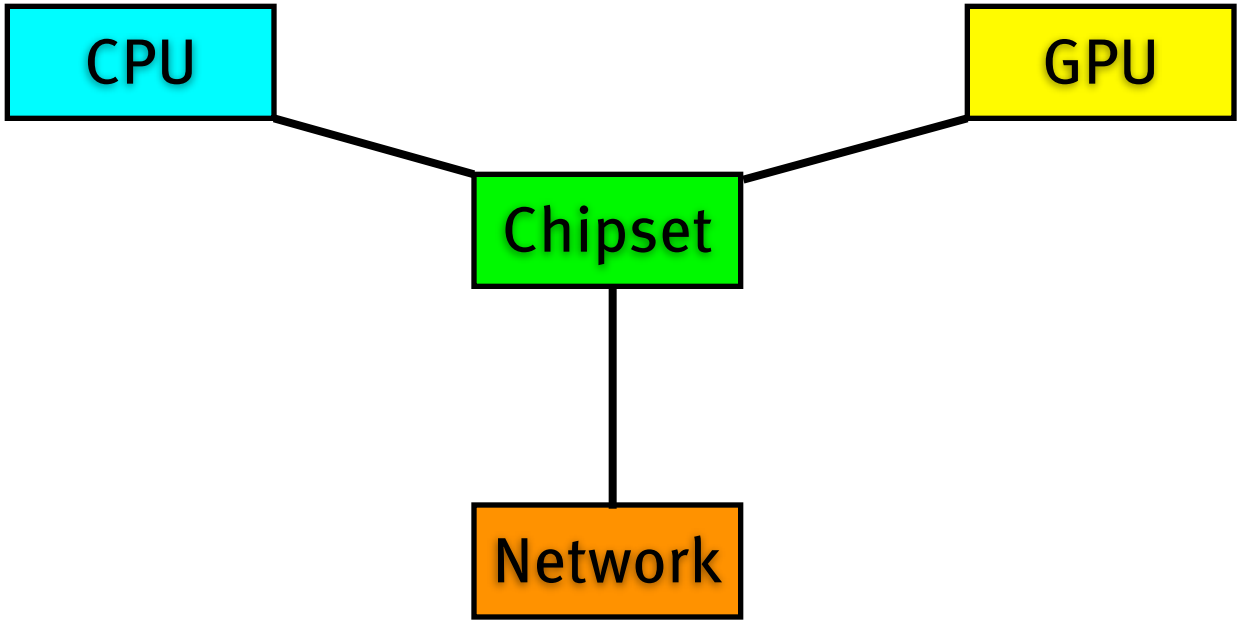
GPU in system (3 alternatives)



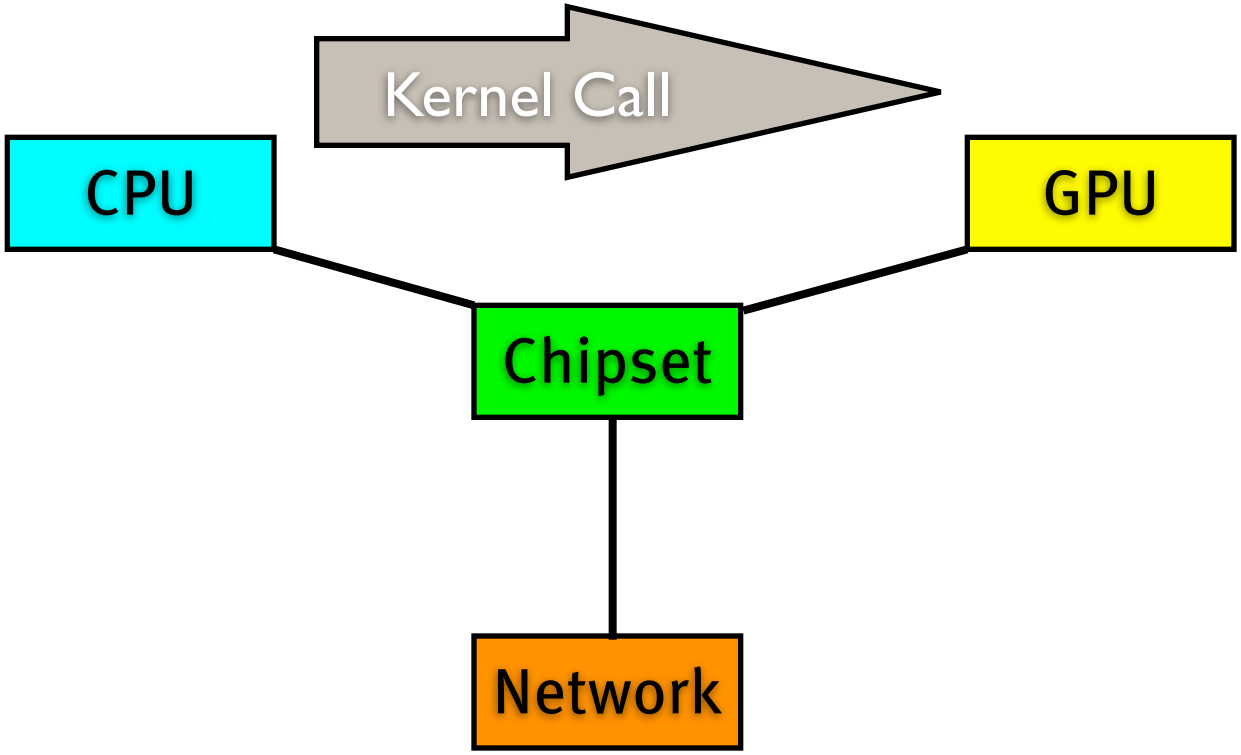




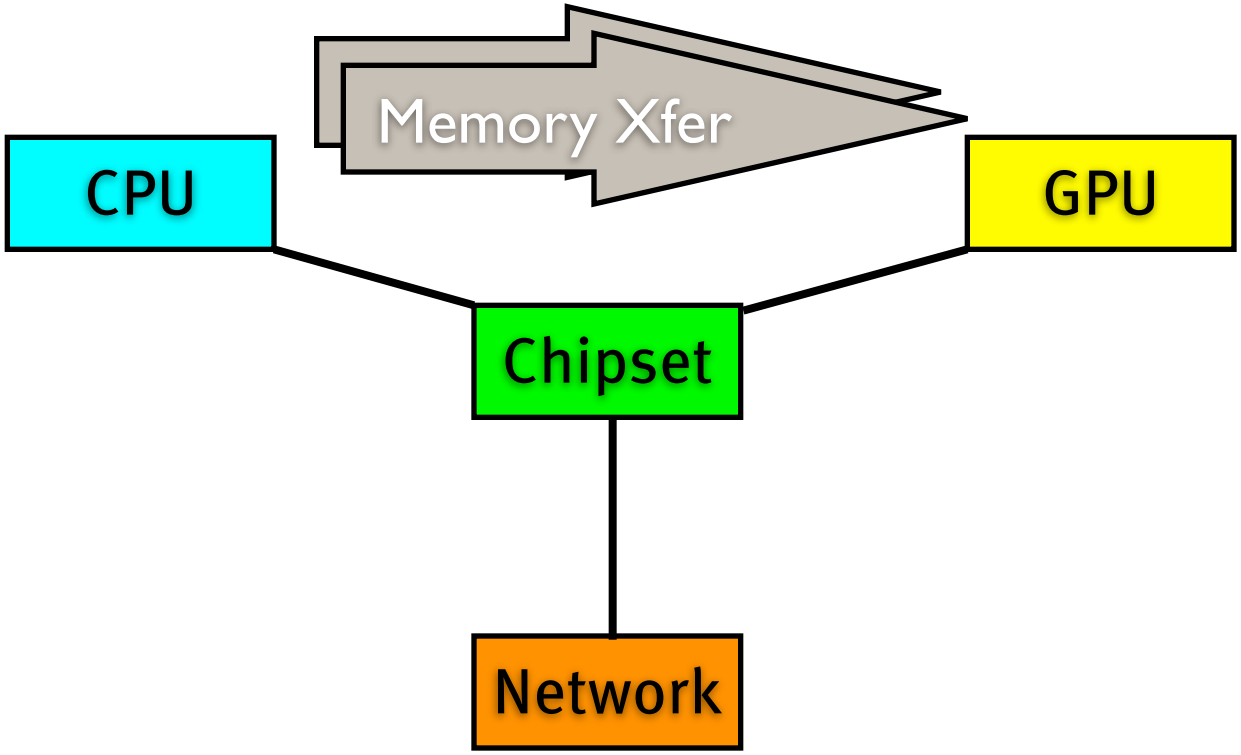
A Modern Computer



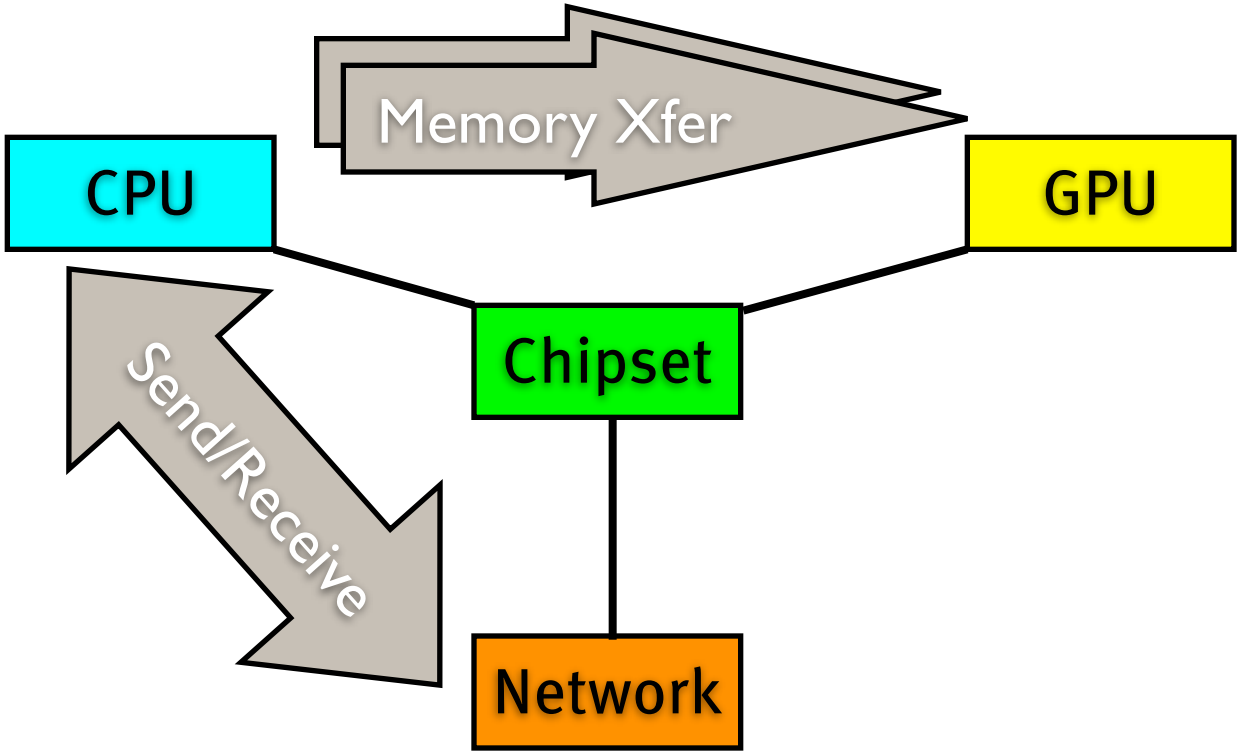
A Modern Computer



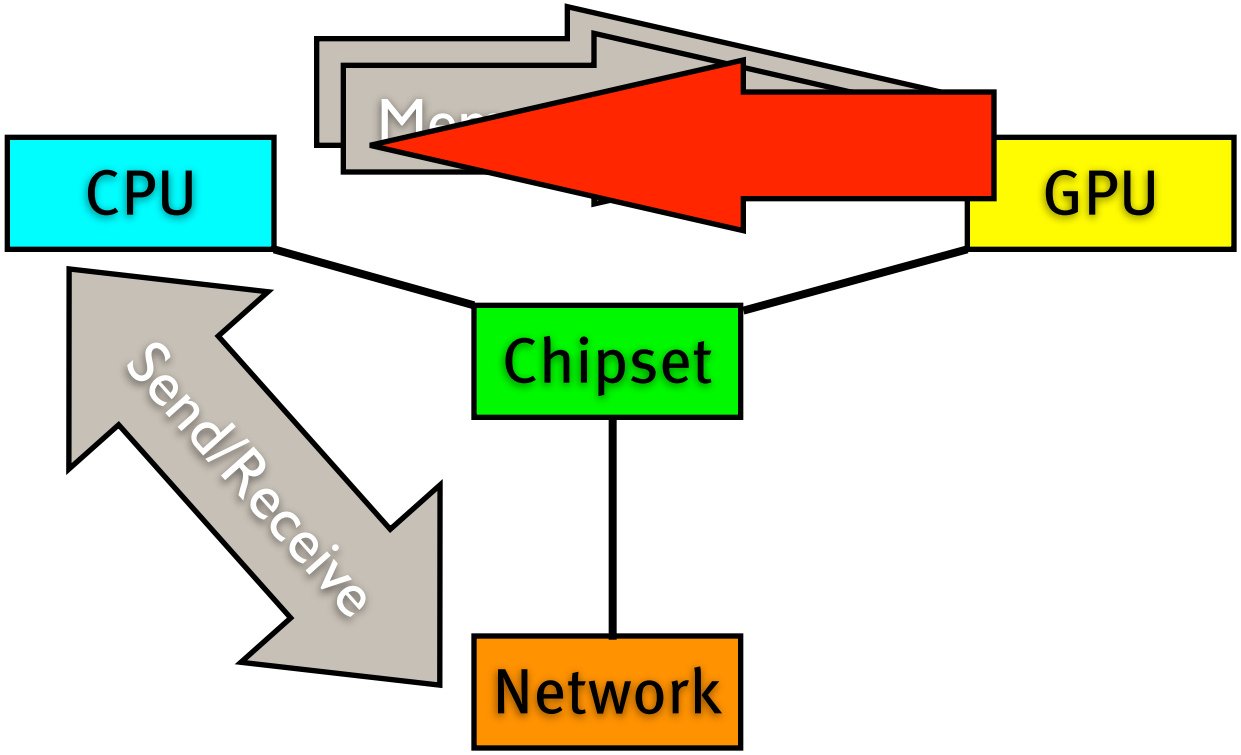
A Modern Computer



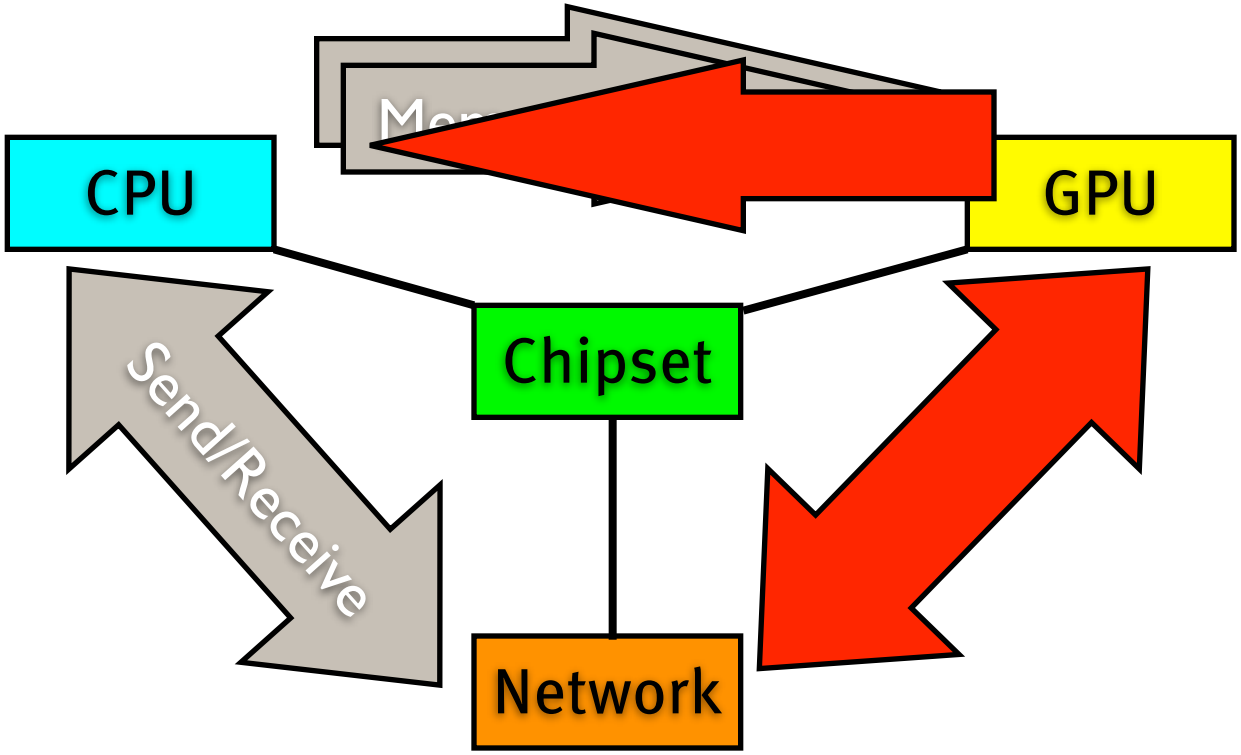
A Modern Computer



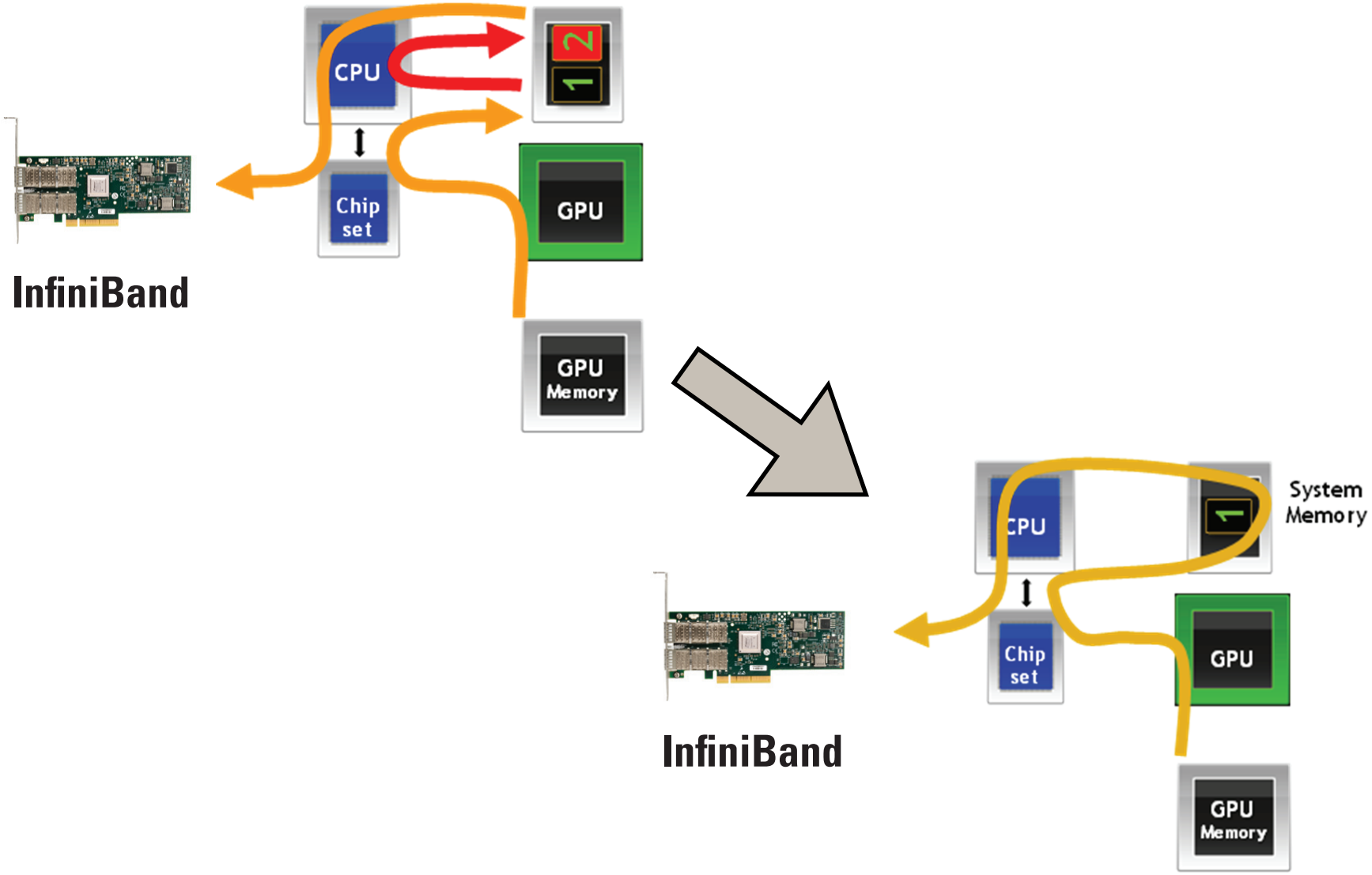
A Modern Computer



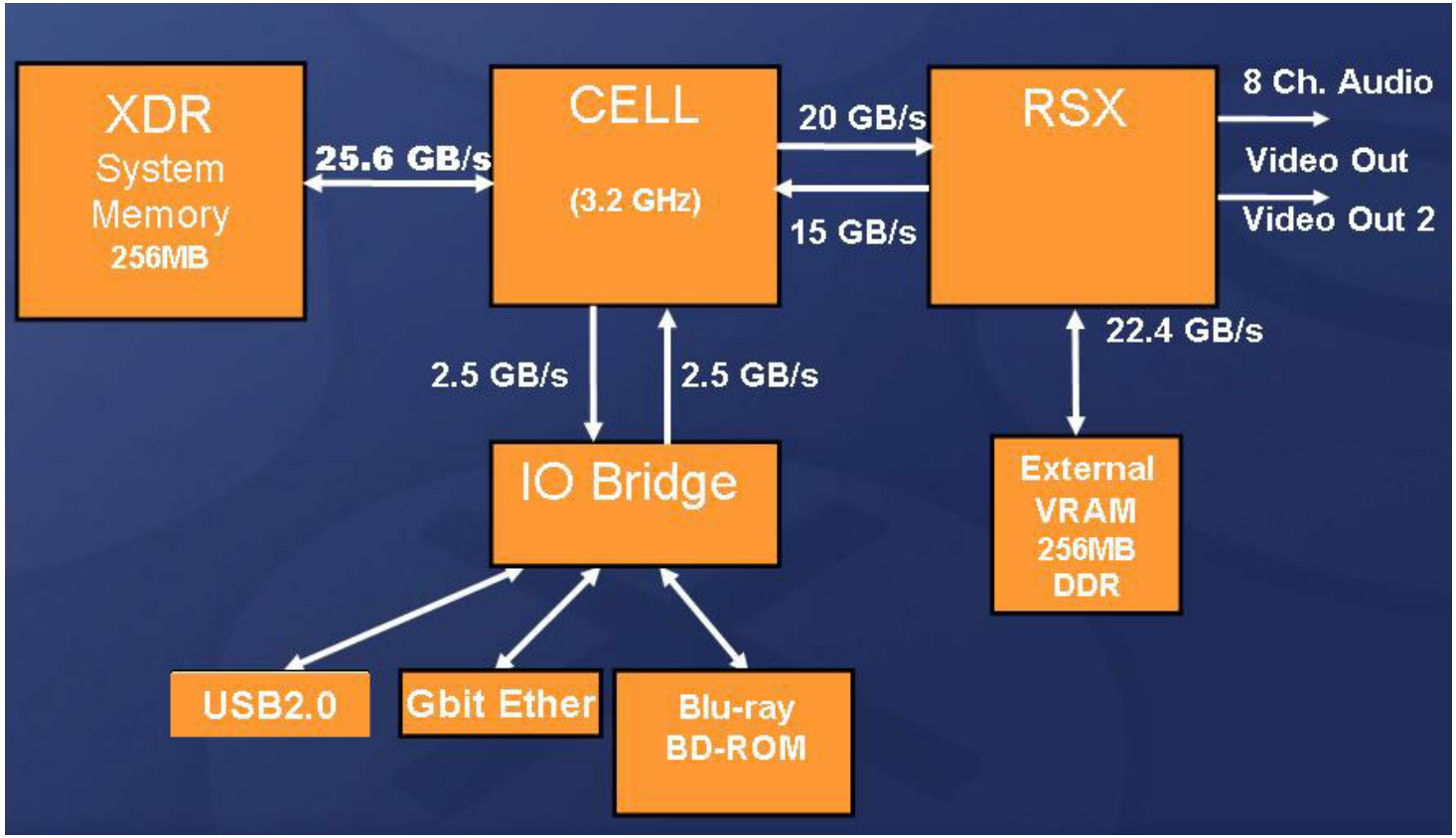
A Modern Computer



Mellanox GPUDirect



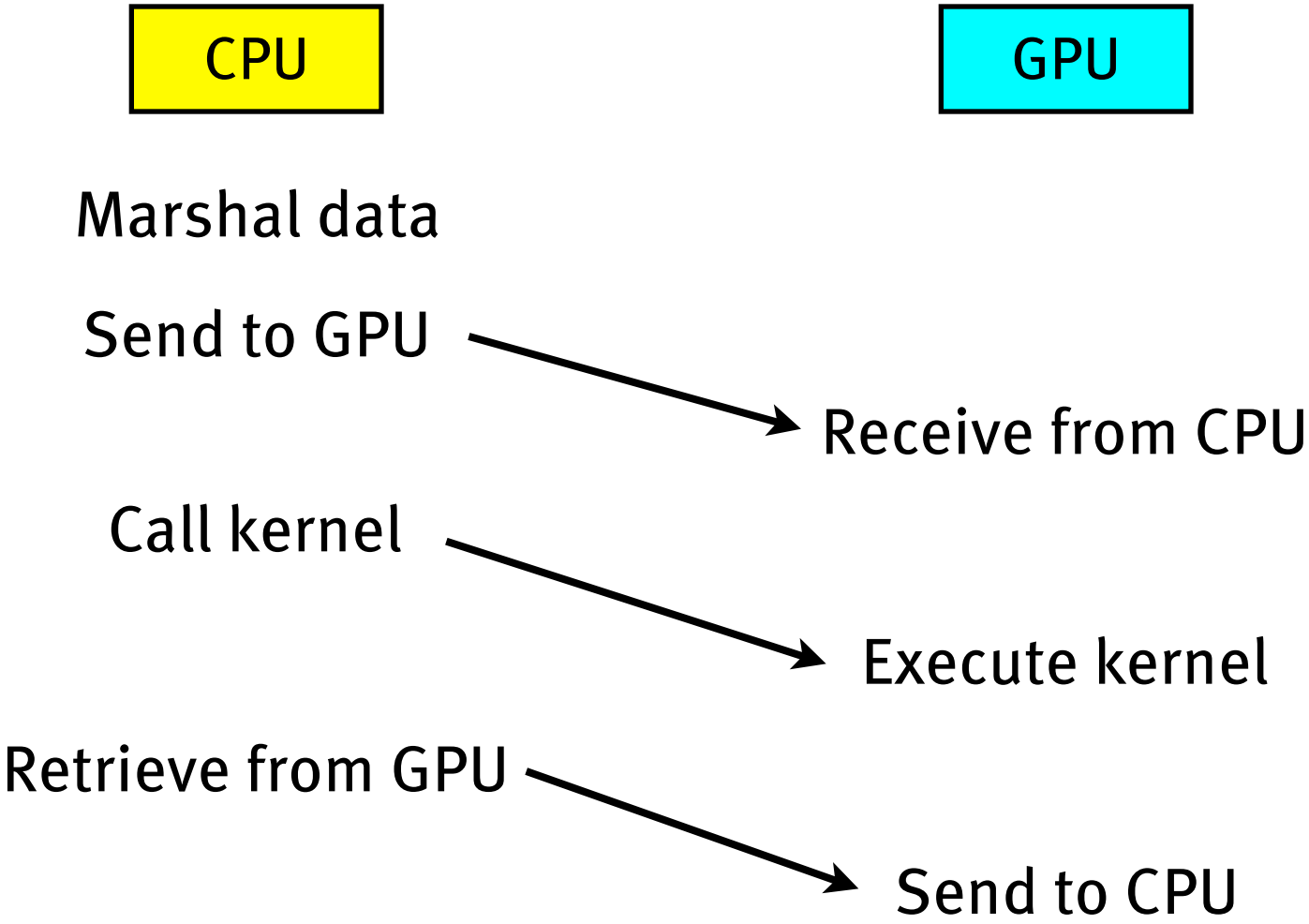
Fast & Flexible Communication



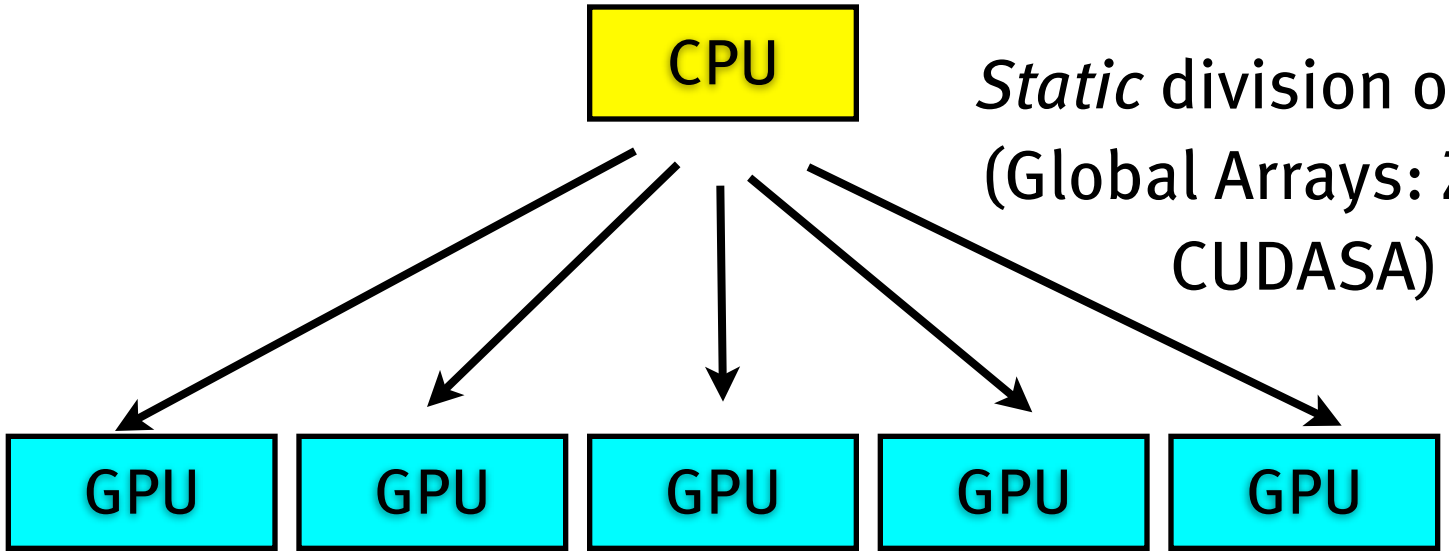
- CPUs are good at creating & manipulating data structures?
- GPUs are good at accessing & updating data structures?

http://www.watch.impress.co.jp/game/docs/20060329/3dps303.jpg

Structuring CPU-GPU Programs

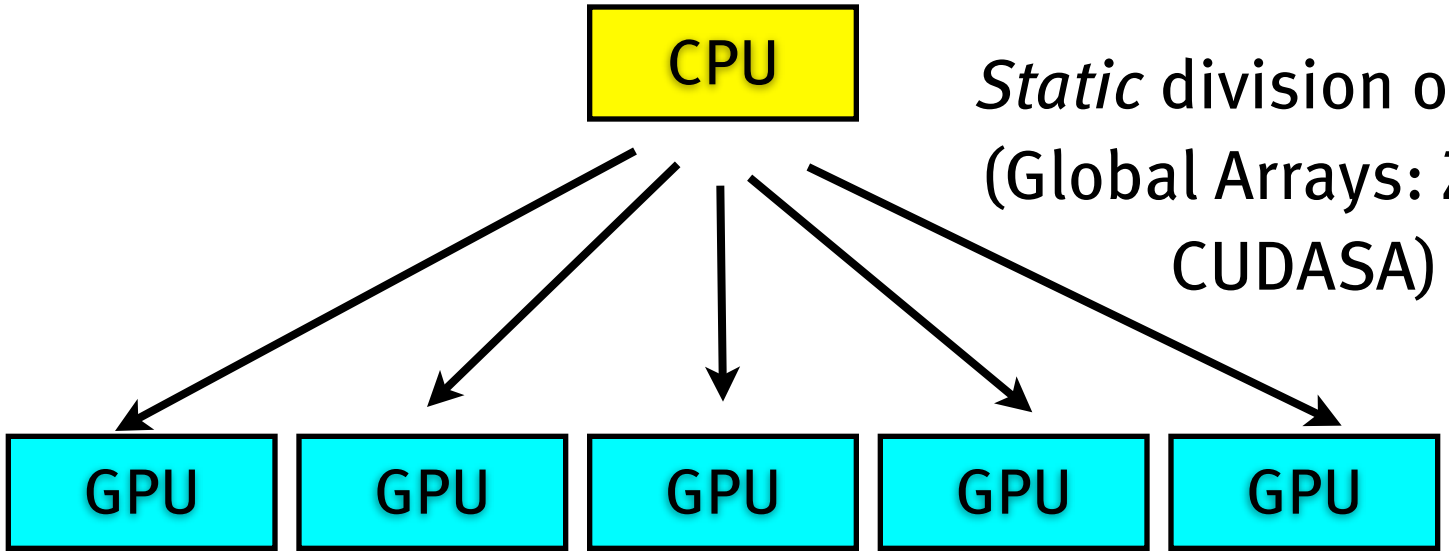


Structuring Multi-GPU Programs



Static division of work (Global Arrays: Zippy, CUDASA)

Structuring Multi-GPU Programs

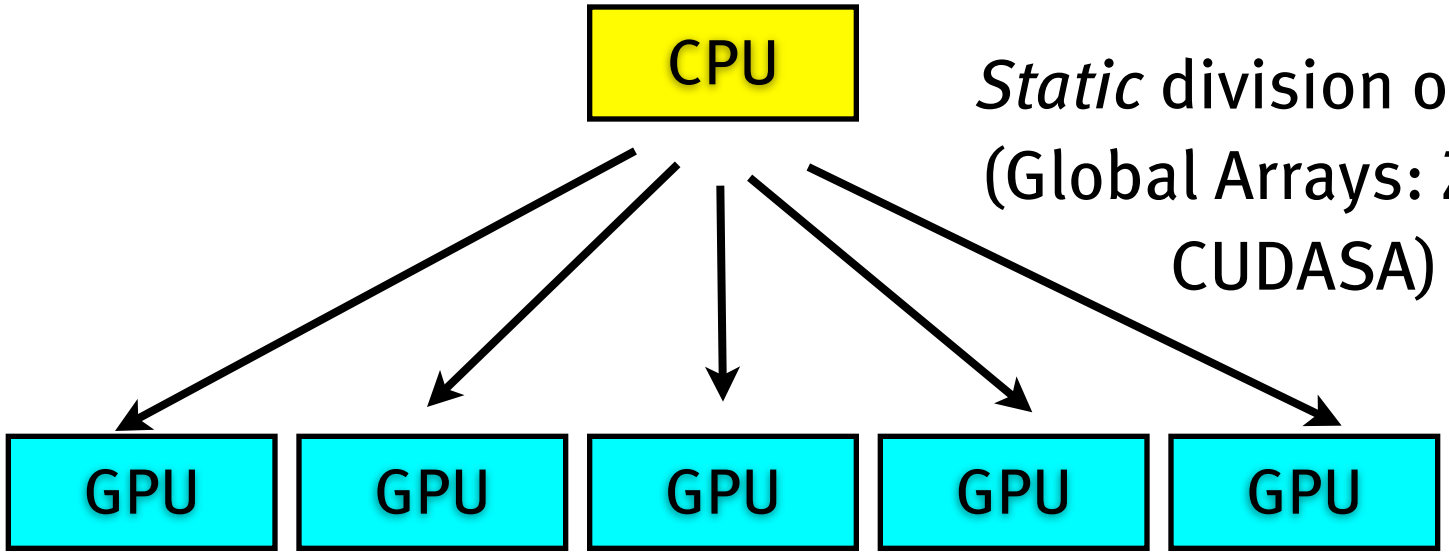


Static division of work (Global Arrays: Zippy, CUDASA)

Want to run on GPU:

```
if (foo == true) {
  GPU[x][bar] = baz;
} else {
  bar = GPU[y][baz];
}
```

Structuring Multi-GPU Programs



Static division of work (Global Arrays: Zippy, CUDASA)

Want to run on GPU:

```
if (foo == true) {
  GPU[x][bar] = baz;
} else {
  bar = GPU[y][baz];
}
```

Instead, *GPU as slave*.

Goal: GPU as first-class citizen.

Our Research Program

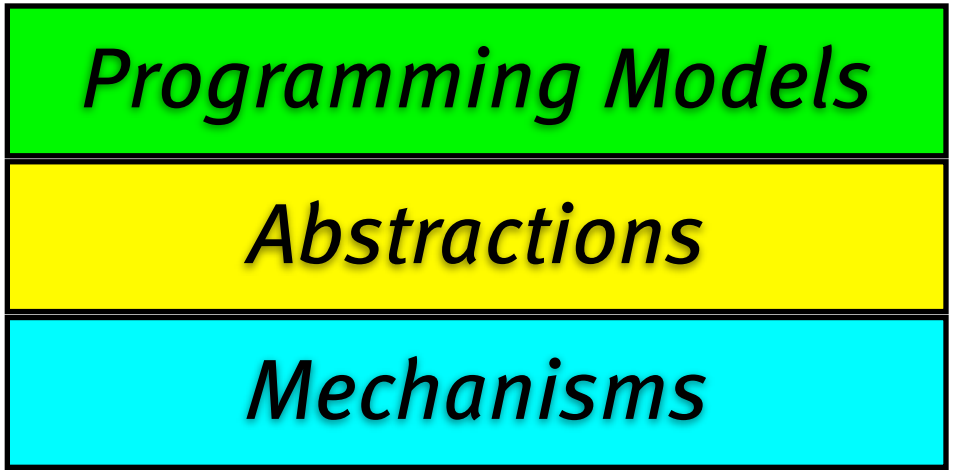
Programming Models

Abstractions

Mechanisms

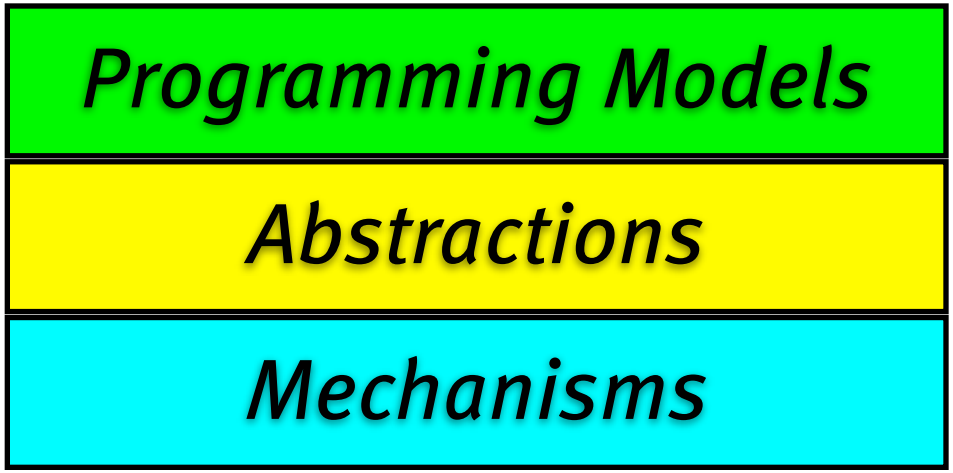
Example

- Abstraction: GPU initiates network send
- Problems:
  - GPU can't communicate with NI
  - GPU signals CPU



Example

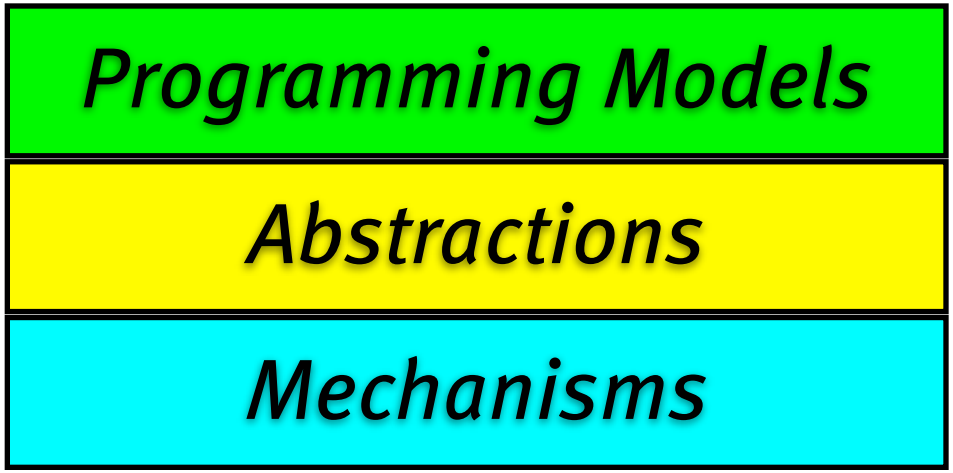
- Abstraction: GPU initiates network send
- Solution:
  - CPU allocates "mailbox" in GPU mem
  - GPU sets mailbox to initiate network send
  - CPU polls mailbox



Example

Take-home: Abstraction does not change even if underlying mechanisms change

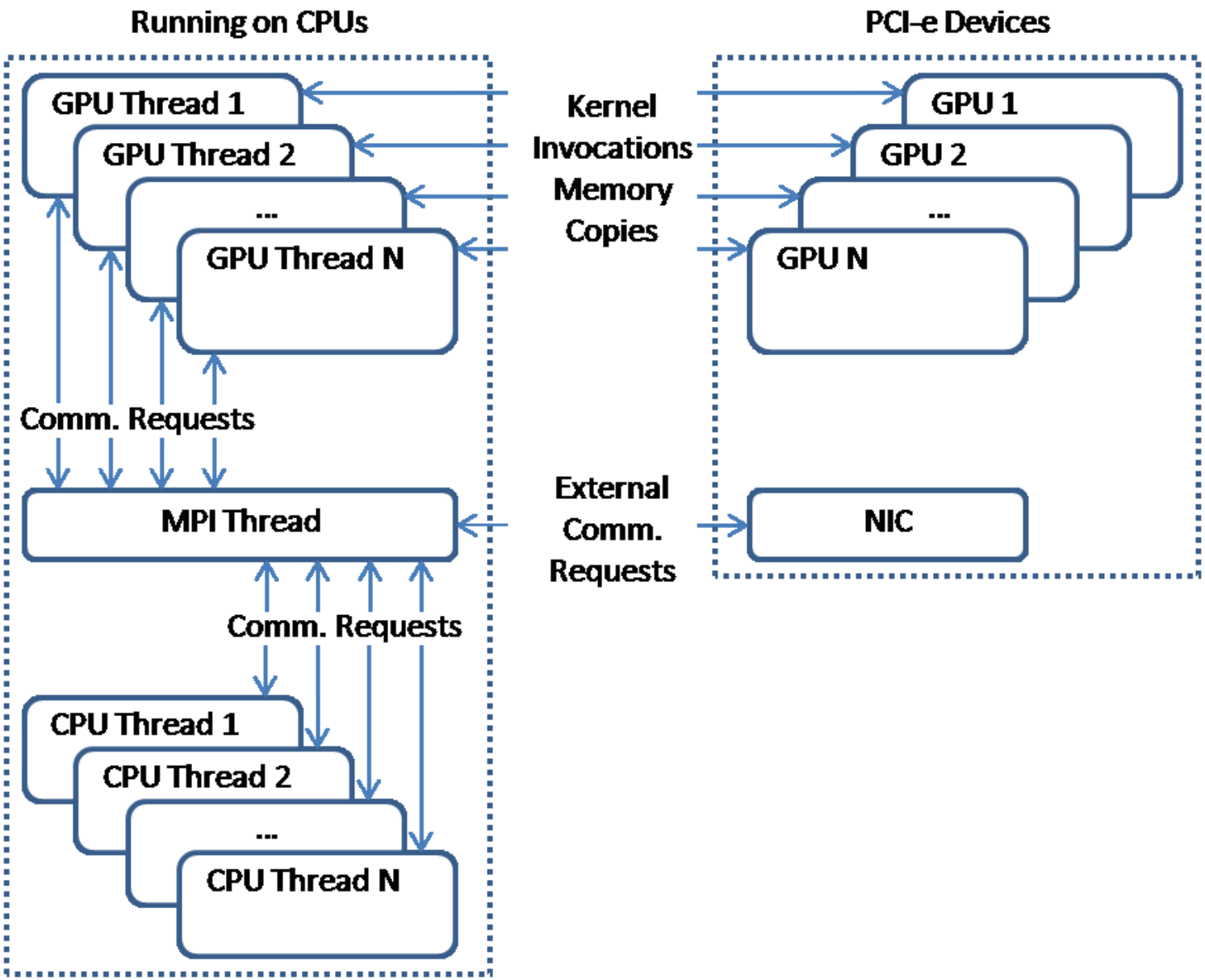
- Abstraction: GPU initiates network send
- Solution:
  - CPU allocates "mailbox" in GPU mem
  - GPU sets mailbox to initiate network send
  - CPU polls mailbox



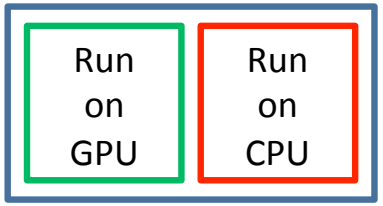
DCGN: MPI-Like Programming Model

- Distributed Computing for GPU Networks (DCGN, pronounced decagon)
  - MPI-like interface
- Allows communication between all CPUs and GPUs in system
  - Allow GPU to source/sink communication
  - Multithreaded communication via MPI
  - Both synchronous and asynchronous (<- overlap!)
  - Collectives
  - Multiplex MPI addresses (“slots”)

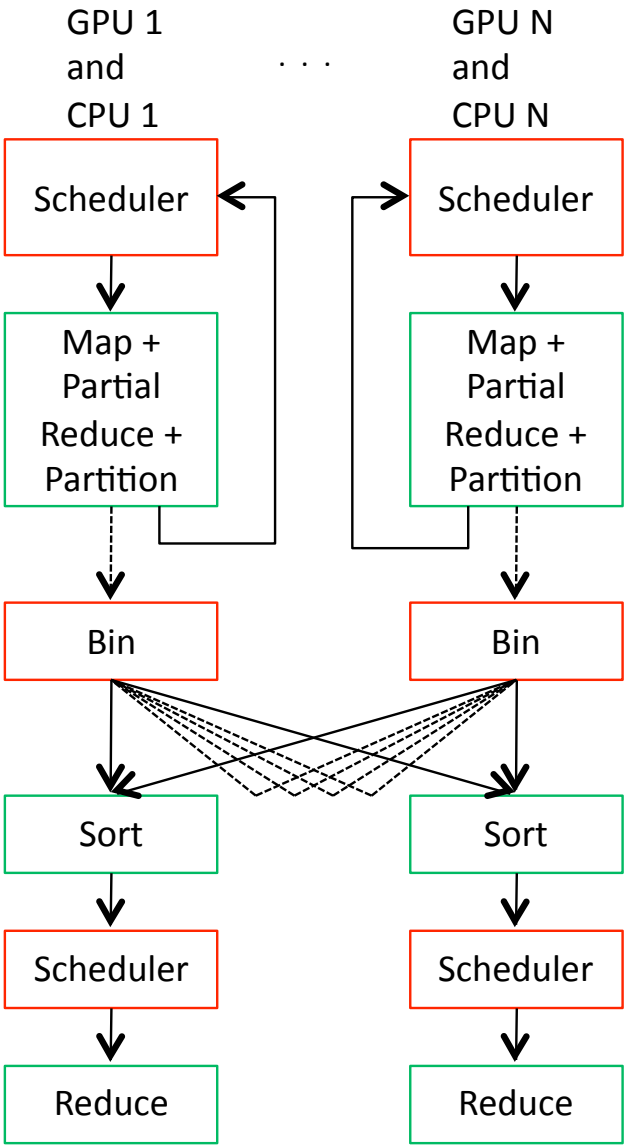
Architecture



MapReduce: Keys to Performance

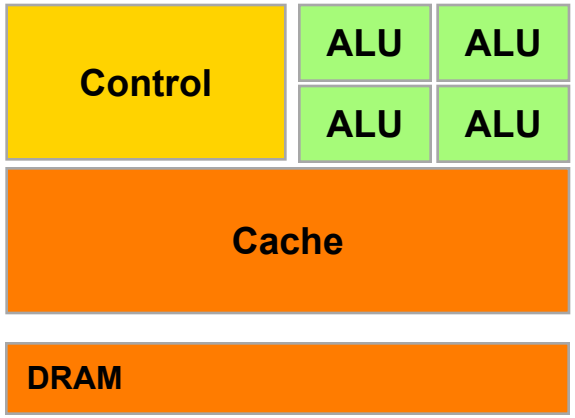


- Process data in chunks
  - More efficient transmission & computation
  - Also allows out of core
- Overlap computation and communication
- Accumulate
- Partial Reduce

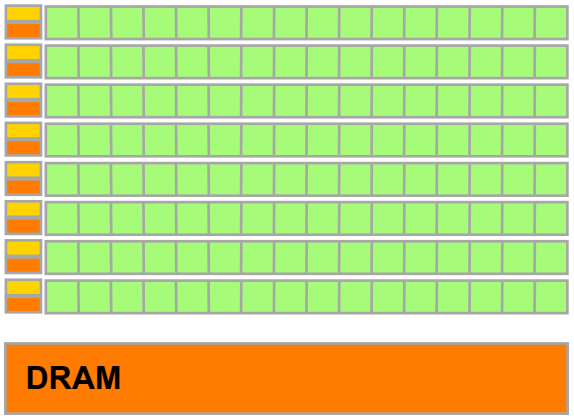


Why is data-parallel computing fast?

- The GPU is specialized for compute-intensive, highly parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control



CPU



GPU

Programming Model: A Massively Multi-threaded Processor

- Move data-parallel application portions to the GPU
- Differences between GPU and CPU threads
  - Lightweight threads
  - GPU supports 1000s of threads
- Today:
  - GPU hardware
  - CUDA programming environment

Big Idea #1

- One thread per data element.
- Doesn’t this mean that large problems will have millions of threads?

Big Idea #2

- Write one program.
- That program runs on ALL threads in parallel.
- NVIDIA's terminology here is "SIMT": single-instruction, multiple-thread.
  - Roughly: SIMD means many threads run in lockstep; SIMT means that some divergence is allowed and handled by the hardware

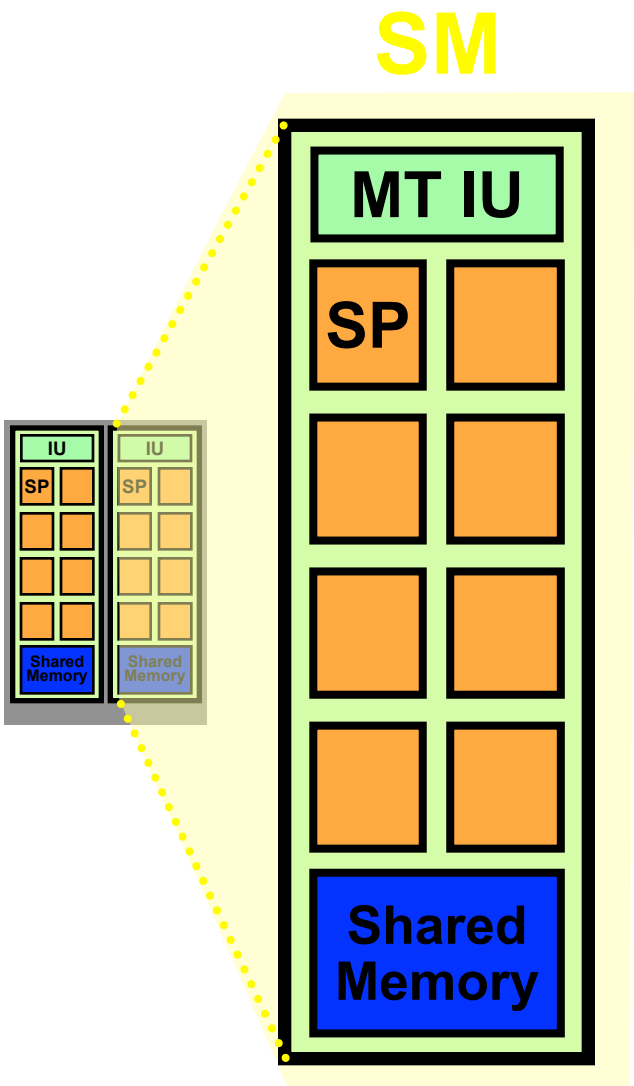
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One SIMT kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA *must* use 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions:
Device = GPU; *Host* = CPU
Kernel = function that runs on the device

SM Multithreaded Multiprocessor

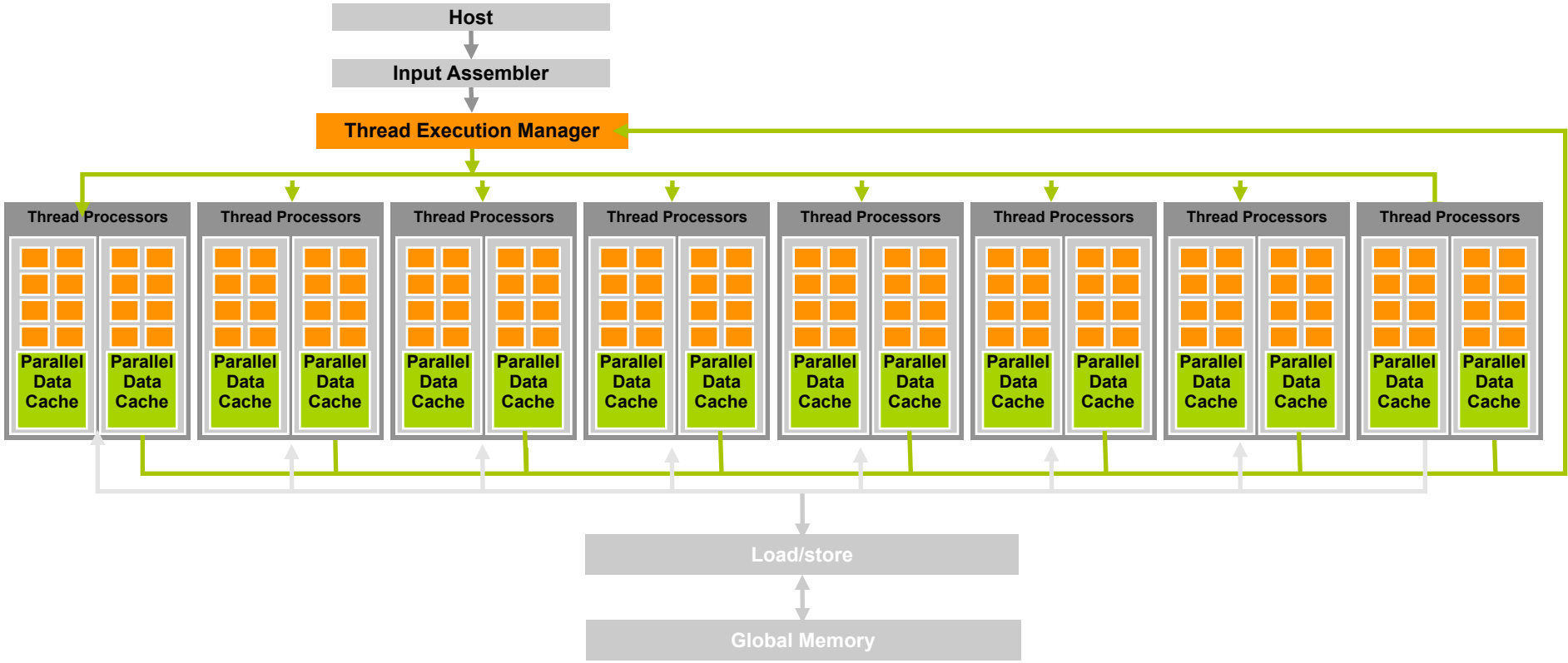
This figure is 1 generation old



- **Each SM runs a *block* of threads**
- SM has 8 SP Thread Processors
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- Scalar ISA
- Up to 768 threads, hardware multithreaded
- 16KB Shared Memory
  - Concurrent threads share data
  - Low latency load/store

GPU Computing (G80 GPUs)

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing



NVIDIA Fermi

Performance

- 7x Double Precision of CPUs
- IEEE 754-2008 SP & DP Floating Point

Flexibility

- Increased Shared Memory from 16 KB to 64 KB
- Added L1 and L2 Caches
- ECC on all Internal and External Memories
- Enable up to 1 TeraByte of GPU Memories
- High Speed GDDR5 Memory Interface

Usability

- Multiple Simultaneous Tasks on GPU
- 10x Faster Atomic Operations
- C++ Support
- System Calls, printf support

Slide courtesy NVIDIA, image from http://images.anandtech.com/reviews/video/NVIDIA/GTX460/fullGF100.jpg

Big Idea #3

- Latency hiding.
  - It takes a long time to go to memory.
  - So while one set of threads is waiting for memory ...
  - ... run another set of threads during the wait.
    - In practice, 32 threads run in a “warp” and an efficient program usually has 128–256 threads in a block.

HW Goal: Scalability

- Scalable execution
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

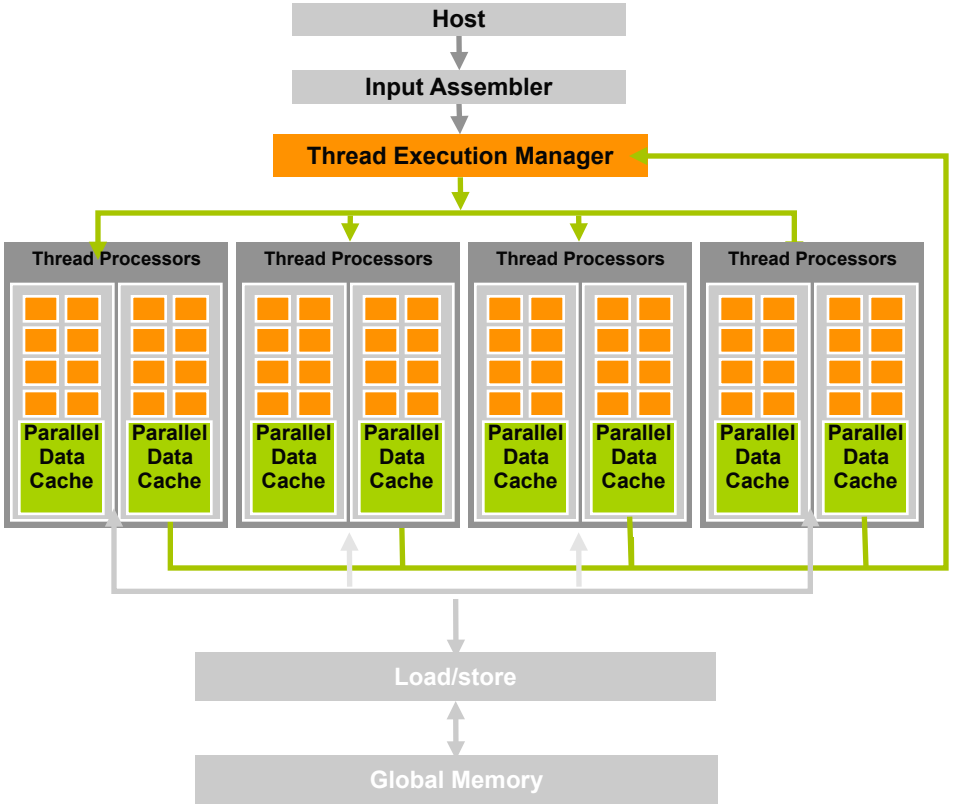
- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

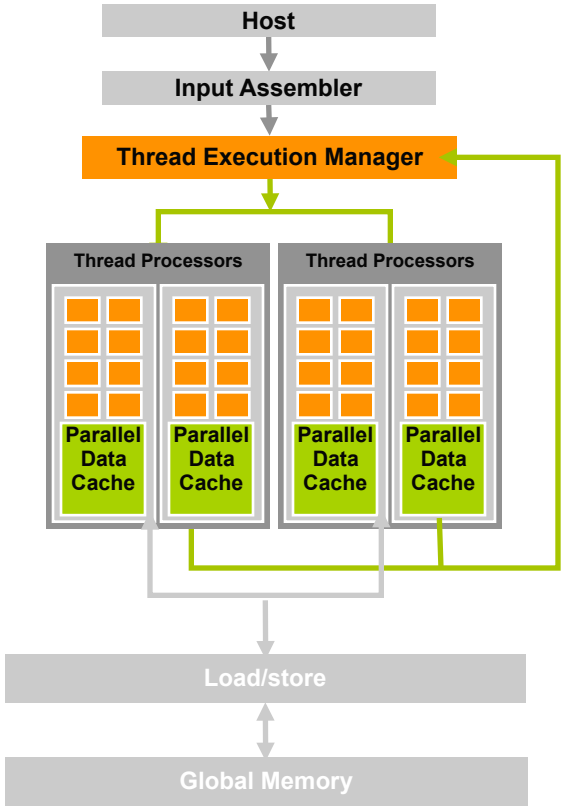
This is very important.

- Hardware distributes ***independent*** blocks to SMs as available

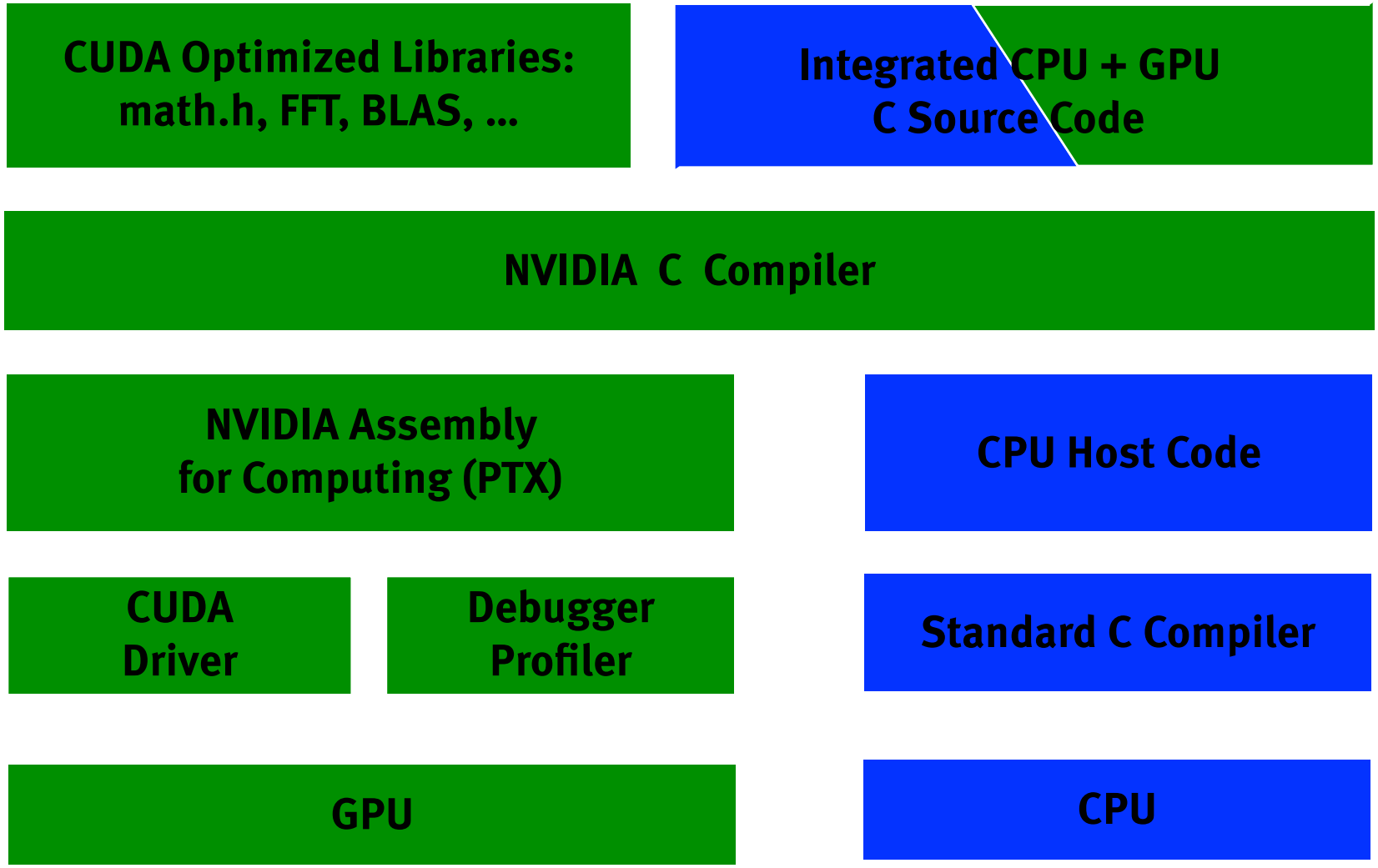
Scaling the Architecture

- Same program
- Scalable performance

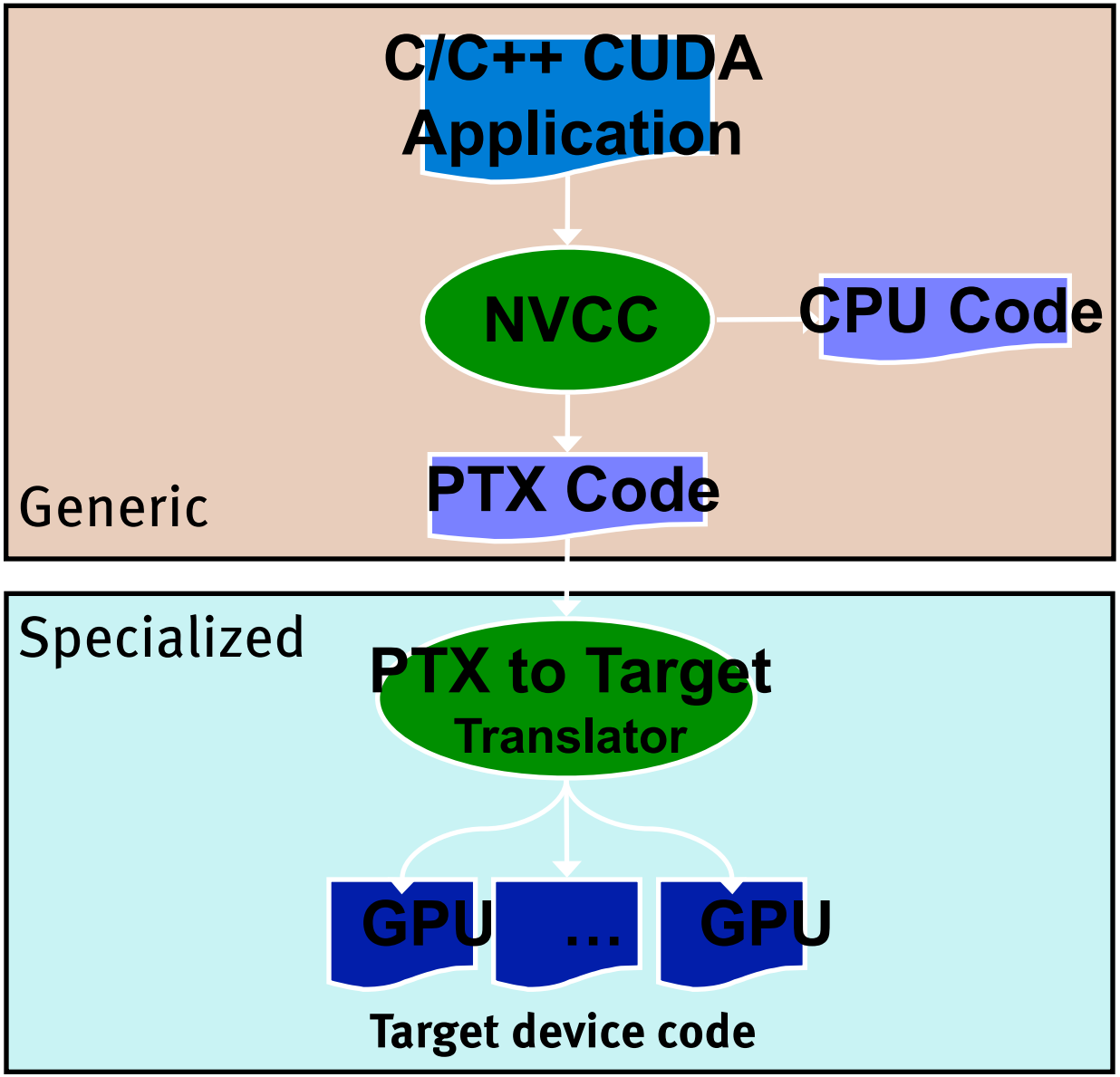




CUDA Software Development Kit

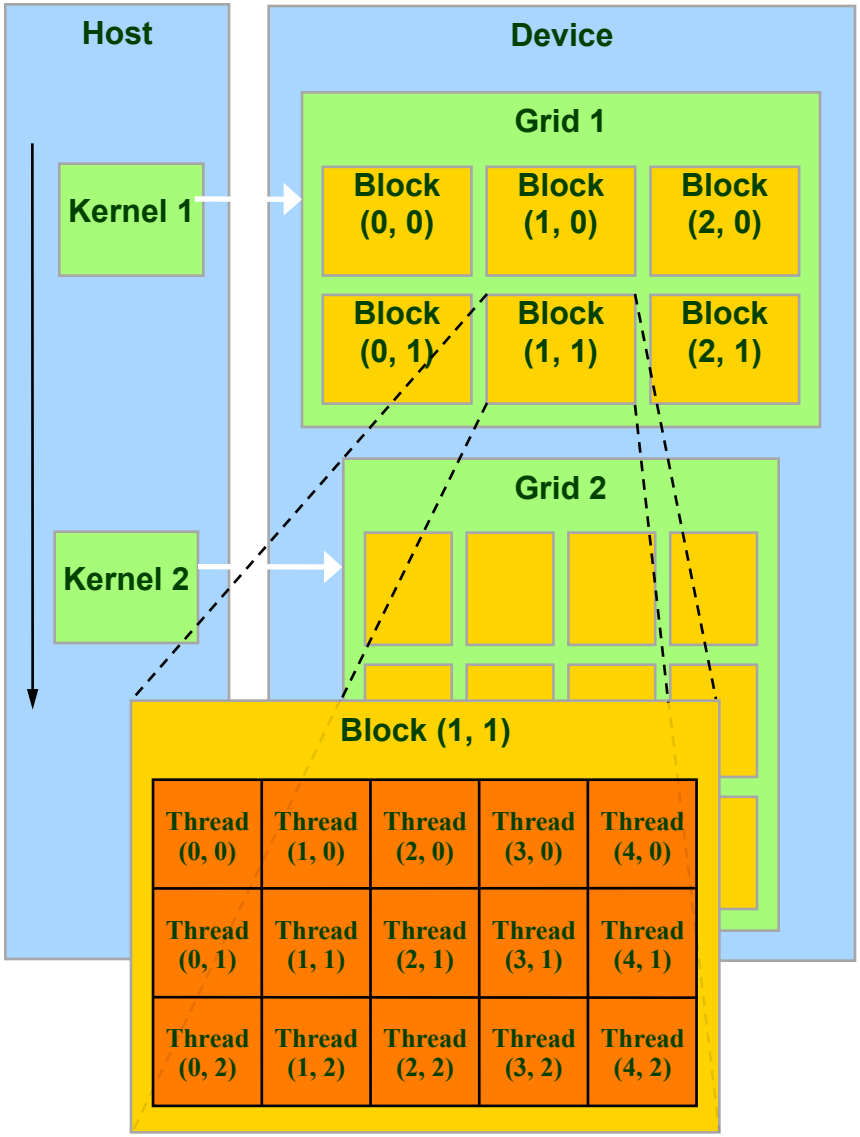


Compiling CUDA for GPUs



Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate
  - Blocks are *independent*



Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - 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*

Big Idea #4

- Organization into independent blocks allows scalability / different hardware instantiations
  - If you organize your kernels to run over many blocks ...
  - ... the same code will be efficient on hardware that runs one block at once and on hardware that runs many blocks at once

CUDA: Programming GPU in C

- Philosophy: provide minimal set of extensions necessary to expose power
- 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;          // shared within thread block
```

- Extend function invocation syntax for parallel kernel launch

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

- Special variables for thread identification in kernels

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

- Intrinsics that expose specific operations in kernel code

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

Example: Vector Addition Kernel

- Compute vector sum C = A+B means:
- n = length(C)
- for i = 0 to n-1:
  - C[i] = A[i] + B[i]
- So C[0] = A[0] + B[0], C[1] = A[1] + B[1], etc.

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

Host Code

```
int main()
{
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Synchronization of blocks

- Threads within block may synchronize with *barriers*

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

- Blocks *coordinate* via atomic memory operations
  - e.g., increment shared queue pointer with ***atomicInc()***
- Implicit barrier between *dependent kernels*

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

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory

  `cudaMalloc()`, `cudaFree()`

- Explicit memory copy for host ↔ device, device ↔ device

  `cudaMemcpy()`, `cudaMemcpy2D()`, …

- Texture management

  `cudaBindTexture()`, `cudaBindTextureToArray()`, …

- OpenGL & DirectX interoperability

  `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, …

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__ void vecAdd(float* A, float* B, float* C){
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}

int main(){
    // Run N/256 blocks of 256 threads each
    vecAdd<<< N/256, 256>>>(d_A, d_B, d_C);
}
```

Example: Host code for **vecAdd**

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for **vecAdd**

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for **vecAdd**

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Example: Host code for **vecAdd**

```
// allocate and initialize host (CPU) memory
float *h_A = …,   *h_B = …;
// allocate device (GPU) memory
float *d_A, *d_B, *d_C;

cudaMalloc( (void**) &d_A, N * sizeof(float));
cudaMalloc( (void**) &d_B, N * sizeof(float));
cudaMalloc( (void**) &d_C, N * sizeof(float));
// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

Basic Efficiency Rules

- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
- Exploit per-block shared memory as scratchpad
- Expose enough parallelism

Summing Up

- Four big ideas:

  One thread per data element

  Write one program, runs on all threads

  Hide latency by switching to different work

  Independent blocks allow scalability

- Three key abstractions:

  hierarchy of parallel threads

  corresponding levels of synchronization

  corresponding memory spaces

GPU Computing Challenges

- Addressing other dwarfs
- Sparseness & adaptivity
- Scalability: Multi-GPU algorithms and data structures
- Heterogeneity (Fusion/Knight's Corner architectures)
- Irregularity
- Incremental data structures
- Abstract models of GPU computation

Thanks to …

- David Luebke and Rich Vuduc for slides
- NVIDIA for hardware donations; Argonne and University of Illinois / NCSA for cluster access
- Funding agencies: Department of Energy (SciDAC Institute for Ultrascale Visualization, Early Career Principal Investigator Award), NSF, LANL, BMW, NVIDIA, HP, Intel, UC MICRO, Microsoft, ChevronTexaco, Rambus