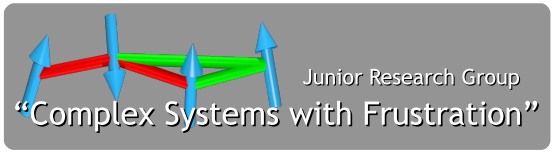
Simulating spin models on GPU
Lecture 1: GPU architecture and CUDA programming

Martin Weigel

Applied Mathematics Research Centre, Coventry University, Coventry, United Kingdom and
Institut für Physik, Johannes Gutenberg-Universität Mainz, Germany

IMPRS School 2012: GPU Computing,
Wroclaw, Poland, October 30, 2012



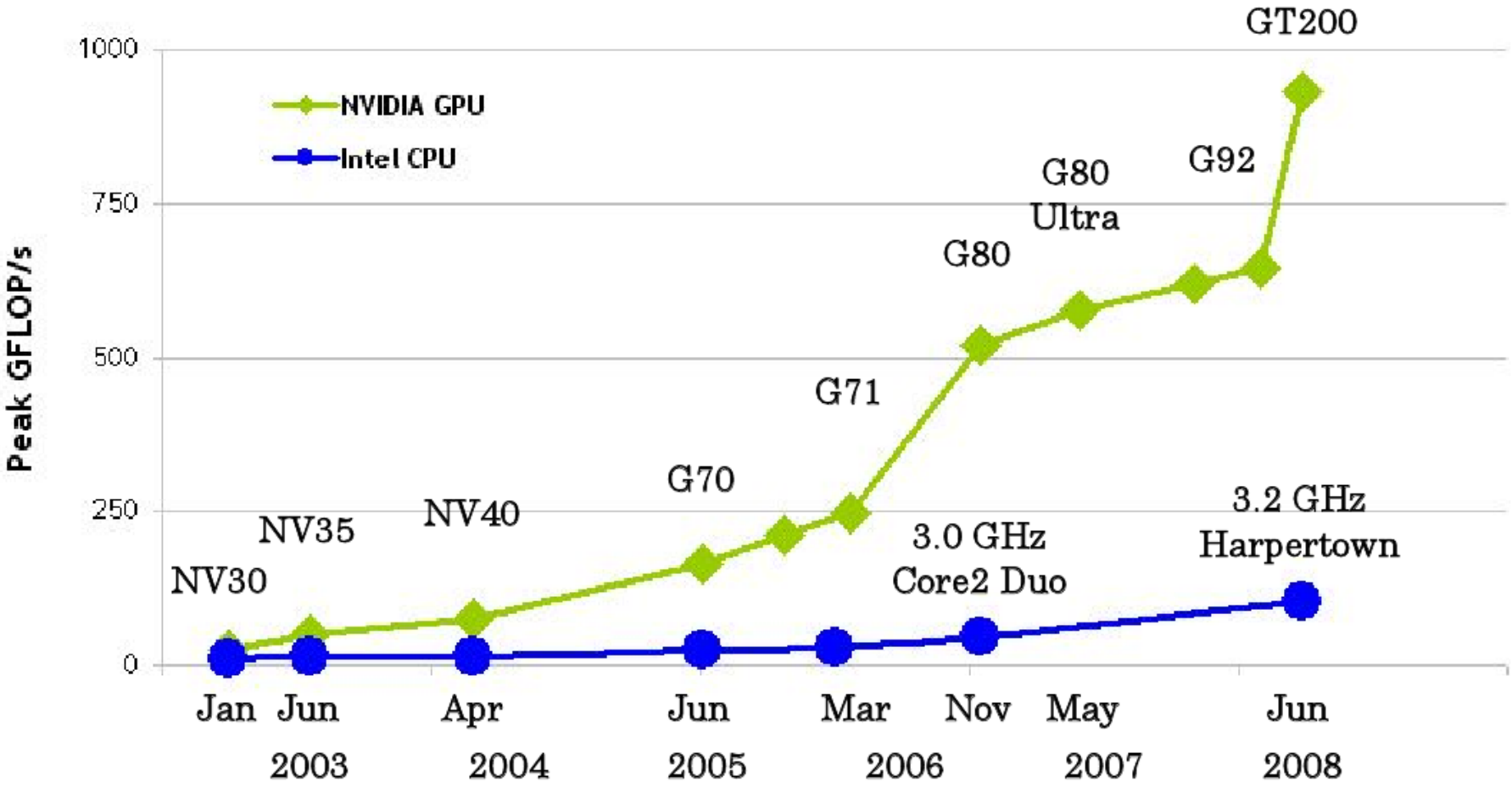
GPU computing

traditional interpretation of GPU computing

GPU computing

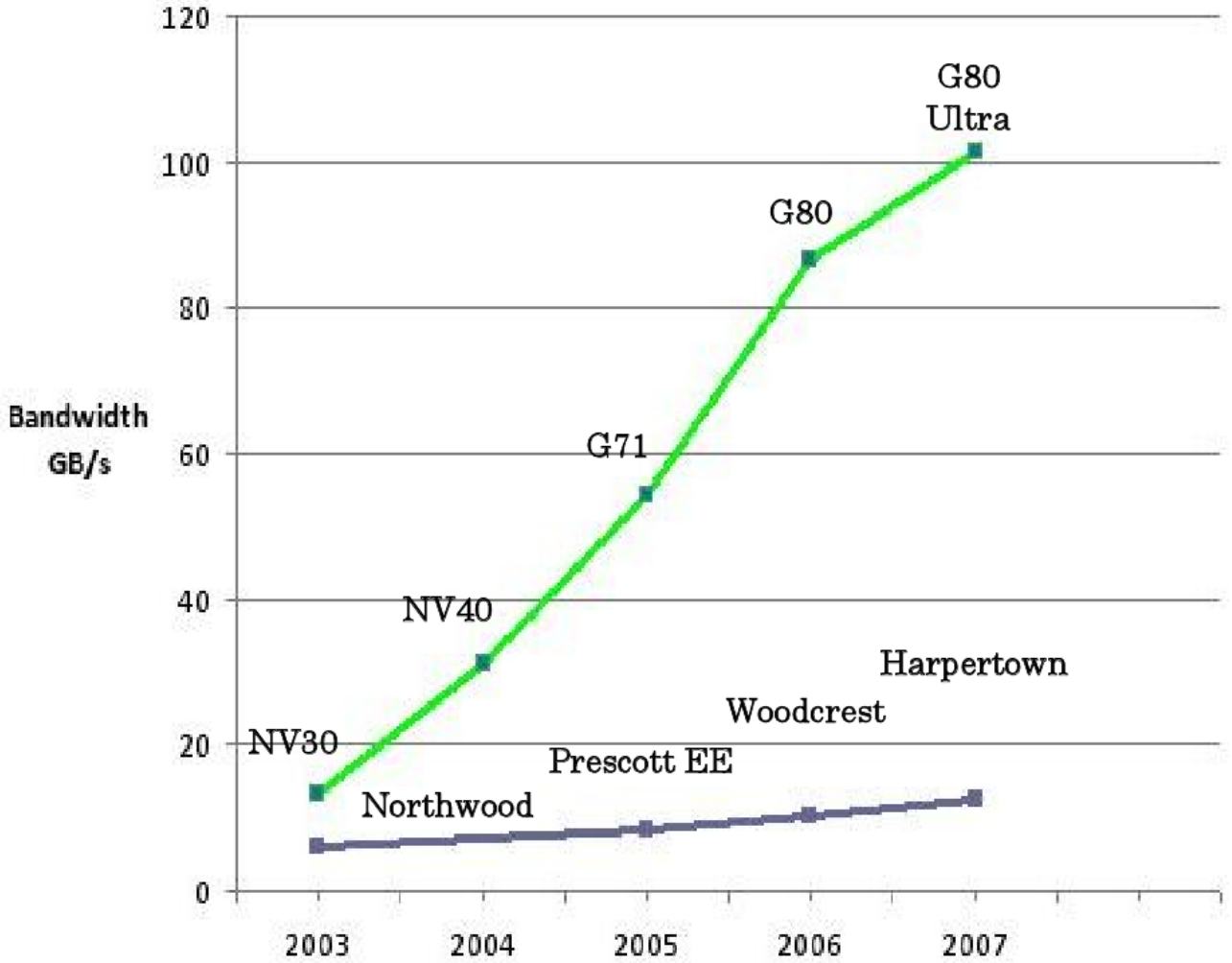
traditional interpretation of GPU computing

GPU computing



- Core i7 IvyBridge i7-3870: 122 GFLOP/s
- NVIDIA Tesla K10: 4580 GFLOP/s (single precision)

GPU computing



- Core i7 IvyBridge i7-3870: $\approx$ 21 GB/s
- NVIDIA Tesla K10: 320 GB/s

Outline

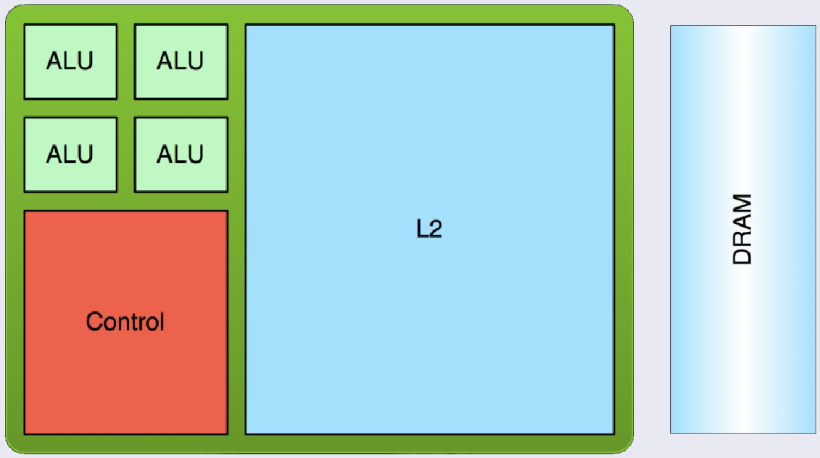
1. Latency vs. throughput
2. GPU architecture
3. Execution model
4. Memory hierarchy
5. CUDA Programming
6. A worked example
7. New in Kepler

Outline

1. Latency vs. throughput
2. GPU architecture
3. Execution model
4. Memory hierarchy
5. CUDA Programming
6. A worked example
7. New in Kepler

CPU vs. GPU hardware

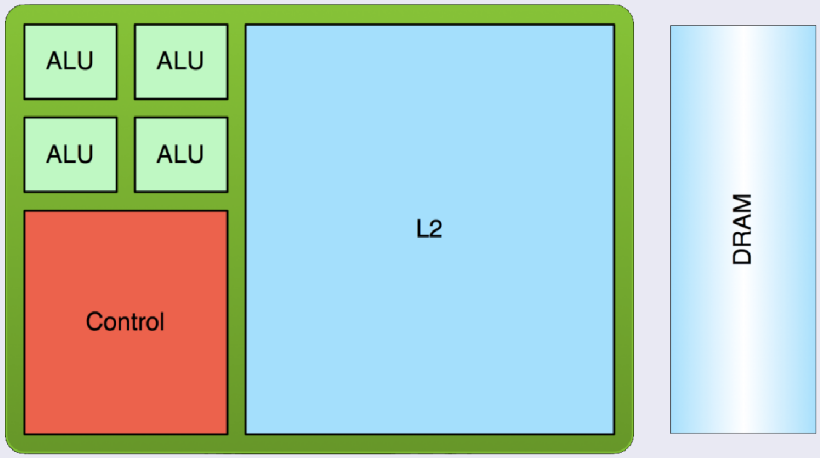
CPU



- optimized for low-latency access to cached data
- extensive logic for branch prediction and out-of-order execution
- do an unpredictable scalar job as fast as possible

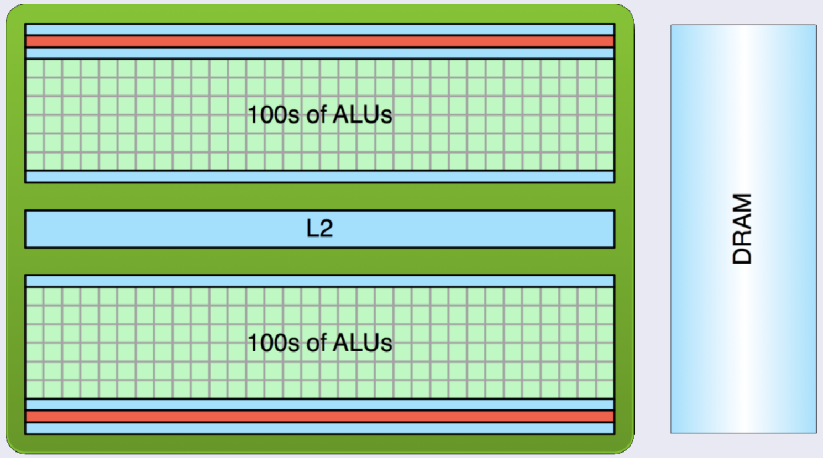
CPU vs. GPU hardware

CPU



- optimized for low-latency access to cached data
- extensive logic for branch prediction and out-of-order execution
- do an unpredictable scalar job as fast as possible

GPU



- optimized for data-parallel throughput computations
- latency hiding
- do as many simple, deterministic jobs in parallel as possible

Latency hiding

GPU threads

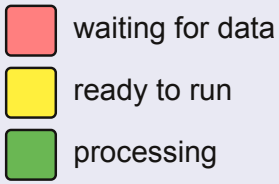
Legend

- waiting for data
- ready to run
- processing

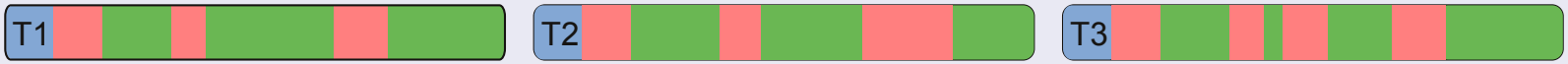
Latency hiding

GPU threads

Legend



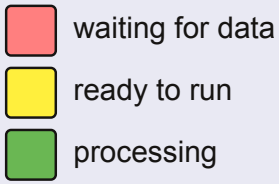
CPU threads



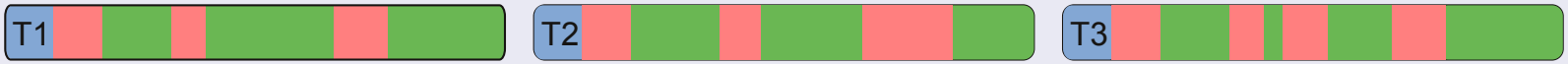
Latency hiding

GPU threads

Legend

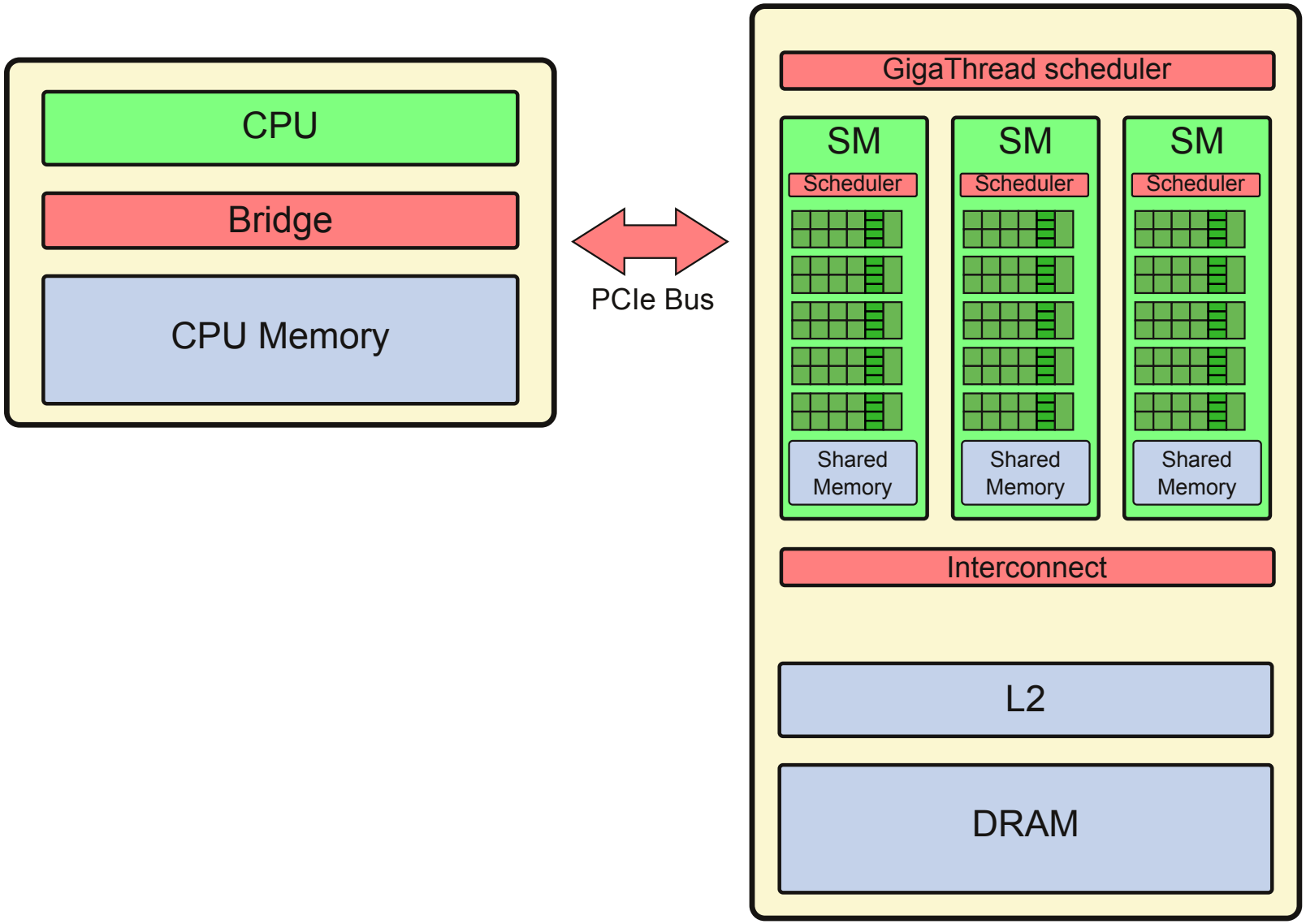


CPU threads

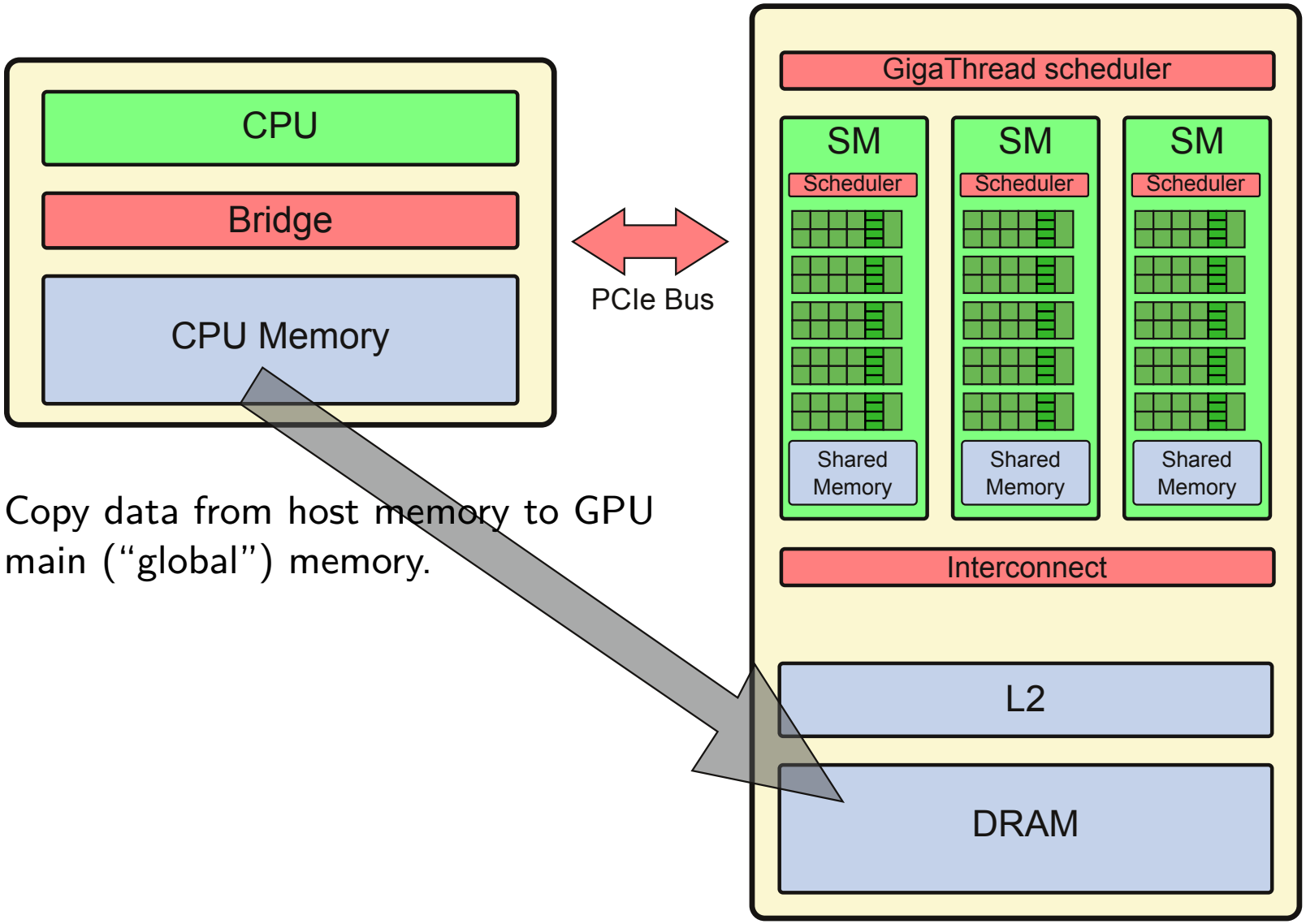


- CPU must minimize latency of individual thread for responsiveness
- GPU *hides* latency through interleaved execution

General processing flow

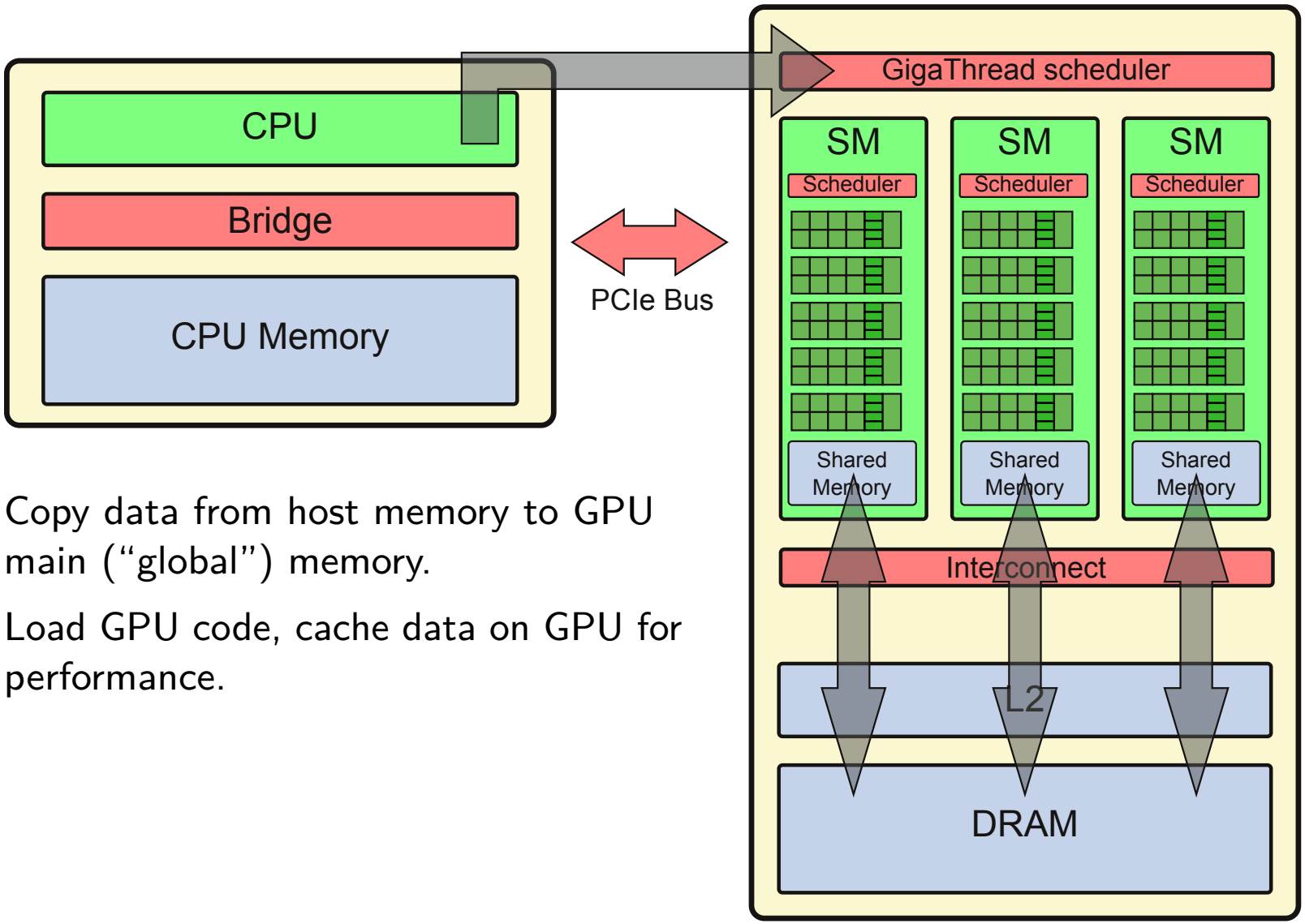


General processing flow



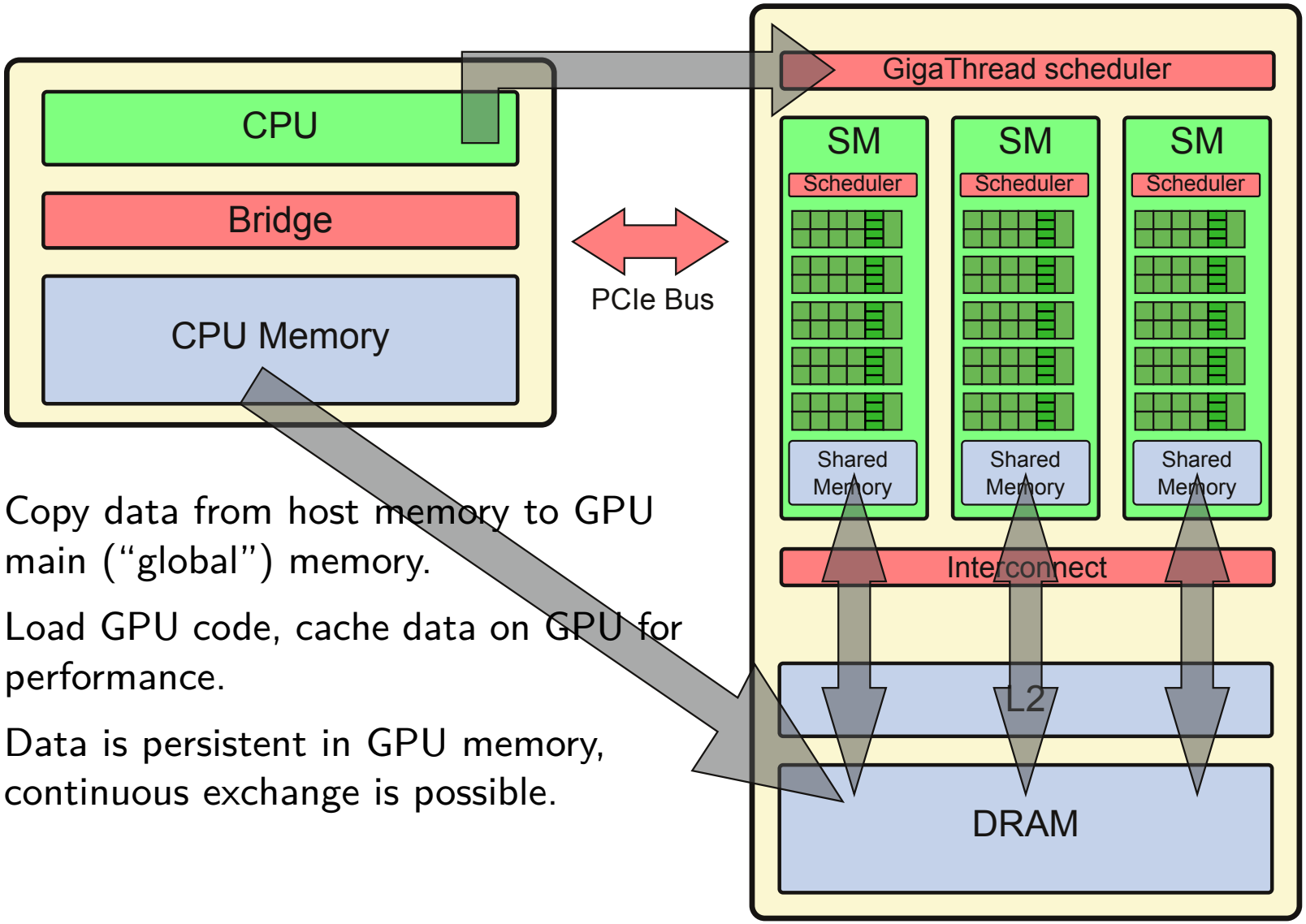
1. Copy data from host memory to GPU main ("global") memory.

General processing flow



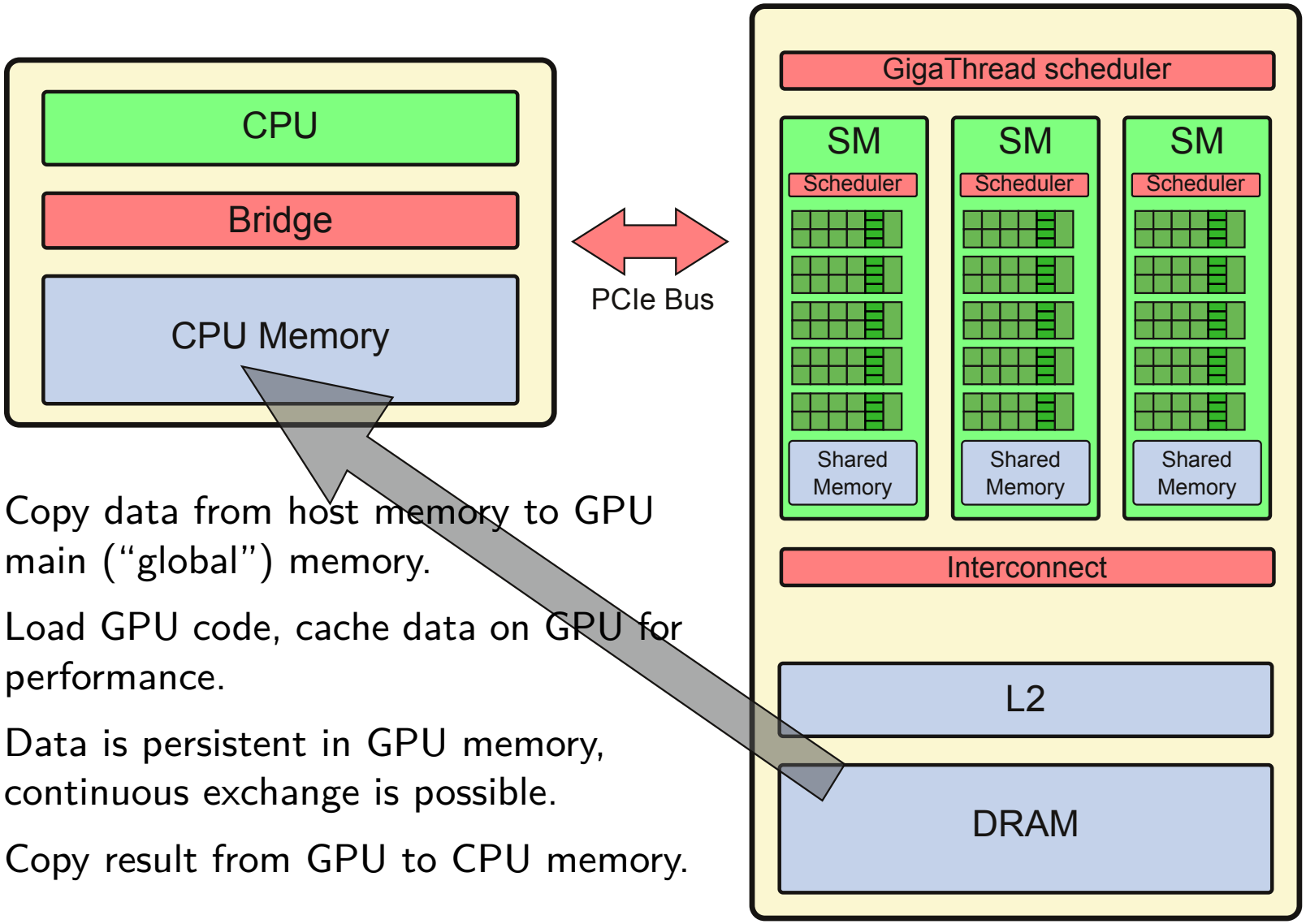
1. Copy data from host memory to GPU main ("global") memory.
2. Load GPU code, cache data on GPU for performance.

General processing flow



1. Copy data from host memory to GPU main ("global") memory.
2. Load GPU code, cache data on GPU for performance.
3. Data is persistent in GPU memory, continuous exchange is possible.

General processing flow



1. Copy data from host memory to GPU main ("global") memory.
2. Load GPU code, cache data on GPU for performance.
3. Data is persistent in GPU memory, continuous exchange is possible.
4. Copy result from GPU to CPU memory.

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

6 A worked example

7 New in Kepler

GPGPU history

NVIDIA

- introduced CUDA in 2007
- developed into a fully blown ecosystem
- series of computing cards
- academic support programs
  - CUDA professorships
  - CUDA research centers
  - CUDA teaching centers

GPGPU history

NVIDIA

- introduced CUDA in 2007
- developed into a fully blown ecosystem
- series of computing cards
- academic support programs
  - CUDA professorships
  - CUDA research centers
  - CUDA teaching centers

ATI

- ATI Stream introduced in 2007
- less viral marketing
- higher peak performance
- somewhat less flexible architecture

GPGPU history

NVIDIA

- introduced CUDA in 2007
- developed into a fully blown ecosystem
- series of computing cards
- academic support programs
  - CUDA professorships
  - CUDA research centers
  - CUDA teaching centers

ATI

- ATI Stream introduced in 2007
- less viral marketing
- higher peak performance
- somewhat less flexible architecture

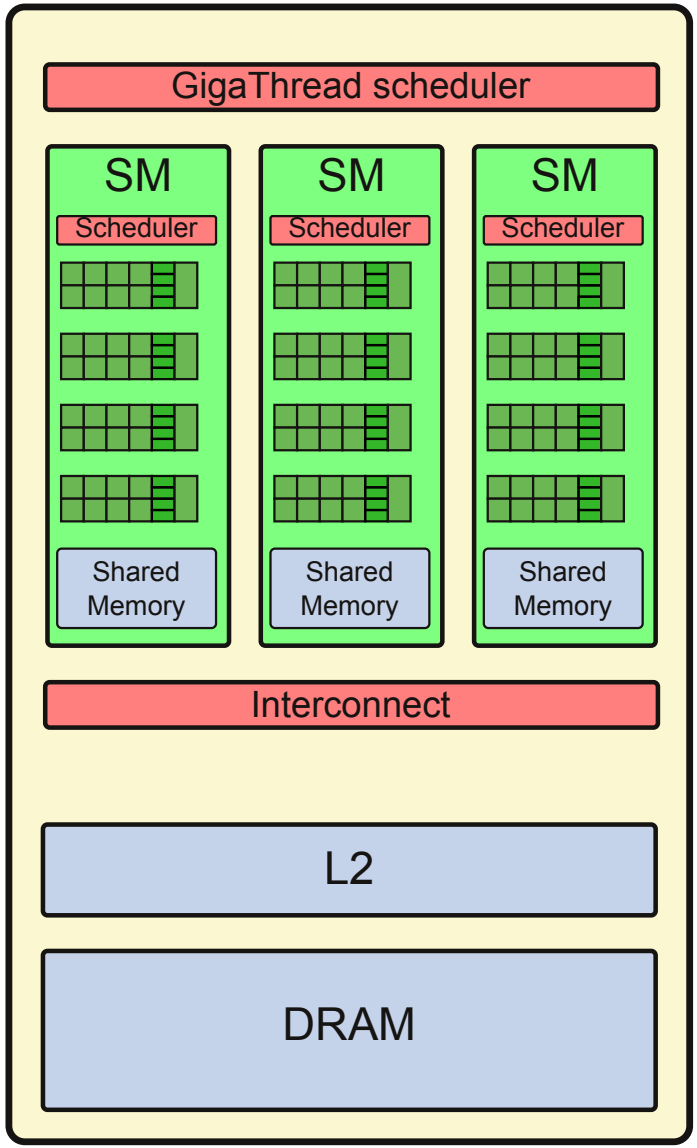
Other architectures

- Intel MIC
  - started as Larabee in 2006
  - prototype board Knight's Ferry (2010)
    - 32 cores with 4 threads/core
    - 2 GB DDR5 memory
  - Knight's Corner developed in connection with SGI in 2012
    - more than 50 cores per chip
    - supposedly used in a number of new supercomputers
- (some) special-purpose machines
  - ANTON by D. E. Shaw research, composed of purpose-built ASICs
  - Janus, FPGA based machine for spin-model simulations
  - QPace, based on the Cell processor known from Playstation 3
  - GRAPE, based on FPGAs and used for astrophysical N-body simyulations

GPU architecture: main components

Main memory

- up to 8GB in current GPUs
- maximum bandwidth approx. 160 GB/s (Fermi) resp. 320 GB/s (Kepler)
- accessible from CPU and GPU sides
- large latency (see below)
- optional error correction (ECC on/off, Fermi onwards)



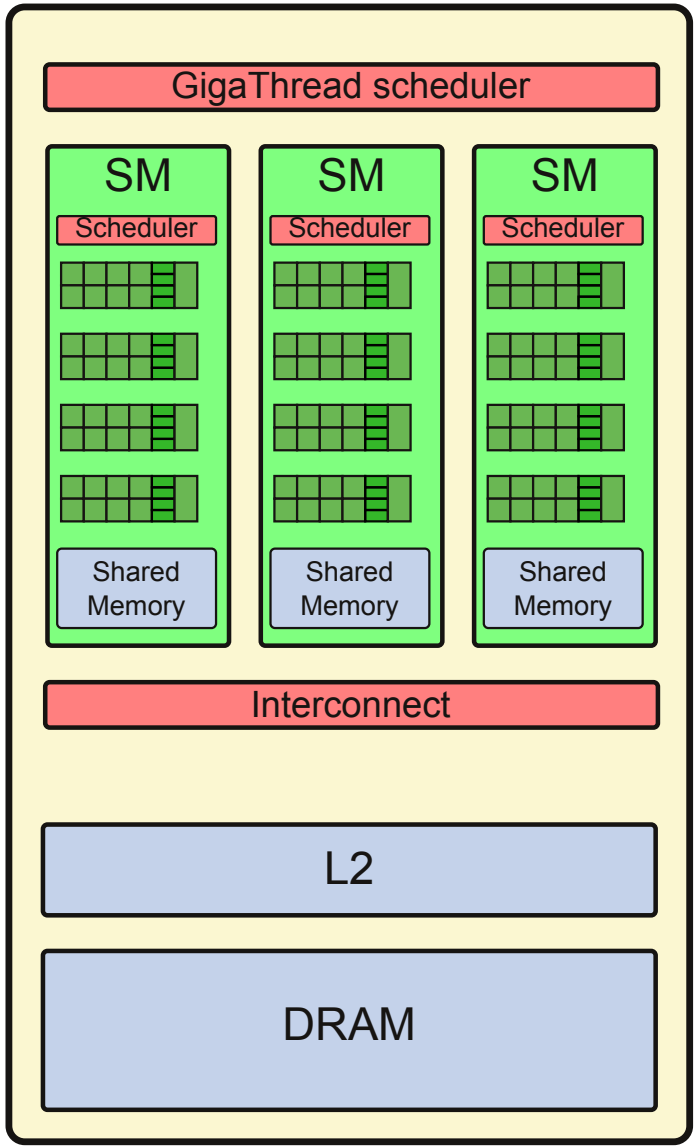
GPU architecture: main components

Main memory

- up to 8GB in current GPUs
- maximum bandwidth approx. 160 GB/s (Fermi) resp. 320 GB/s (Kepler)
- accessible from CPU and GPU sides
- large latency (see below)
- optional error correction (ECC on/off, Fermi onwards)

Several multiprocessors

- similar to a multi-core CPU
- each has its own set of registers, scheduler, caches etc.



GPU architecture: main components

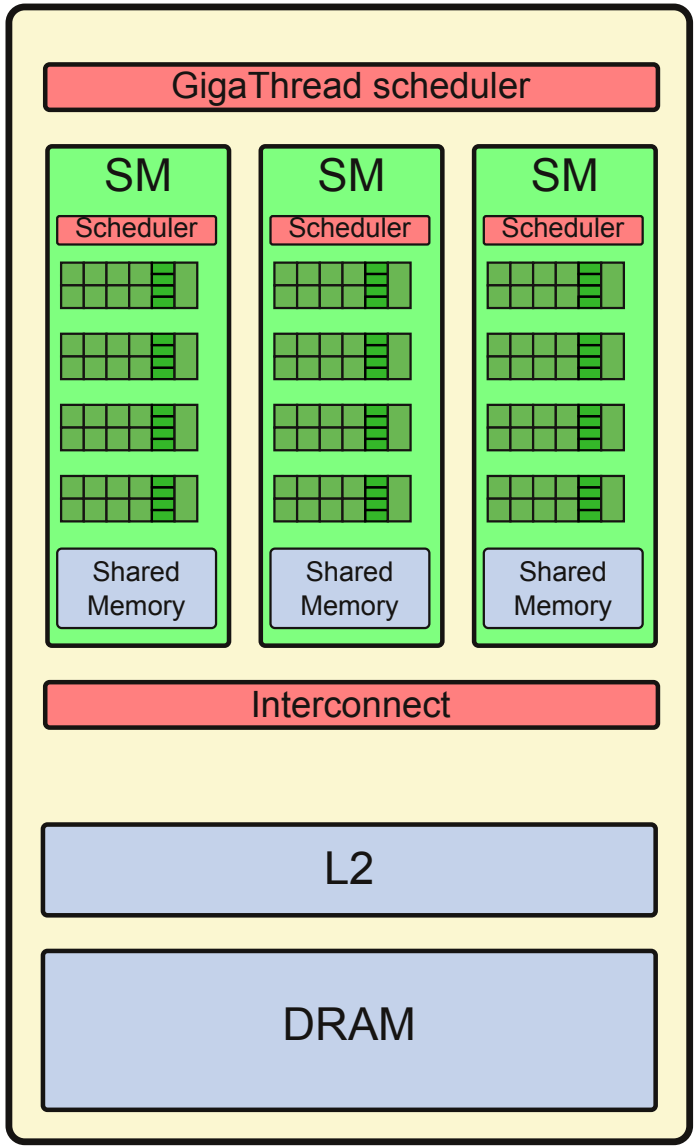
Main memory

- up to 8GB in current GPUs
- maximum bandwidth approx. 160 GB/s (Fermi) resp. 320 GB/s (Kepler)
- accessible from CPU and GPU sides
- large latency (see below)
- optional error correction (ECC on/off, Fermi onwards)

Several multiprocessors

- similar to a multi-core CPU
- each has its own set of registers, scheduler, caches etc.

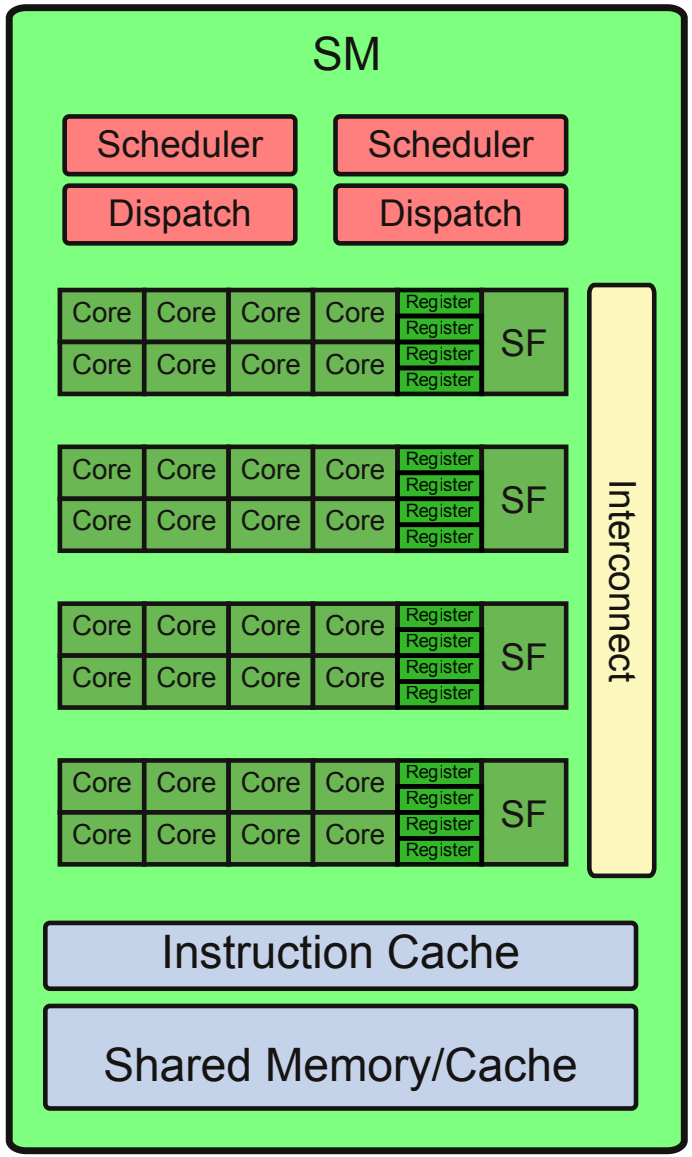
+ scheduling units, PCIe logic etc.



GPU architecture: streaming multiprocessor (Fermi)

SM components

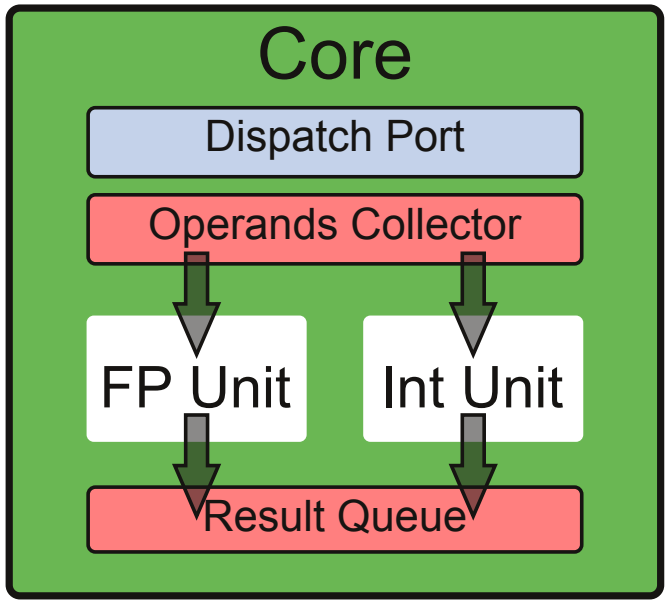
- 32 cores per SM (Fermi)
  - 32 fp32 ops/s
  - 16 fp64 ops/s (Tesla)
  - 32 int32 ops/s
- 2 warp schedulers (warp = 32 threads)
  - up to 1536 threads resident in total
- extra special function units (4 for Fermi)
- 64 KB cache on die, re-configurable as 16 KB cache + 32 KB shared memory or vice versa
- 32K 32-bit registers



GPU architecture: computing core

Computing core

- Integer and floating-point units:
  - IEEE-2008 compliant floating point arithmetic (starting from Fermi)
  - Fused multiply-and-add instruction in hardware
- Logic unit
- Move, compare unit
- Branch unit



NVIDIA GTX 480

- 480 streaming processor cores
- 1.4 GHz clock frequency
- single precision peak performance 1.3 TFLOP/s
- 1.5 GB GDDR5 RAM
- memory bandwidth 178 GB/s

GPU computation frameworks

GPGPU = General Purpose Computation on Graphics Processing Unit

"Old" times: use original graphics primitives

- OpenGL
- DirectX

Vendor specific APIs for GPGPU:

- NVIDIA CUDA: library of functions performing computations on GPU (C, C++, Fortran), additional preprocessor with language extensions
- ATI/AMD Stream: similar functionality for ATI GPUs

Device independent schemes:

- BrookGPU (Standford University): compiler for the "Brook stream program language" with backends for different hardware; now merged with AMD Stream
- Sh (University of Waterloo): metaprogramming language for programmable GPUs
- OpenCL (Open Computing Language): open framework for parallel programming across a wide range of devices, ranging from CPUs, Cell processors and GPUs to handheld devices

GPU computation frameworks

GPGPU = General Purpose Computation on Graphics Processing Unit

"Old" times: use original graphics primitives

- OpenGL
- DirectX

Vendor specific APIs for GPGPU:

- NVIDIA CUDA: library of functions performing computations on GPU (C, C++, Fortran), additional preprocessor with language extensions
- ATI/AMD Stream: similar functionality for ATI GPUs

Device independent schemes:

- BrookGPU (Standford University): compiler for the "Brook stream program language" with backends for different hardware; now merged with AMD Stream
- Sh (University of Waterloo): metaprogramming language for programmable GPUs
- OpenCL (Open Computing Language): open framework for parallel programming across a wide range of devices, ranging from CPUs, Cell processors and GPUs to handheld devices

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

6 A worked example

7 New in Kepler

Definitions

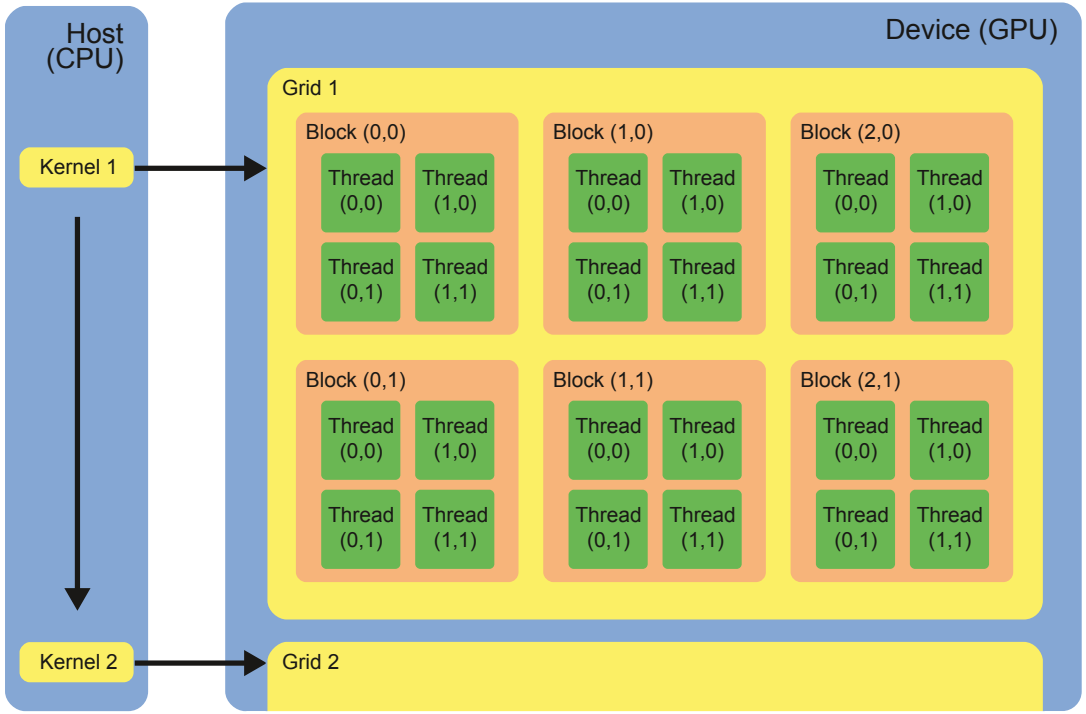
Kernel GPU program that runs on a grid of threads

Thread scalar execution unit

Warp block of 32 threads executed in lockstep

Block a set of warps executed on the same SM

Grid a set of blocks usually executed on different SMs



Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device

Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
  - CUDA threads are lightweight
    - very little creation overhead
    - low cost of thread switching
  - CUDA needs thousands of threads for efficiency

Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
  - CUDA threads are lightweight
    - very little creation overhead
    - low cost of thread switching
  - CUDA needs thousands of threads for efficiency
- Kernels are executed by an array of threads
  - all threads run the same code
  - each thread has a unique `threadid` for control decisions and memory access

Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
  - CUDA threads are lightweight
    - very little creation overhead
    - low cost of thread switching
  - CUDA needs thousands of threads for efficiency
- Kernels are executed by an array of threads
  - all threads run the same code
  - each thread has a unique `threadid` for control decisions and memory access
- Kernel launched are in grid of thread blocks
  - threads within block cooperate via shared memory
  - threads within a block can synchronize
  - threads within different blocks cannot cooperate (or only via global memory)
  - block executes on one SM and does not migrate

Threading hierarchy

- Parallel portions of an application are executed on GPU as kernels.
  - one kernel is executed at a time (later on modified in concept of streams)
  - each kernel executes in many threads, but on one device
- Compare CUDA threads to CPU threads
  - CUDA threads are lightweight
    - very little creation overhead
    - low cost of thread switching
  - CUDA needs thousands of threads for efficiency
- Kernels are executed by an array of threads
  - all threads run the same code
  - each thread has a unique `threadid` for control decisions and memory access
- Kernel launched are in grid of thread blocks
  - threads within block cooperate via shared memory
  - threads within a block can synchronize
  - threads within different blocks cannot cooperate (or only via global memory)
  - block executes on one SM and does not migrate
- allows programs to transparently scale to GPUs with different numbers of cores

CUDA code: C with some extra reserved words

Consider a simple "SAXPY" computation, i.e., "Single-Precision $A \cdot X + Y$".

Standard C code

```
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 serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

CUDA C code

```
__global__ void saxpy_parallel(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_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

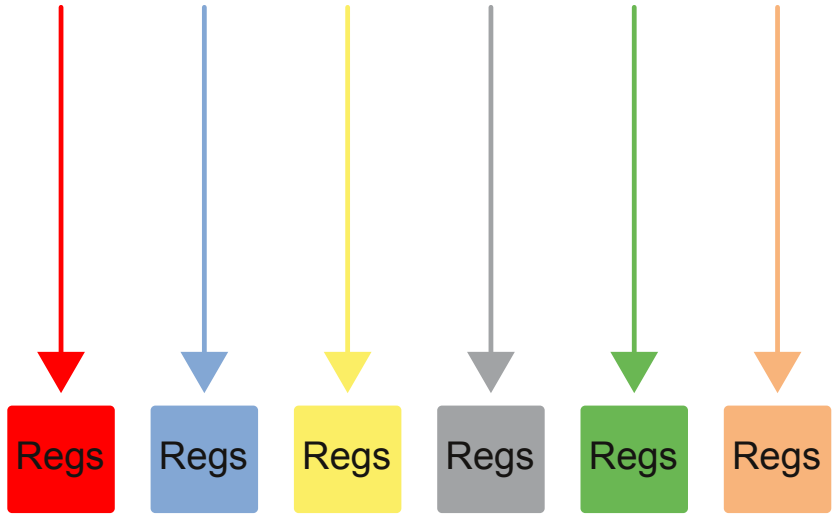
6 A worked example

7 New in Kepler

Memory hierarchy

Per thread

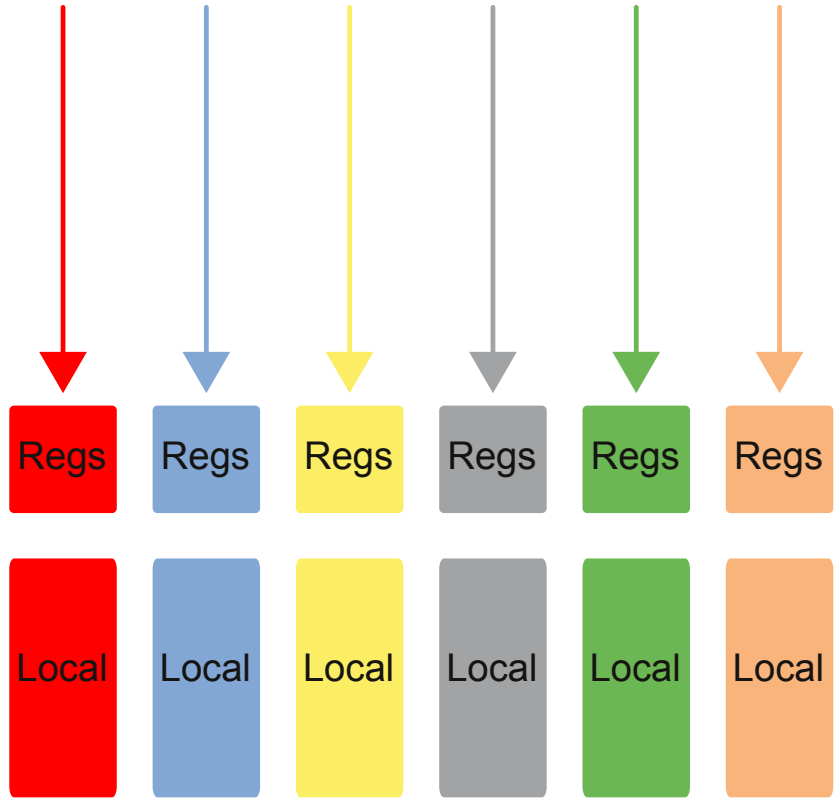
- Registers (extra fast, no copy for ops)



Memory hierarchy

Per thread

- Registers (extra fast, no copy for ops)
- Local memory



Memory hierarchy

Per thread

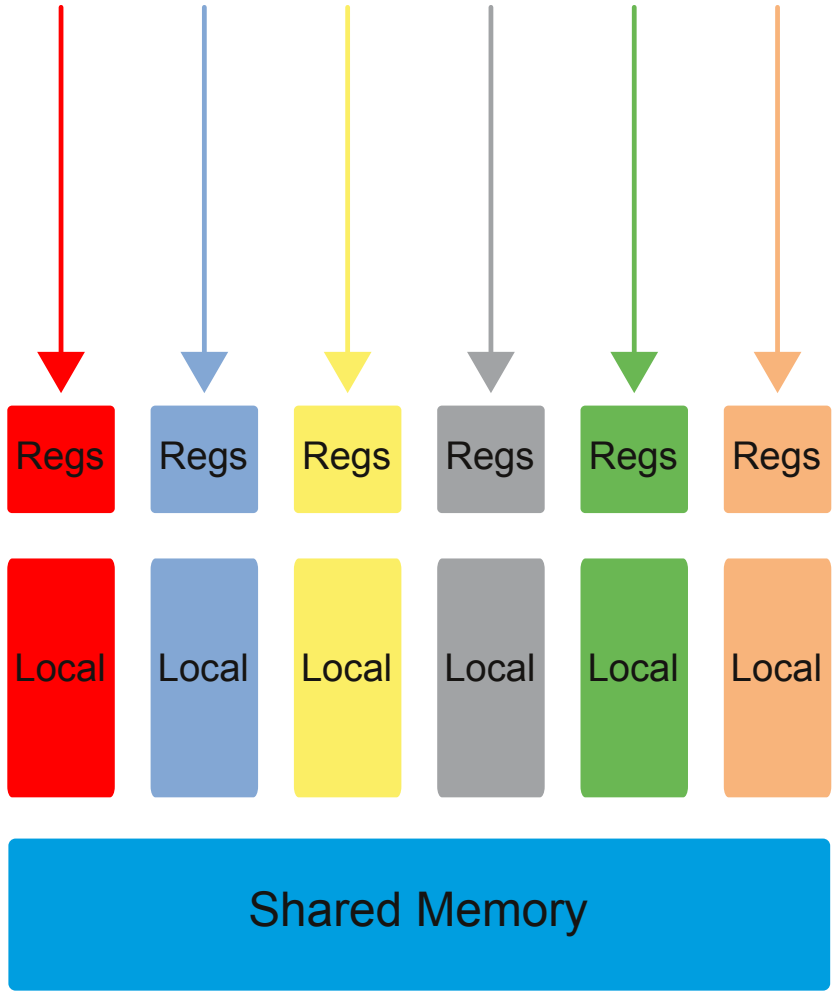
- Registers (extra fast, no copy for ops)
- Local memory

Thread blocks: shared memory

- allocated by thread block, same lifetime as block
- allocate as

```
__shared__ int shared_array[
    DIM];
```

- low latency (of the order of 10 cycles), bandwidth up to 1 TB/s
- use for data sharing and user-managed cache



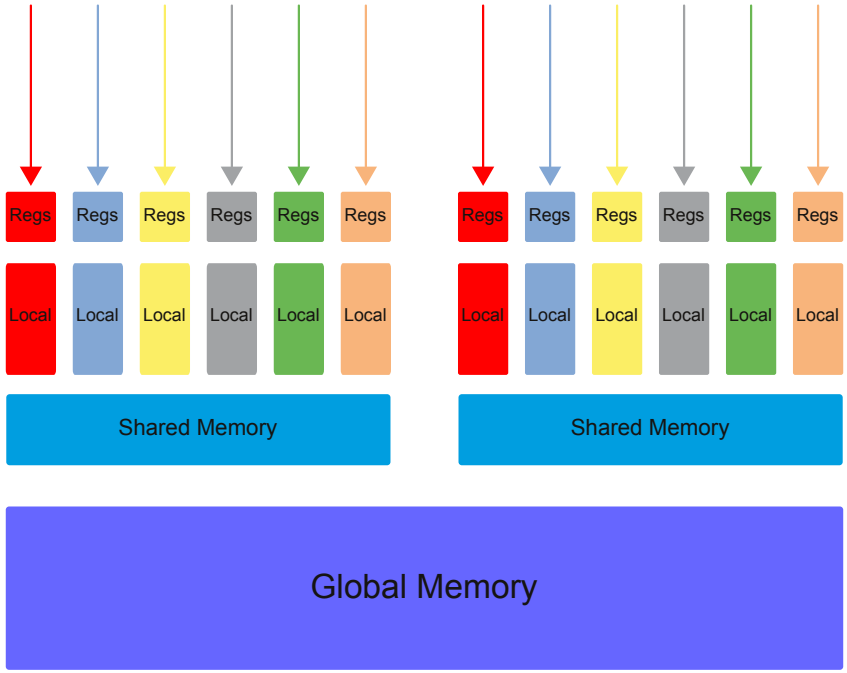
Memory hierarchy

Per device: global memory

- accessible to all threads on device
- lifetime is user-defined

```
cuda_malloc(void **pointer,
    size_t nbytes);
cuda_free(void* pointer);
```

- latency several hundred clock cycles
- bandwidth ≈ 160 GB/s on Fermi
  (access pattern needs to conform to coalescence rules for good performance)



Memory hierarchy

Per device: global memory

- accessible to all threads on device
- lifetime is user-defined

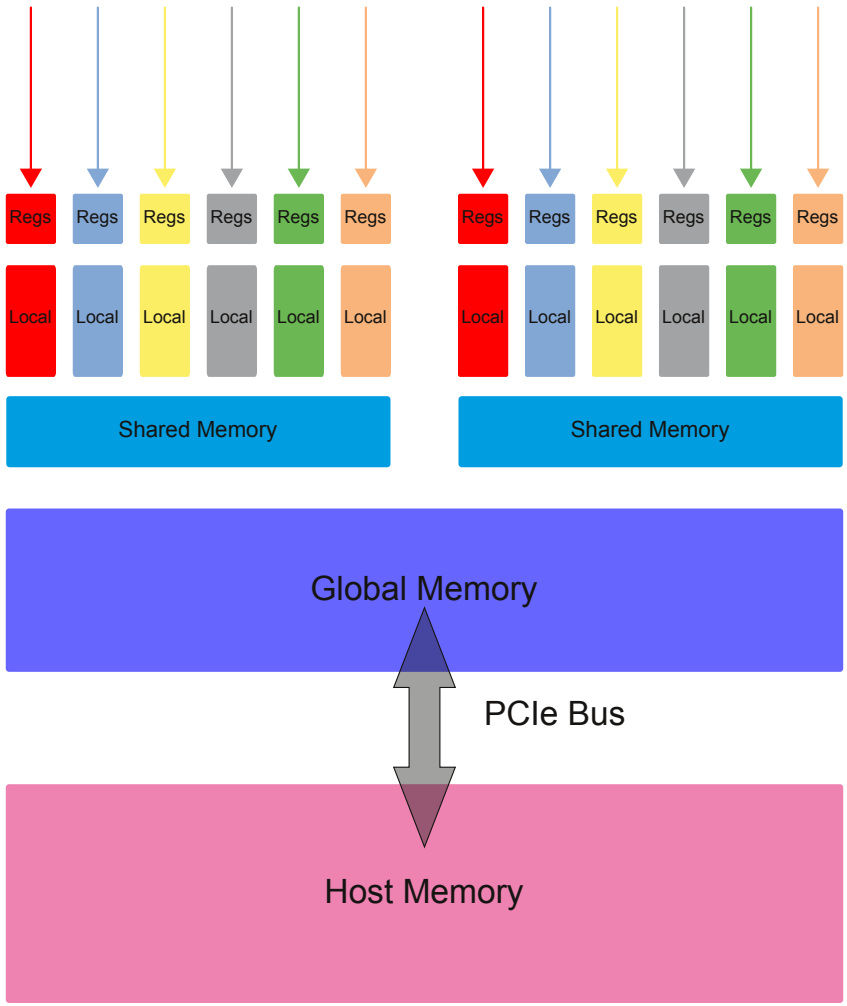
```
cuda_malloc(void **pointer,
    size_t nbytes);
cuda_free(void* pointer);
```

- latency several hundred clock cycles
- bandwidth ≈ 160 GB/s on Fermi
  (access pattern needs to conform to coalescence rules for good performance)

Per host: device memory

- no direct access from CUDA threads
- copy data to/from device with

```
cudaMemcpy(void* dest, void*
    src, size_t nbytes,
    cudaMemcpyHostToDevice);
```

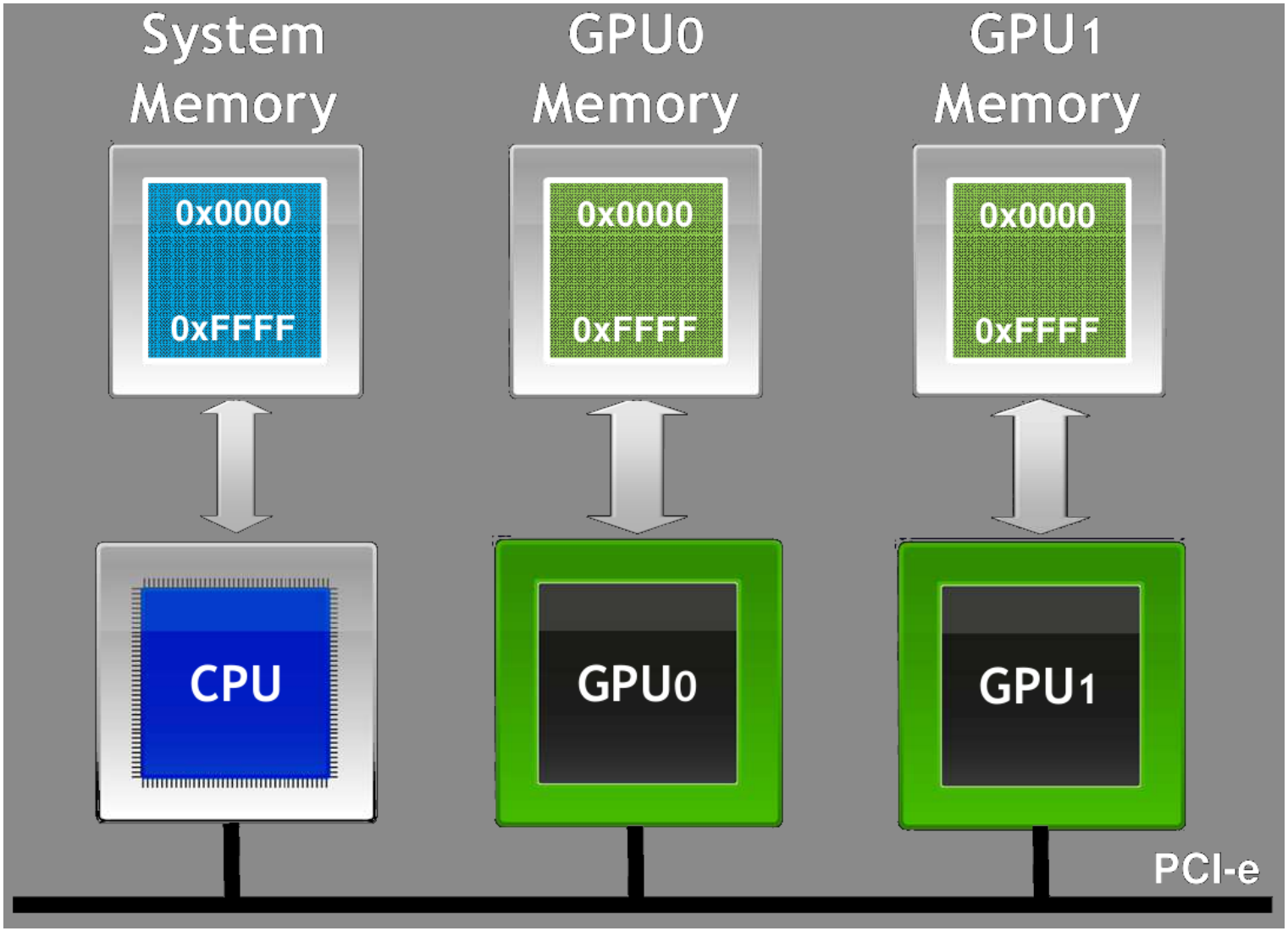


Memory hierarchy (summary)

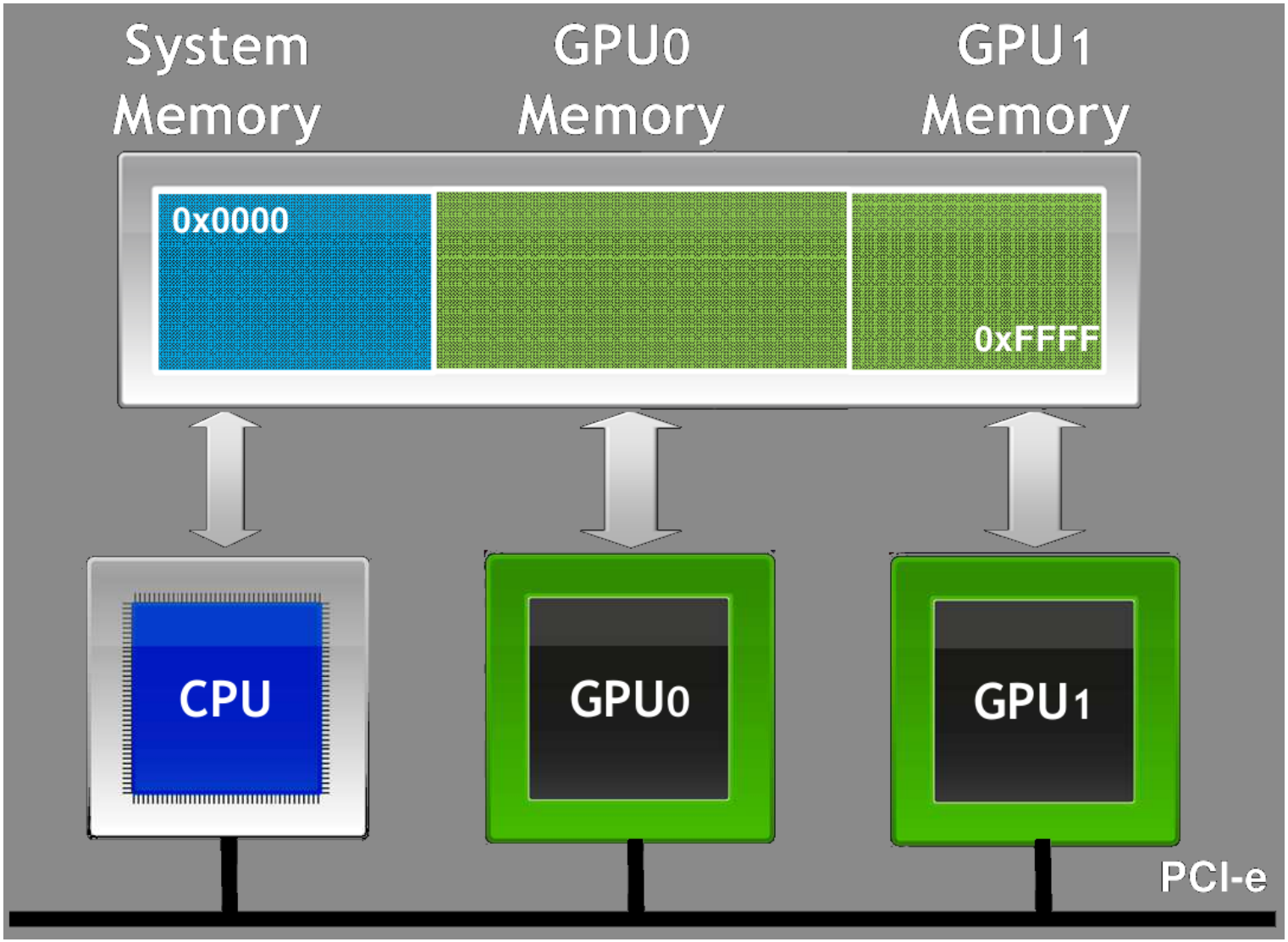
More generally, the different types of memory have the following characteristics:

Memory	**Location**	**Cached**	**Access**	**Scope**	**Lifetime**
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	(Yes)	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application

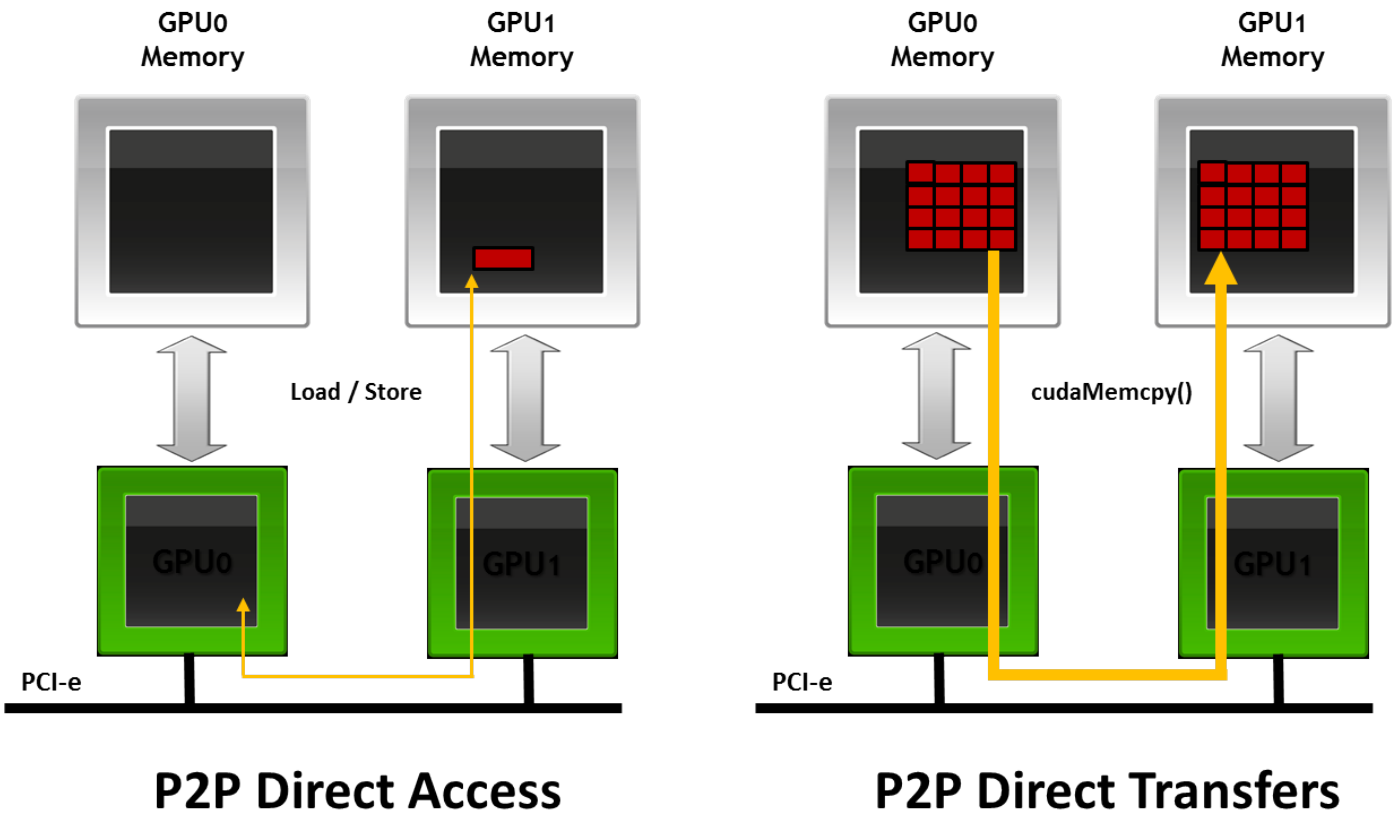
Unified virtual addressing



Unified virtual addressing



DirectGPU data transfers



- GPUDirect 1.0: direct GPU memory access by devices such as network adaptors
- GPUDirect 2.0: direct copies from GPU to GPU inside a node
- GPUDirect/CUDA 5.0: direct communication between GPUs in different nodes
- MPI integration under development

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

6 A worked example

7 New in Kepler

CUDA variables

Variable declaration	Memory	Scope	Lifetime	Penalty/Latency
`int var;`	register	thread	thread	1X
`int array_var[10];`	local	thread	thread	100X (pre-Fermi)
`__shared__ int shared_var;`	shared	block	block	10X
`__device__ int global_var;`	global	grid	application	100X
`__constant__ int constant_var;`	constant	grid	application	1X

CUDA variables

Variable declaration	Memory	Scope	Lifetime	Penalty/Latency
`int var;`	register	thread	thread	1X
`int array_var[10];`	local	thread	thread	100X (pre-Fermi)
`__shared__ int shared_var;`	shared	block	block	10X
`__device__ int global_var;`	global	grid	application	100X
`__constant__ int constant_var;`	constant	grid	application	1X

- automatic scalar variables reside in registers, compiler will spill into local memory in shortage of registers
- automatic array variables (in the absence of qualifiers) reside in thread-local memory
- the type of memory used will be crucial for the performance of the application

Elementary data transfers with CUDA

Memory allocation

Arrays in device global memory are typically allocated from CPU code. Functions:

- `cudaMalloc(void ** pointer, size_t nbytes);`
- `cudaMemset(void * pointer, int value, size_t count);`
- `cudaFree(void* pointer);`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Elementary data transfers with CUDA

Memory allocation

Arrays in device global memory are typically allocated from CPU code. Functions:

- `cudaMalloc(void ** pointer, size_t nbytes);`
- `cudaMemset(void * pointer, int value, size_t count);`
- `cudaFree(void* pointer);`

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Data transfers

The elementary function for data transfers is

- `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`
  - `direction` is one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost` or `cudaMemcpyDeviceToDevice` and specifies location of `src` and `dst`
  - blocks CPU thread (asynchronous transfers possible in streams)

Kernel execution

Function qualifiers

```
__global__ void f()
```

- function called from host, executed on device
- must return void

Kernel execution

Function qualifiers

```
__global__ void f()
```

- function called from host, executed on device
- must return void

```
__device__ int f()
```

- function called from device, executed on device

Kernel execution

Function qualifiers

```
__global__ void f()
```

- function called from host, executed on device
- must return void

```
__device__ int f()
```

- function called from device, executed on device

```
__host__ int f()
```

- function called from host, executed on host
- `__host__` and `__device__` can be combined to generate CPU and GPU code

Kernel execution

Function qualifiers

```
__global__ void f()
```

- function called from host, executed on device
- must return void

```
__device__ int f()
```

- function called from device, executed on device

```
__host__ int f()
```

- function called from host, executed on host
- `__host__` and `__device__` can be combined to generate CPU and GPU code

Built-in variables

All `__global__` and `__device__` functions have the following automatic variables:

- `dim3 gridDim;` — dimension of the grid in blocks
- `dim3 blockDim;` — dimension of the block in threads
- `dim3 blockIdx;` — block index within grid
- `dim3 threadIdx;` — thread index within block

The indices can be used to construct a global thread index, for instance for a block size of 5 threads,

```
thread_index = blockIdx.x*blockDim.
    x + threadIdx.x;
```

Kernel execution

Execution configuration

Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...);
```

Execution configuration:

- `dG`: dimension and size of grid in blocks
  - two-dimensional, `dG.x` and `dG.y`
  - total number of blocks launched is `dG.x` × `dG.y`
- `dB`: dimension and size of each block
  - two- or three-dimensional, `dB.x`, `dB.y`, and `dB.z`
  - total number of threads per block is `dB.x` × `dB.y` × `dB.z`
  - if not specified, `dB.z = 1` is assumed

Built-in variables

All `__global__` and `__device__` functions have the following automatic variables:

- `dim3 gridDim;` — dimension of the grid in blocks
- `dim3 blockDim;` — dimension of the block in threads
- `dim3 blockIdx;` — block index within grid
- `dim3 threadIdx;` — thread index within block

The indices can be used to construct a global thread index, for instance for a block size of 5 threads,

```
thread_index = blockIdx.x*blockDim.
    x + threadIdx.x;
```

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

6 A worked example

7 New in Kepler

The Julia set

Definition

Let $f(z) = p(z)/q(z)$ be a complex function, where $p(z)$ and $q(z)$ are complex polynomials.

The Julia set

Definition

Let $f(z) = p(z)/q(z)$ be a complex function, where $p(z)$ and $q(z)$ are complex polynomials.

The Julia set of f can be described as the set of points for which

$$\lim_{n\to\infty} |f^{(n)}(z)| < \infty,$$

where $f^{(n)}(z)$ denotes the n-fold repeated application of f on z.

The Julia set

Definition

Let $f(z) = p(z)/q(z)$ be a complex function, where $p(z)$ and $q(z)$ are complex polynomials.

The Julia set of f can be described as the set of points for which

$$\lim_{n\to\infty} |f^{(n)}(z)| < \infty,$$

where $f^{(n)}(z)$ denotes the n-fold repeated application of f on z.

In general, the Julia set is a self-similar fractal. Standard example:

$$f(x) = z^2 + c,$$

where c is a complex constant.

The Julia set

Definition

Let $f(z) = p(z)/q(z)$ be a complex function, where $p(z)$ and $q(z)$ are complex polynomials.

The Julia set of f can be described as the set of points for which

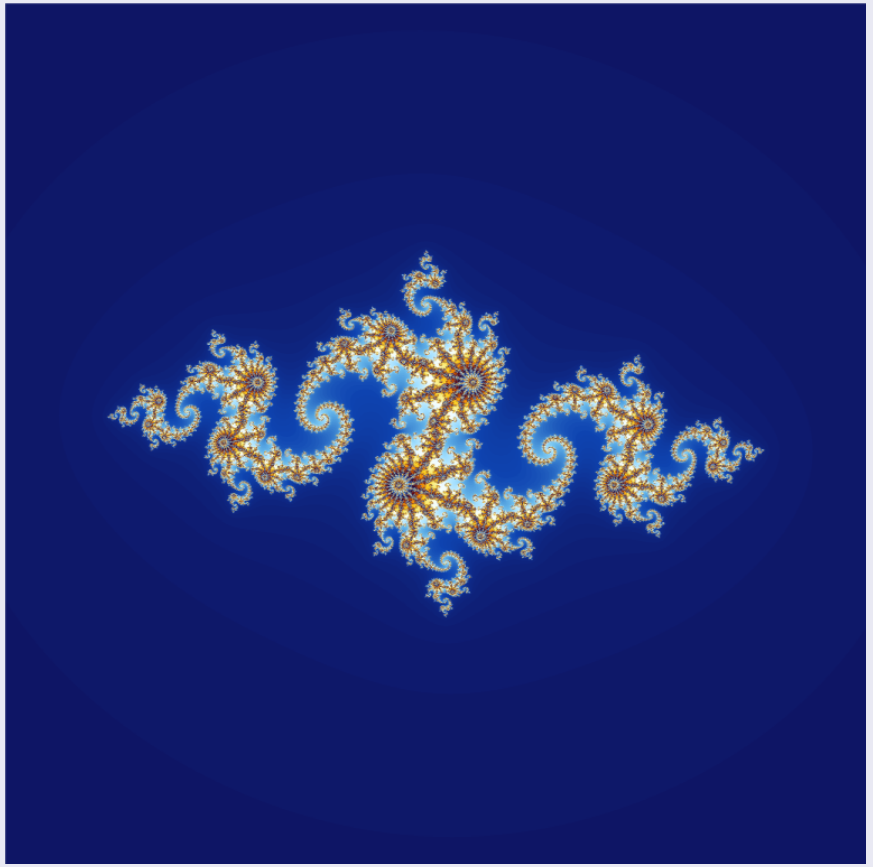
$$\lim_{n\to\infty} |f^{(n)}(z)| < \infty,$$

where $f^{(n)}(z)$ denotes the n-fold repeated application of f on z.

In general, the Julia set is a self-similar fractal. Standard example:

$$f(x) = z^2 + c,$$

where c is a complex constant.



(Color codes are for different rates of divergence.)

Julia: CPU code 1

Driver

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

- `CPUbitmap` externally defined
- `display_and_exit()` externally defined

Julia: CPU code 1

Driver

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

Kernel

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
 }
```

- `CPUbitmap` externally defined
- `display_and_exit()` externally defined

- `julia()` will return 1 if point in set
- choose red and black colors, respectively

Julia: CPU code 2

Julia set function

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Julia: CPU code 2

Julia set function

```
int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

- shift center of image to $(0, 0)$
- re-scale to unit square in complex plane
- use `scale` factor to zoom
- arbitrary constant c in $z_{n+1} = z_n^2 + c$

Julia: CPU code 3

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

Julia: CPU code 3

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    cuComplex( float a, float b ) : r(a), i(b)  {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

- use `cuComplex` structure to represent complex numbers
- operator overloading for natural semantics

Julia: GPU code 1

Driver

```
int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char     *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
    data.dev_bitmap = dev_bitmap;

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    bitmap.display_and_exit();
}
```

Julia: GPU code 1

Driver

```
int main( void ) {
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char     *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );
    data.dev_bitmap = dev_bitmap;

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );

    bitmap.display_and_exit();
}
```

- use `cudaMalloc` to allocate memory on device
- assign one thread per pixel, one thread per block, resulting in a `DIM` $\times$ `DIM` grid
- third dimension in `dim3` defaults to one $\Rightarrow$ 2D grid

Julia: GPU code 2

Kernel

```
__global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

Julia: GPU code 2

Kernel

```
__global__ void kernel( unsigned char *ptr ) {
    // map from blockIdx to pixel position
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // now calculate the value at that position
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

- `__global__` qualifier for device function to be called from host
- each thread gets `threadIdx.x`, `threadIdx.y`, `blockIdx.x`, and `blockIdx.y`
- translate to `offset` in image array
- note that `for` loops have gone away
- four chars to represent R, G, B and alpha channels

Julia: GPU code 3

Julia function

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

Julia: GPU code 3

Julia function

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

- `__device__` qualifier for device function to be called from device code
- identical to CPU code apart from function qualifier

Julia: GPU code 4

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    __device__ cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

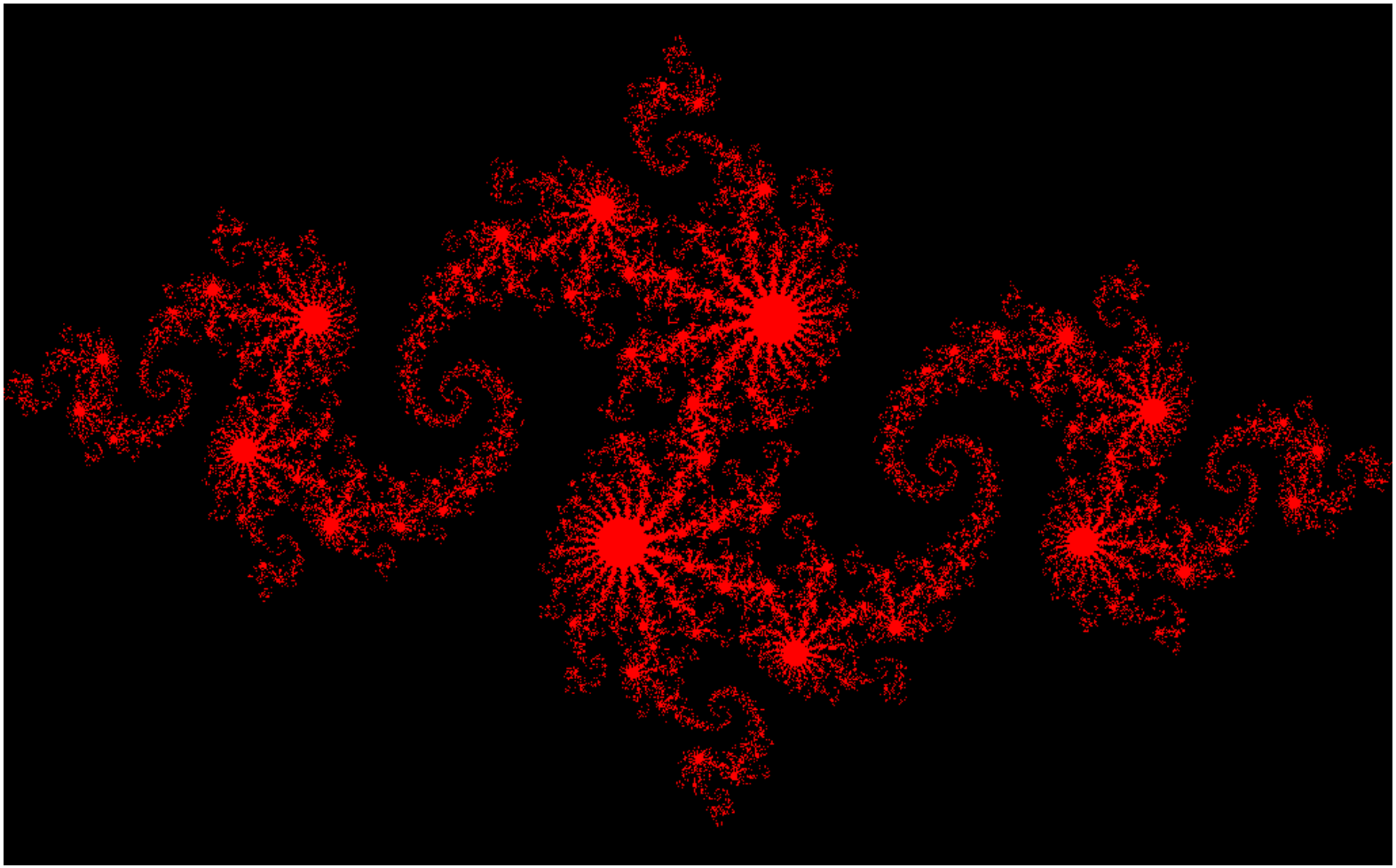
Julia: GPU code 4

Data structure

```
struct cuComplex {
    float   r;
    float   i;
    __device__ cuComplex( float a, float b ) : r(a), i(b)  {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};
```

- almost identical to CPU version
- only difference are `__device__` function qualifiers

Julia: result



Play around with code, cf.

J. Sanders, E. Kandrot: "*CUDA by example — An Introduction to General-Purpose GPU Programming*", (Addison Wesley, Upper Saddle River, 2011).

Julia: GPU code — improvements

- On Fermi, maximum number of resident blocks per SM is 8 (16 for Kepler), hence 24 out 32 cores (176 out of 192 for Kepler) are idle

Julia: GPU code — improvements

- On Fermi, maximum number of resident blocks per SM is 8 (16 for Kepler), hence 24 out 32 cores (176 out of 192 for Kepler) are idle
- To improve, one could introduce tiles of, say, 16×16 pixels with execution configuration

```
dim3    grid((DIM+15)/16,(DIM+15)/16);
dim3    block(16, 16);
kernel<<<grid,block>>>( dev_bitmap );
```

and corresponding modification to kernel,

```
int x = blockIdx.x*blockDim.x+threadIdx.x;
int y = blockIdx.y*blockDim.y+threadIdx.y;
int offset = x + y * DIM;
if(x >= DIM or y >= DIM) return;
```

Julia: GPU code — improvements

- On Fermi, maximum number of resident blocks per SM is 8 (16 for Kepler), hence 24 out 32 cores (176 out of 192 for Kepler) are idle
- To improve, one could introduce tiles of, say, 16×16 pixels with execution configuration

```
dim3    grid((DIM+15)/16,(DIM+15)/16);
dim3    block(16, 16);
kernel<<<grid,block>>>( dev_bitmap );
```

and corresponding modification to kernel,

```
int x = blockIdx.x*blockDim.x+threadIdx.x;
int y = blockIdx.y*blockDim.y+threadIdx.y;
int offset = x + y * DIM;
if(x >= DIM or y >= DIM) return;
```

- could have used virtual unified addressing to eliminate the CPU copy of the image

Julia: GPU code — improvements

- On Fermi, maximum number of resident blocks per SM is 8 (16 for Kepler), hence 24 out 32 cores (176 out of 192 for Kepler) are idle
- To improve, one could introduce tiles of, say, 16×16 pixels with execution configuration

```
dim3    grid((DIM+15)/16,(DIM+15)/16);
dim3    block(16, 16);
kernel<<<grid,block>>>( dev_bitmap );
```

and corresponding modification to kernel,

```
int x = blockIdx.x*blockDim.x+threadIdx.x;
int y = blockIdx.y*blockDim.y+threadIdx.y;
int offset = x + y * DIM;
if(x >= DIM or y >= DIM) return;
```

- could have used virtual unified addressing to eliminate the CPU copy of the image
- could integrate with OpenGL for interactive rendering
- ...

Outline

1 Latency vs. throughput

2 GPU architecture

3 Execution model

4 Memory hierarchy

5 CUDA Programming

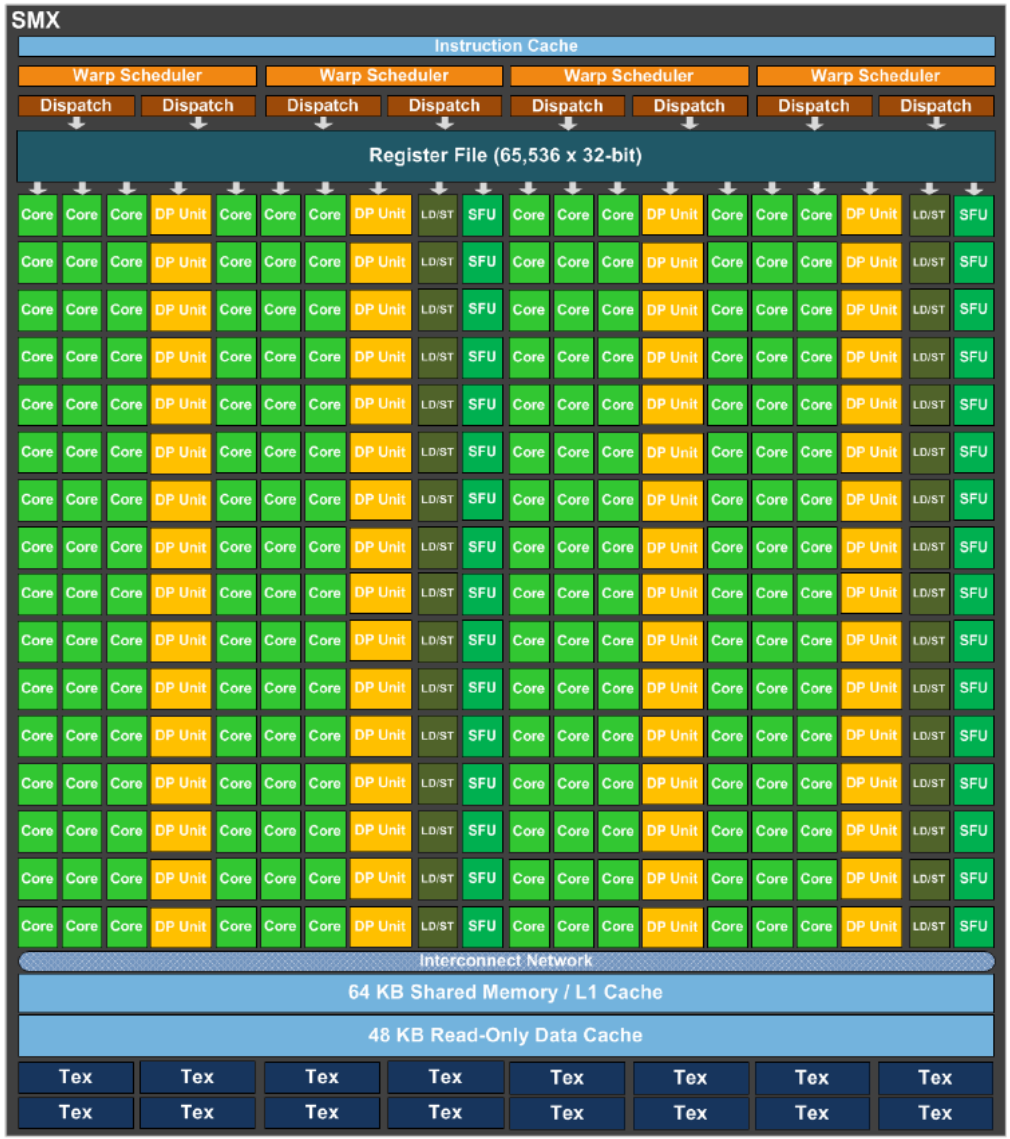
6 A worked example

7 New in Kepler

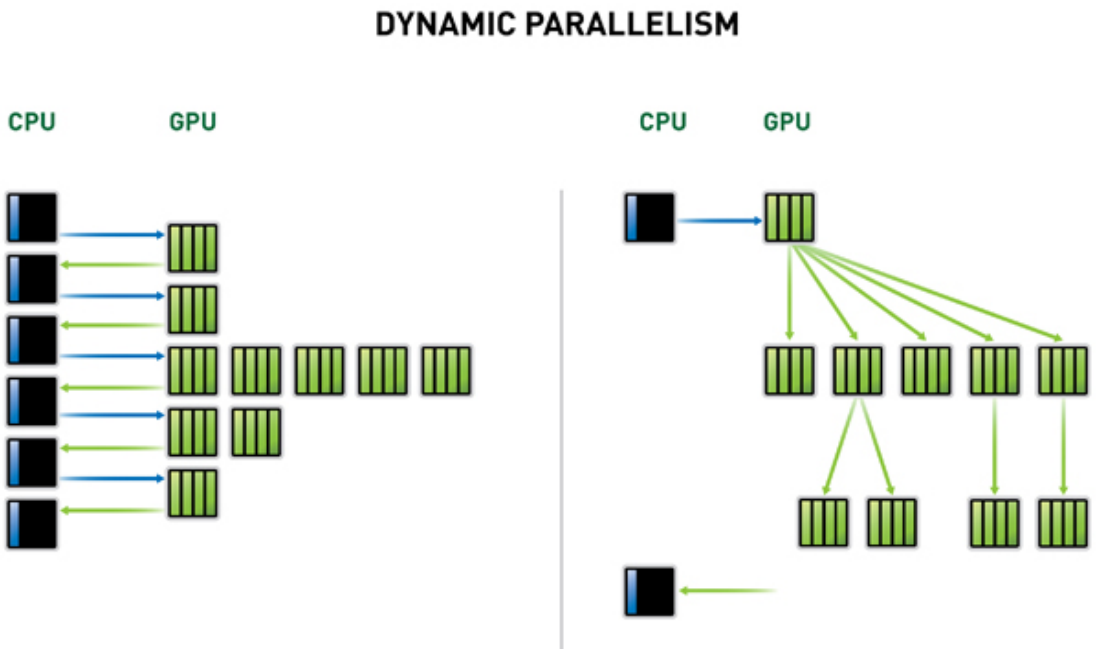
Extended streaming multiprocessor (SMX)

Features

- 192 (instead of 32) cores
- one special function unit per 6 cores (instead of 8)
- maximum number of threads/SM 2048 (instead of 1536)
- 64K 32-bit registers
- 4 warp schedulers
- more flexible shared memory configurations

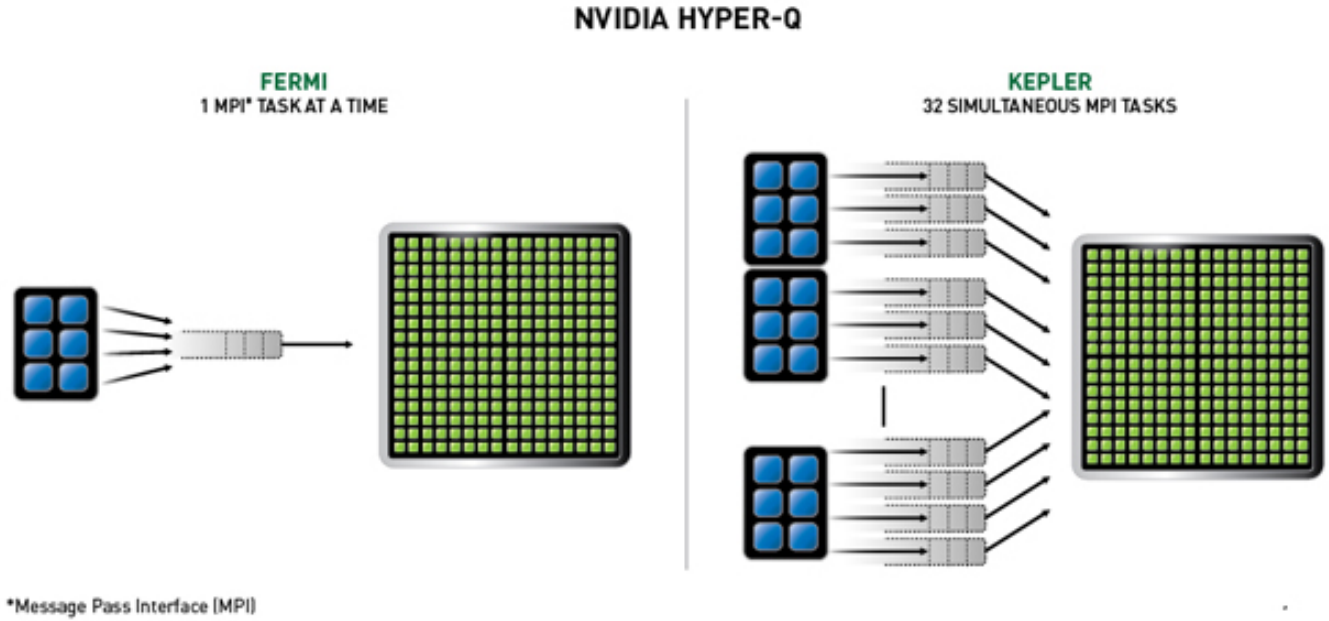


Dynamic parallelism



- kernels can more easily generate new threads
- kernels can manage associated streams
- more versatile for recursive algorithms
- altogether better load balancing

Hyper-Q



- several work queues allowed to access GPU
- in particular, several CPU threads can access came GPU
- optimized for mixed-type parallelism in MPI clusters

Kepler: further improvements

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

- quad-warp scheduler
- warp shuffle instruction
- some features, such as HyperQ and Dynamic Parallelism are *disabled* in desktop/gaming GPUs!

Summary and outlook

This lecture

This lecture has given a basic introduction into GPGPU and, in particular, the CUDA framework of GPU programming. A basic example has provided a feel for how to go about in using these devices for scientific computing.

Summary and outlook

This lecture

This lecture has given a basic introduction into GPGPU and, in particular, the CUDA framework of GPU programming. A basic example has provided a feel for how to go about in using these devices for scientific computing.

Next lecture

In lecture 2, we will start using GPUs for simulating spin models with local algorithms. In terms of GPU programming, a number of additional concepts such as thread synchronization, memory coalescence, and atomic operations will be introduced.

Summary and outlook

This lecture

This lecture has given a basic introduction into GPGPU and, in particular, the CUDA framework of GPU programming. A basic example has provided a feel for how to go about in using these devices for scientific computing.

Next lecture

In lecture 2, we will start using GPUs for simulating spin models with local algorithms. In terms of GPU programming, a number of additional concepts such as thread synchronization, memory coalescence, and atomic operations will be introduced.

Reading

- Zillions of internet resources, e.g., N. Matloff, "*Programming on Parallel Machines*", `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`
- D. B. Kirk, W.-m. W. Hwu, "*Programming Massively Parallel Processors*" (Morgan Kaufmann, Amsterdam, 2010).
- J. Sanders, E. Kandrot: "*CUDA by example — An Introduction to General-Purpose GPU Programming*", (Addison Wesley, Upper Saddle River, 2011).