INTRODUCTION TO CUDA PROGRAMMING

BHUPENDER THAKUR

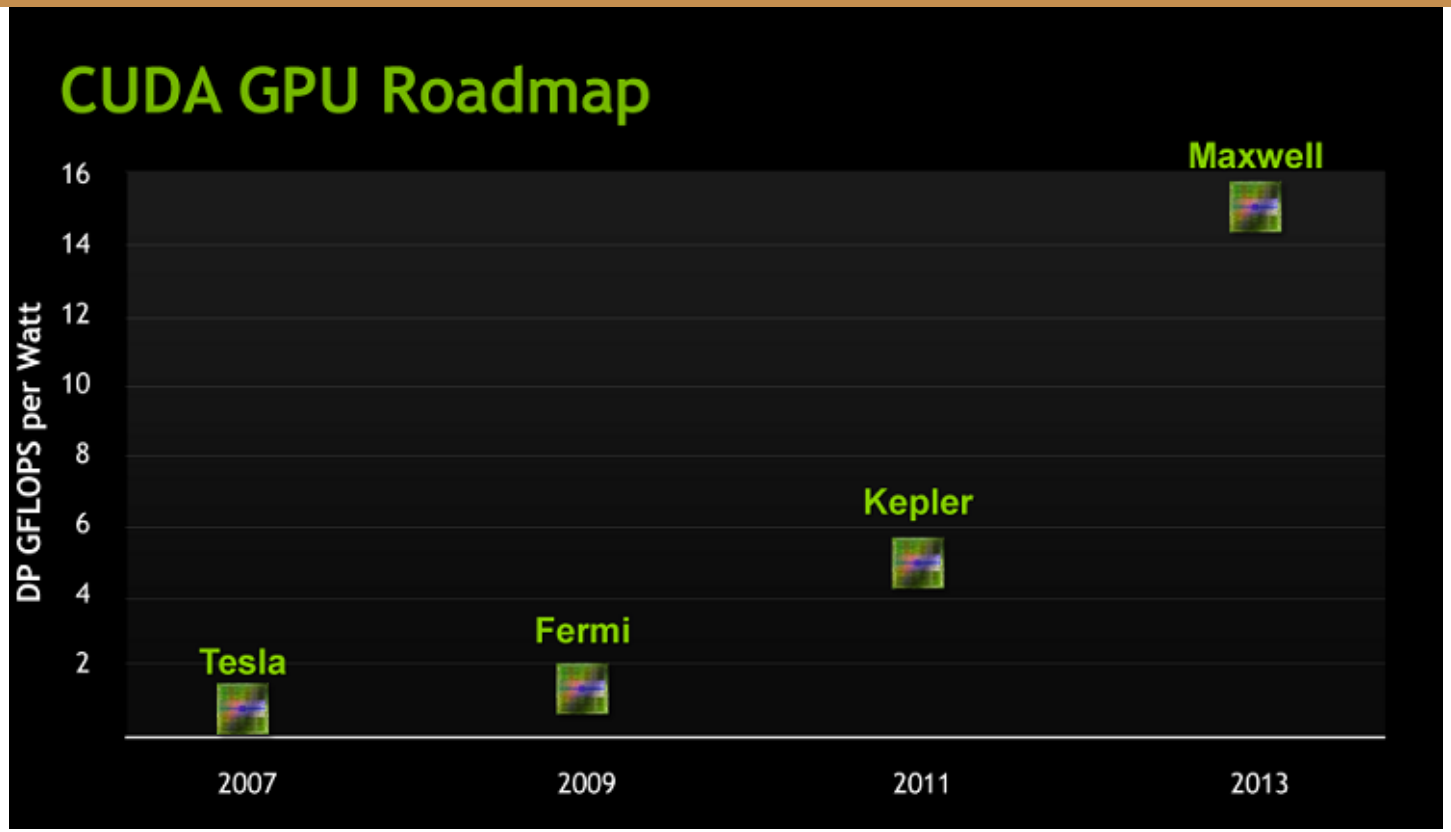
Outline

Outline of the talk

- GPU architecture
- CUDA programming model
- CUDA tools and applications
- Benchmarks

LSU

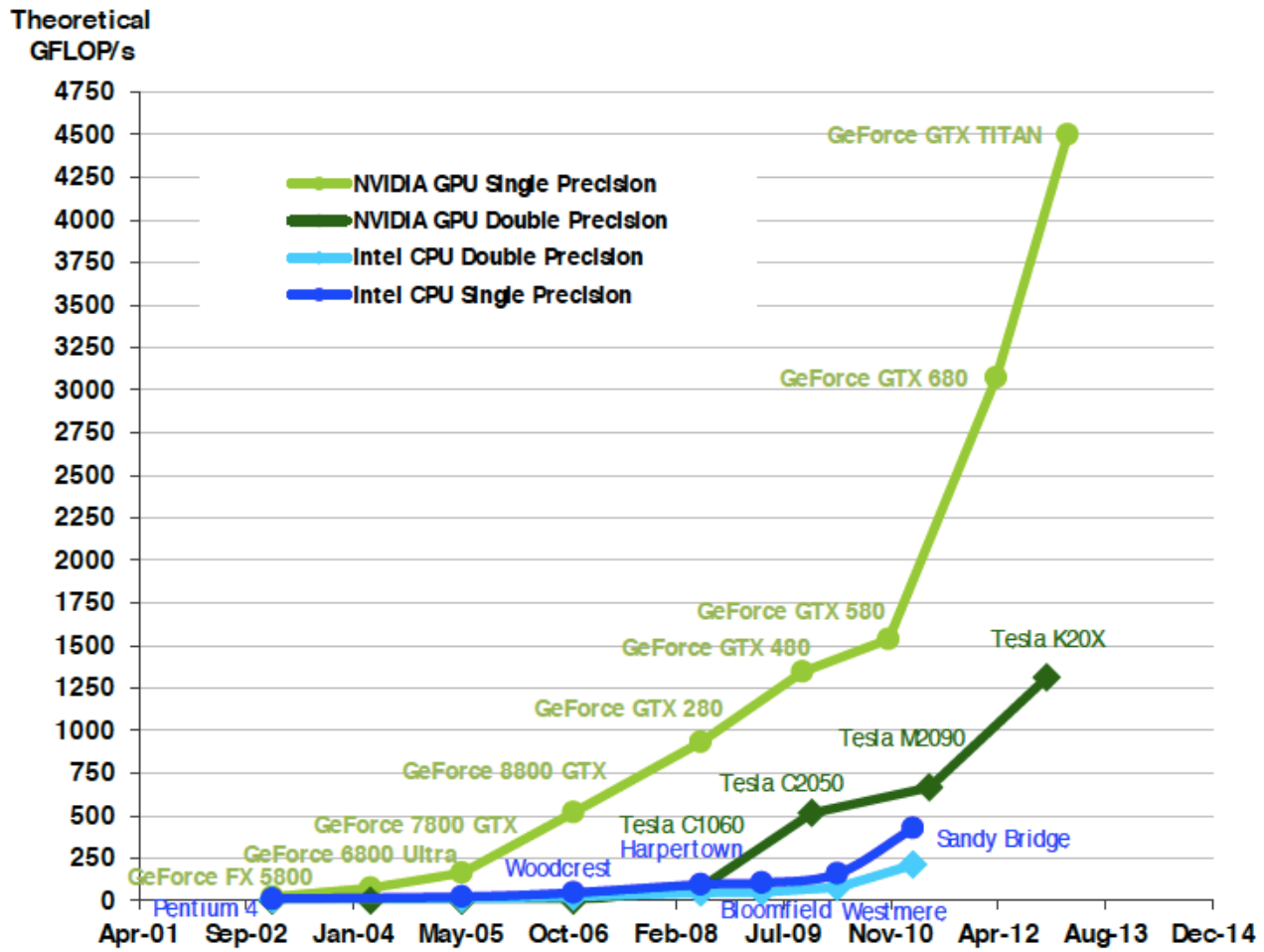
Growth in GPU computing



- Kepler is the current release.
- SuperMike II has two Fermi 2090 GPU's on the gpu nodes
- Queenbee replacement is expected to have Nvidia Kepler GPUs

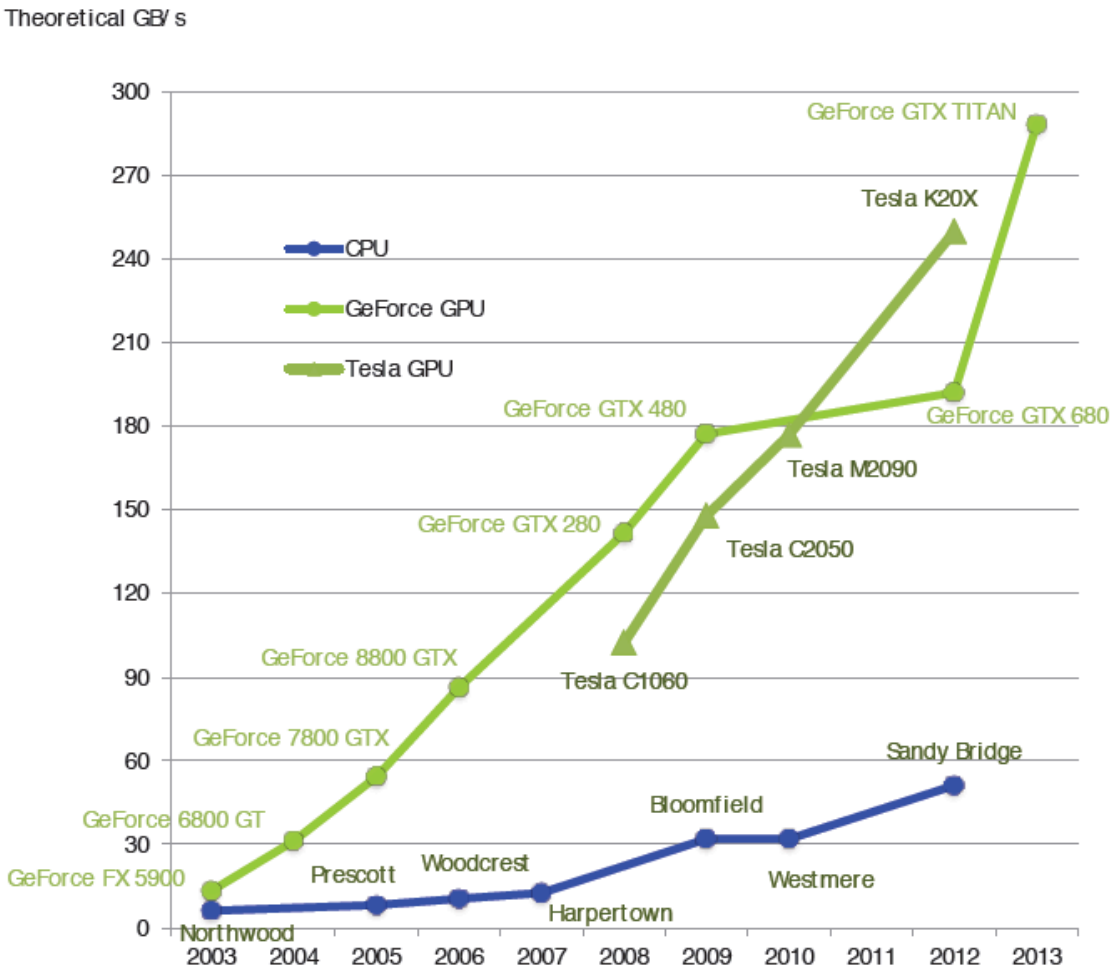
http://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/

Large theoretical GFLOPs count



http://docs.nvidia.com/cuda/cuda-c-programming-guide/

High Bandwidth



http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Notable Data Center products

Tesla Data Center Products

GPU	Compute Capability
Tesla K20	3.5
Tesla K10	3.0
Tesla M2050/M2070/M2075/M2090	2.0
Tesla S1070	1.3
Tesla M1060	1.3
Tesla S870	1.0

Compute capabilities

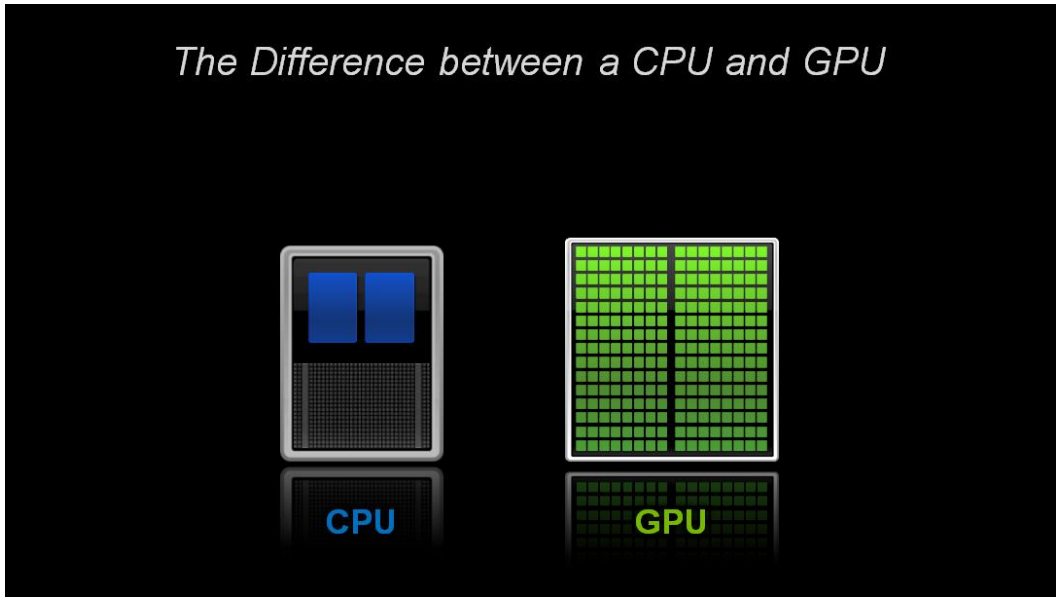
	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
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

- The GPU's were originally designed primarily for 3D game rendering.
- SM – Streaming multi-processors with multiple processing cores
- Streaming multi-processors with multiple processing cores
- On Fermi, each SM contains 32 processing cores, Kepler SMX has 192
- Execute in a Single Instruction Multiple Thread (SIMT) fashion
- Fermi has up to 16 SMs on a card for a maximum of 512 compute cores

http://www.theregister.co.uk/Print/2012/05/18/inside_nvidia_kepler2_gk110_gpu_tesla/

LSU

Outline



- The GPU's were originally designed primarily for 3D game rendering.
- SM – Streaming multi-processors with multiple processing cores
- Streaming multi-processors with multiple processing cores
- On Fermi, each SM contains 32 processing cores, Kepler SMX has 192
- Execute in a Single Instruction Multiple Thread (SIMT) fashion
- Fermi has up to 16 SMs on a card for a maximum of 512 compute cores

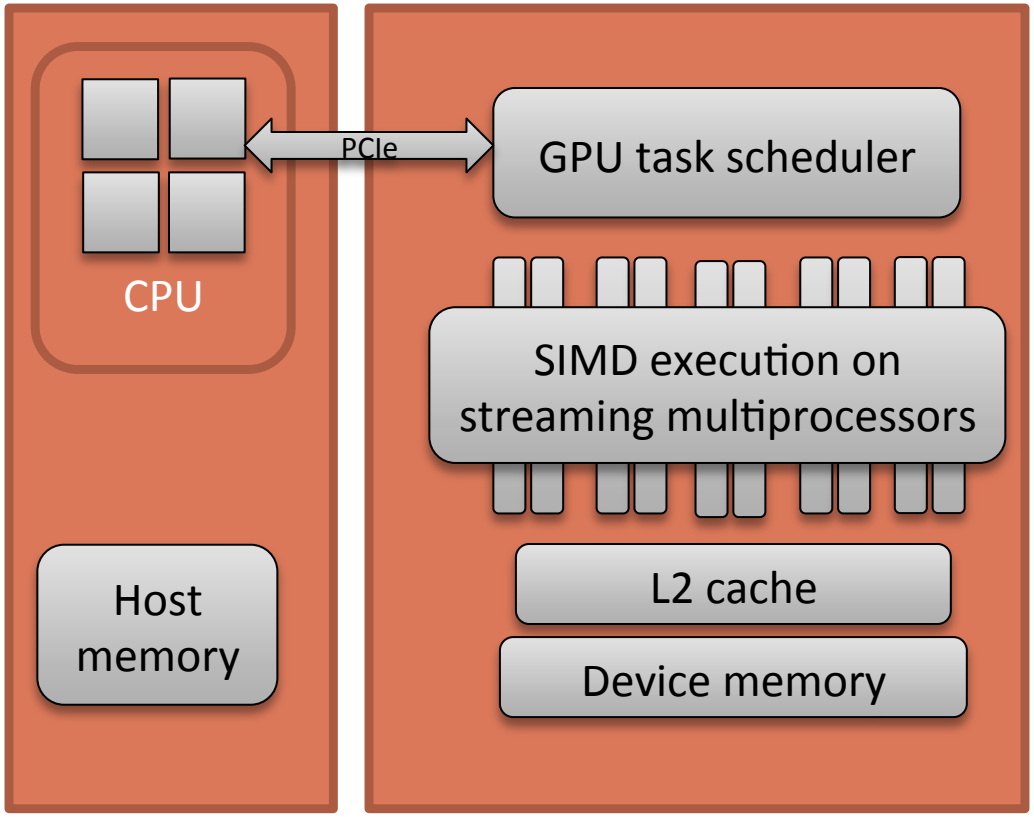
http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

LSU

GPU design

Model GPU design

- Large number of cores working in SIMD mode
- Slow global memory access, high bandwidth
- CPU communication over PCI bus
- Warp scheduling and fast switching queue model



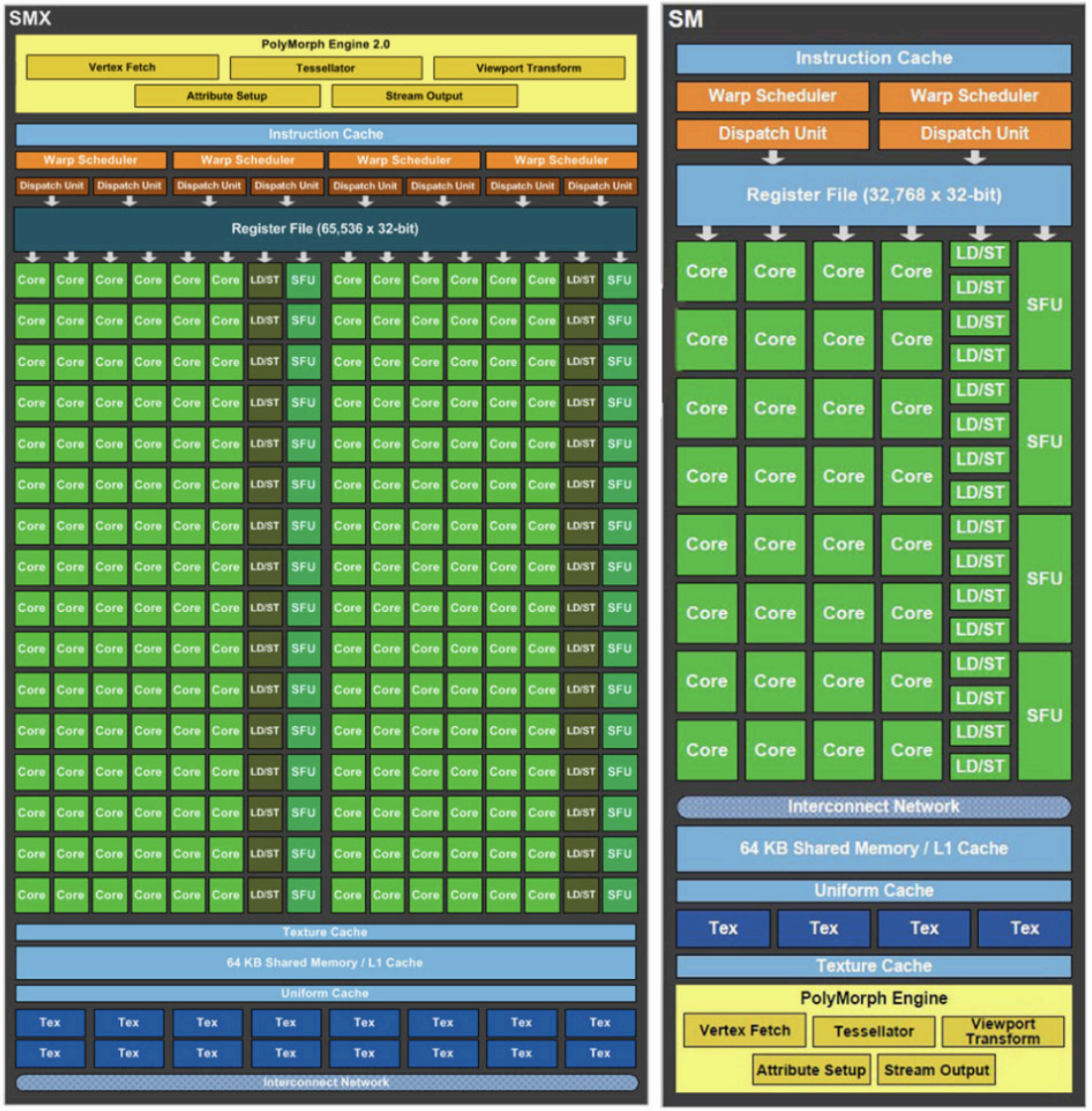
Streaming Multiprocessor

Streaming Multiprocessor

Fermi SM

vs

Kepler SMX



Device query (Fermi M2090)

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla M2090"

CUDA Driver Version / Runtime Version	**5.5 / 5.5**
CUDA Capability Major/Minor version number:	**2.0**
Total amount of global memory:	5375 MBytes (5636554752 bytes)
(16) Multiprocessors, (32) CUDA Cores/MP:	**512 CUDA Cores**
GPU Clock rate:	1301 MHz (1.30 GHz)
Memory Clock rate:	1848 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	786432 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	32768
Warp size:	32
Maximum number of threads per multiprocessor:	1536
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(65535, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	10 / 0

Compute Mode:

< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Device query (Kepler K20xm)

CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "Tesla K20Xm"

CUDA Driver Version / Runtime Version	**5.5 / 5.5**
CUDA Capability Major/Minor version number:	**3.5**
Total amount of global memory:	5760 MBytes (6039339008 bytes)
(14) Multiprocessors, (192) CUDA Cores/MP:	**2688 CUDA Cores**
GPU Clock rate:	732 MHz (0.73 GHz)
Memory Clock rate:	2600 Mhz
Memory Bus Width:	384-bit
L2 Cache Size:	1572864 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 655
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 lay
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	49152 bytes
Total number of registers available per block:	**65536**
Warp size:	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 2 copy engine(s)
Run time limit on kernels:	No
Integrated GPU sharing Host Memory:	No
Support host page-locked memory mapping:	Yes
Alignment requirement for Surfaces:	Yes
Device has ECC support:	Enabled
Device supports Unified Addressing (UVA):	Yes
Device PCI Bus ID / PCI location ID:	42 / 0

Compute Mode:

< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

Memory hierarchy

Memory Model

Registers
Per thread
Data lifetime = *thread lifetime*

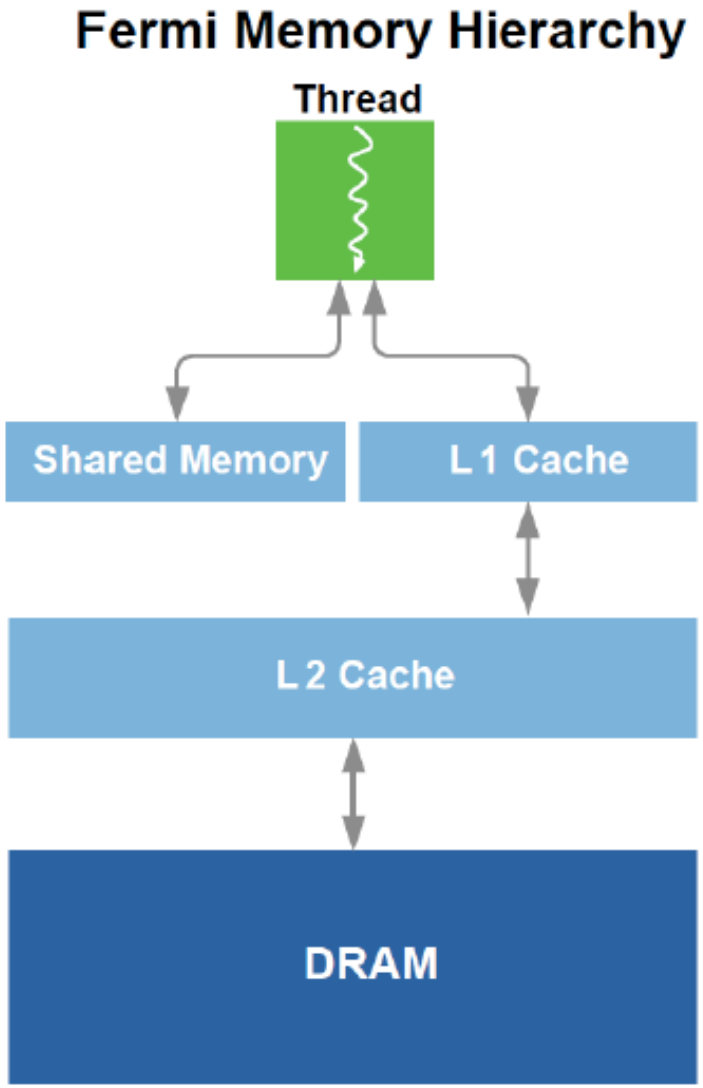
Local memory: Per thread off-chip memory (physically in device DRAM)
Data lifetime = *thread lifetime*

Shared memory
Per thread block on-chip memory
Data lifetime = *block lifetime*

Global (device) memory
Accessible by all threads as well as host (CPU)
Data lifetime = *from allocation to deallocation*

Host (CPU) memory
Not directly accessible by CUDA threads

The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1 cache.



Memory hierarchy

Memory	on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x.

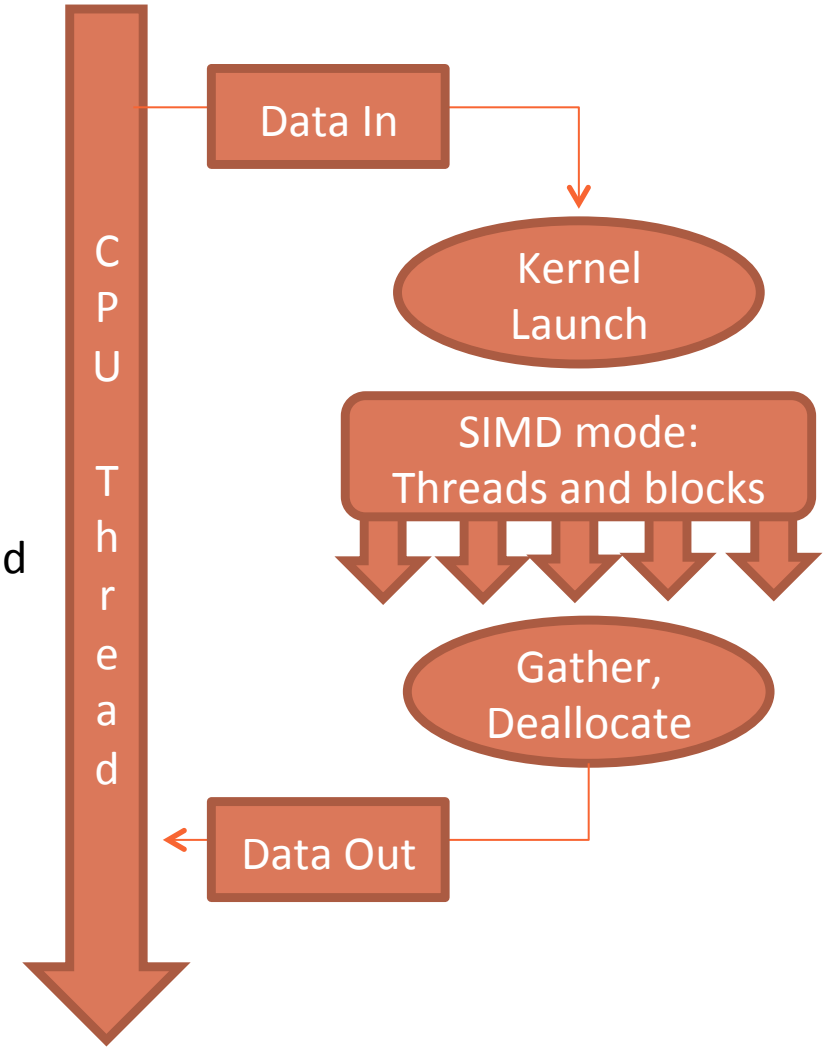
Programming model follows the architecture

Programming model:

1. Allocate device resource
2. Transfer data
3. Parallel execution of device code
4. Copy results back to host thread

General considerations:

- Minimize data transfers between host and device
- Avoid serialization of device code: branching, divergence, low occupancy, non-strided memory access
- Data reuse



Variable declaration

Variable declaration	Memory	Scope	Lifetime
__device__ __local__ int LocalVar;	local	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

LSU

Memory allocation

// Device memory allocation

cudaMalloc(void **pointer, size_t nbytes)

// Initialization

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

// Device memory Release

cudaFree(void *pointer)

```
int n = 1024;
int nbytes = n*sizeof(int);

cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
…
cudaFree(a_d);
```

LSU

Memory copy to/from device

// Copy data to/from GPU

cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

direction specifies locations (host or device) of src and dst
Blocks CPU thread: returns after the copy is complete
Doesn't start copying until previous CUDA calls complete

// Direction of copy is specified by:

enum cudaMemcpyKind
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

LSU

Executing code on the GPU(kernels)

// Kernels

Kernels are C functions with some restrictions
Can only access GPU memory
Must have void return type
No variable number of arguments (“varargs”)
Not recursive
No static variables

// Launching kernels

Modified C function call syntax Execution Configuration (“<<< >>>”):

```
// Allocate and copy data to GPU
kernel<<<dim3 grid, dim3 block>>>(…)
// Copy data back
```

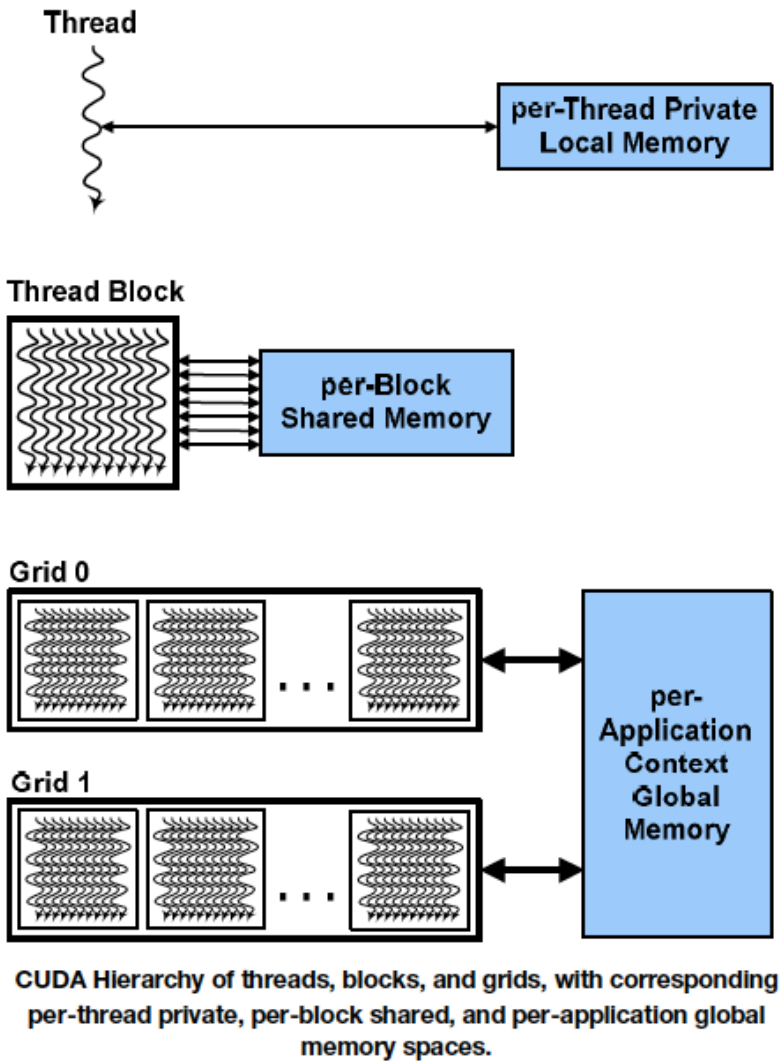
E.g:

```
kernel<<<grid, block>>>(...);
kernel<<<32, 512>>>(...);
```

Kernel configuration

Kernel configuration

- Threads/ local,shared mem
- Threadblocks
- Grid of threadblocks



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

Executing code on the GPU(kernels)

// Built in variables

blockIdx.x, blockIdx.y, blockIdx.z are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.

threadIdx.x, threadIdx.y, threadIdx.z are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.

blockDim.x, blockDim.y, blockDim.z are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

Executing code on the GPU(kernels)

// Built in variables

blockIdx.x, blockIdx.y, blockIdx.z are built-in variables that returns the block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code.

threadIdx.x, threadIdx.y, threadIdx.z are built-in variables that return the thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this stream processor in this particular block.

blockDim.x, blockDim.y, blockDim.z are built-in variables that return the "block dimension" (i.e., the number of threads in a block in the x-axis, y-axis, and z-axis).

Full global thread ID in x and y dimensions can be computed by:

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

LSU

Executing code on the GPU(kernels)

// Built in variables uint3, dim3

uint3 and dim3 are CUDA-defined structures of unsigned integers: x, y, and z.

- struct uint3 {x; y; z;};
- struct dim3 {x; y; z;};

- The unsigned structure components are automatically initialized to 1.
- These vector types are mostly used to define grid of blocks and threads.

// Setting dimensions via uint3, dim3

dim3 gridDim -- Grid dimensions, x and y (z not used).

Number of blocks in grid = gridDim.x * gridDim.y

dim3 blockDim -- Size of block dimensions x, y, and z.

Number of threads in a block = blockDim.x * blockDim.y * blockDim.z

```
dim3 grid(16,16); // grid = 16 x 16 blocks
dim3 block(32,32); // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

LSU

CUDA kernels

Kernels

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>execution configuration syntax

Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

// Kernel launch with one block having N threads

```
__global__ void VecAdd(float* A, float* B, float* C)
{
   int i = threadIdx.x;
   C[i] = A[i] + B[i];
}
int main()
{
   // Kernel invocation with N threads
   VecAdd<<<1, N>>>(A, B, C);
}
```

LSU

CUDA configurations

// Typical access of a 1D matrix

```
__global__ void assign( int* d_a, int value)
{
int idx = blockDim.x * blockIdx.x + threadIdx.x;
d_a[idx] = value;
}
```

LSU

CUDA configurations

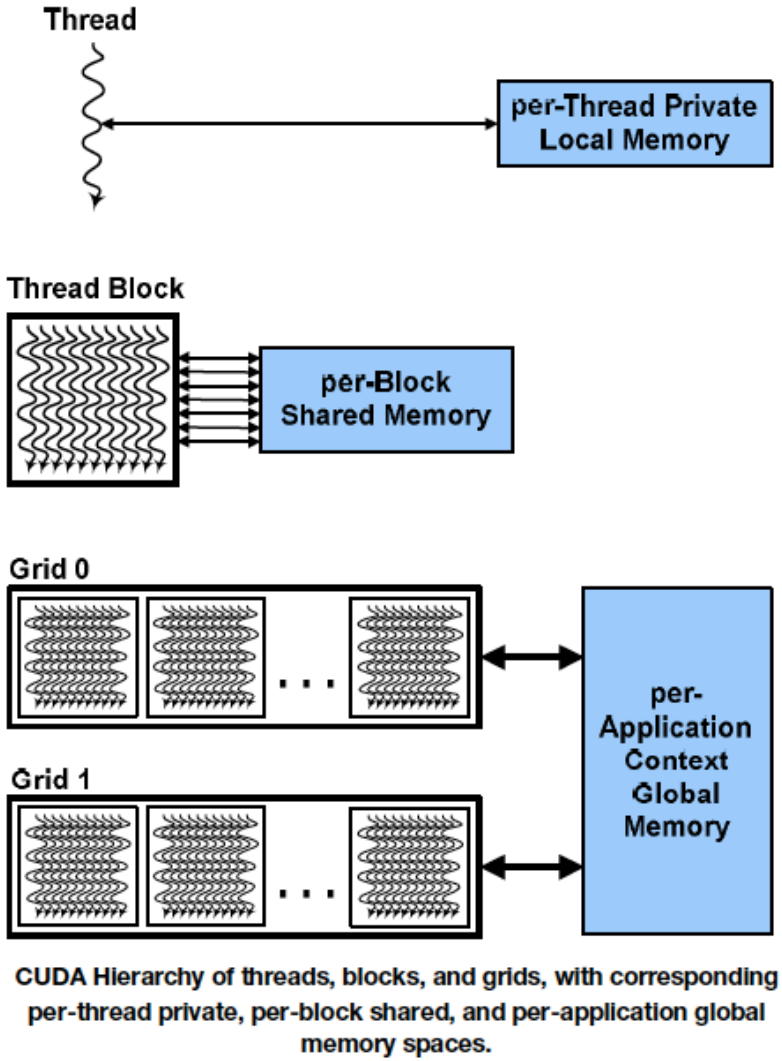
// Accessing a flattened 2D matrix

```
global__ void assign2D(int* d_a, int w, int h, int value)
{
int iy = blockDim.y * blockIdx.y + threadIdx.y;
int ix = blockDim.x * blockIdx.x + threadIdx.x;
int idx = iy * w + ix;
d_a[idx] = value;
}
...
assign2D<<<dim3(64, 64), dim3(16, 16)>>>(...);
```

CUDA programming model

Cuda

- Subset of C with extensions
- Parallel execution using threads and kernels
- CUDA threads are extremely lightweight Very little creation overhead Fast switching
- CUDA uses 1000s of threads
- Data needs to be copied in/out of GPU

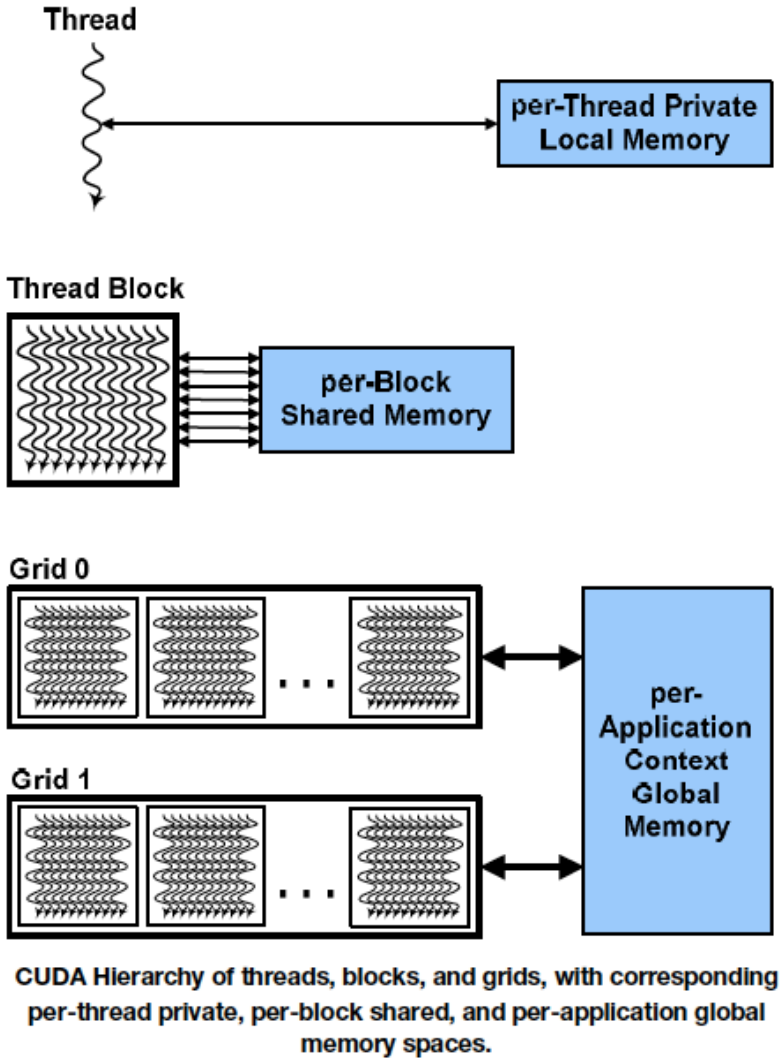


CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

CUDA programming model

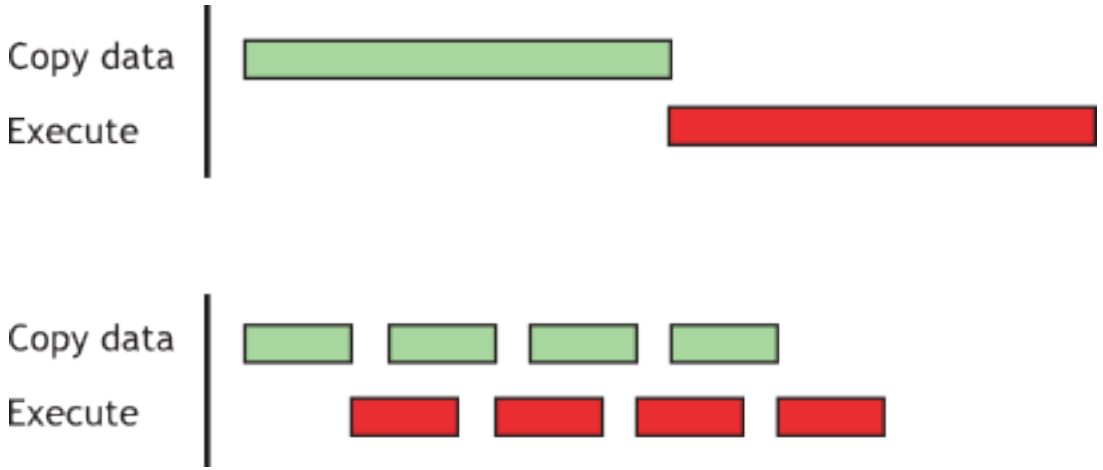
Cuda

- Threads > Blocks > kernel
- A kernel is executed by a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
  - Threads from different blocks cannot cooperate



CUDA Hierarchy of threads, blocks, and grids, with corresponding per-thread private, per-block shared, and per-application global memory spaces.

CUDA programming model



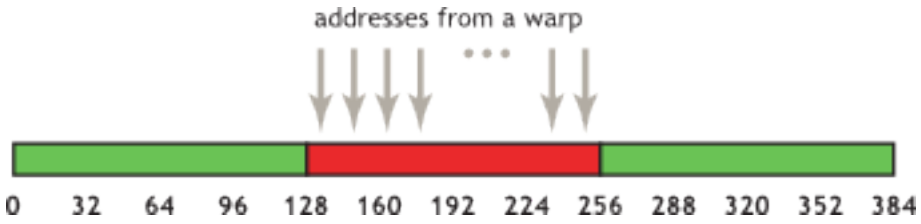
- Asynchronous memory transfers can overlap data copy with kernel execution

CUDA programming model

Synchronization:

- There is no global synchronization/barrier
- Barrier within a threadblock exists and can be called by
  __syncthreads();

CUDA programming model



- Data from global memory is fetched in warps of 32 threads. Non-aligned or random fetches serialize the calls or render them very inefficient.

http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

CUDA runtime API

There are two levels for the runtime API.

- The C API (cuda_runtime_api.h) is a C-style interface that does not require compiling with nvcc.
- The C++ API (cuda_runtime.h) is a C++-style interface built on top of the C API.

Device Management
Error Handling
Stream Management
Event Management
Execution Control
Memory Management
Unified Addressing
Peer Device Memory Access
OpenGL Interoperability
Direct3D 9 Interoperability
Direct3D 10 Interoperability
Direct3D 11 Interoperability
VDPAU Interoperability
Graphics Interoperability
Texture Reference Management
Surface Reference Management
Version Management
C++ API Routines
C++-style interface built on fCUDA runtime API.
Interactions with the CUDA Driver API
Interactions between the CUDA Driver API and the CUDA Runtime API.
Profiler Control
Data types used by CUDA Runtime

LSU

CUDA runtime API : Device Management

Functions

cudaChooseDevice (int *device, const struct cudaDeviceProp *prop)
Select compute-device which best matches criteria.

cudaDeviceSynchronize (void)
Wait for compute device to finish.

cudaGetDeviceCount (int *count)
Returns the number of compute-capable devices.

CUDA runtime API : Device Management

Device Enumeration

A host system can have multiple devices. The following code sample shows how to enumerate these devices, query their properties, and determine the number of CUDA-enabled devices.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
   cudaDeviceProp deviceProp;
   cudaGetDeviceProperties(&deviceProp, device);
   printf("Device %d has compute capability %d.%d.\n",
         device, deviceProp.major, deviceProp.minor);
}
```

CUDA runtime API : Error Management

Functions

cudaGetErrorString (cudaError_t error)
Returns the message string from an error code.

cudaGetLastError (void)
Returns the last error from a runtime call.

cudaPeekAtLastError (void)
Returns the last error from a runtime call.

CUDA runtime API : Stream Management

Functions

cudaStreamCreate (cudaStream_t *pStream)
Create an asynchronous stream.

cudaStreamDestroy (cudaStream_t stream)
Destroys and cleans up an asynchronous stream.

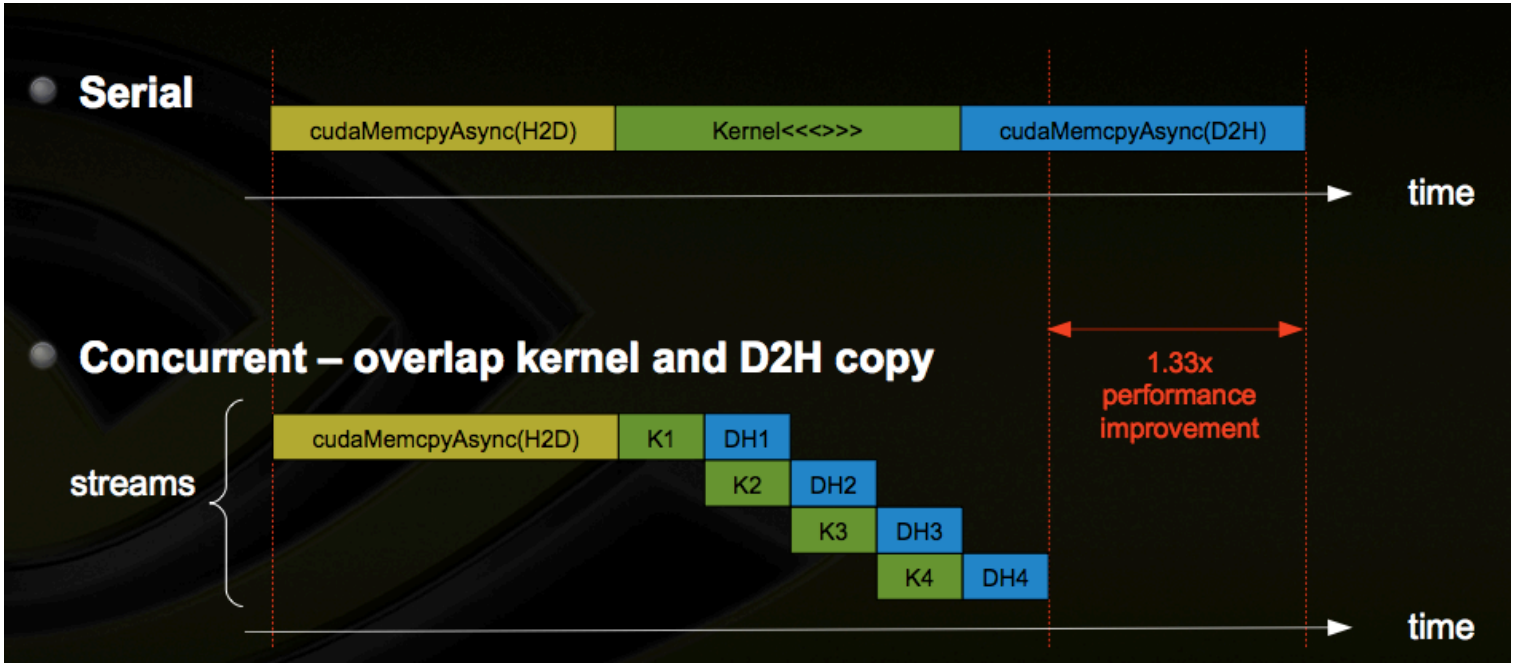
cudaStreamQuery (cudaStream_t stream)
Queries an asynchronous stream for completion status.

cudaStreamSynchronize (cudaStream_t stream)
Waits for stream tasks to complete.

cudaStreamWaitEvent (cudaStream_t stream, cudaEvent_t event, unsigned int flags)
Make a compute stream wait on an event.

LSU

CUDA runtime API : Stream Management



http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

CUDA runtime API : Memory Management

Functions

cudaFree (void *devPtr)
Frees memory on the device.

cudaMemcpyAsync (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

cudaHostAlloc (void **pHost, size_t size, unsigned int flags)
Allocates page-locked memory on the host.

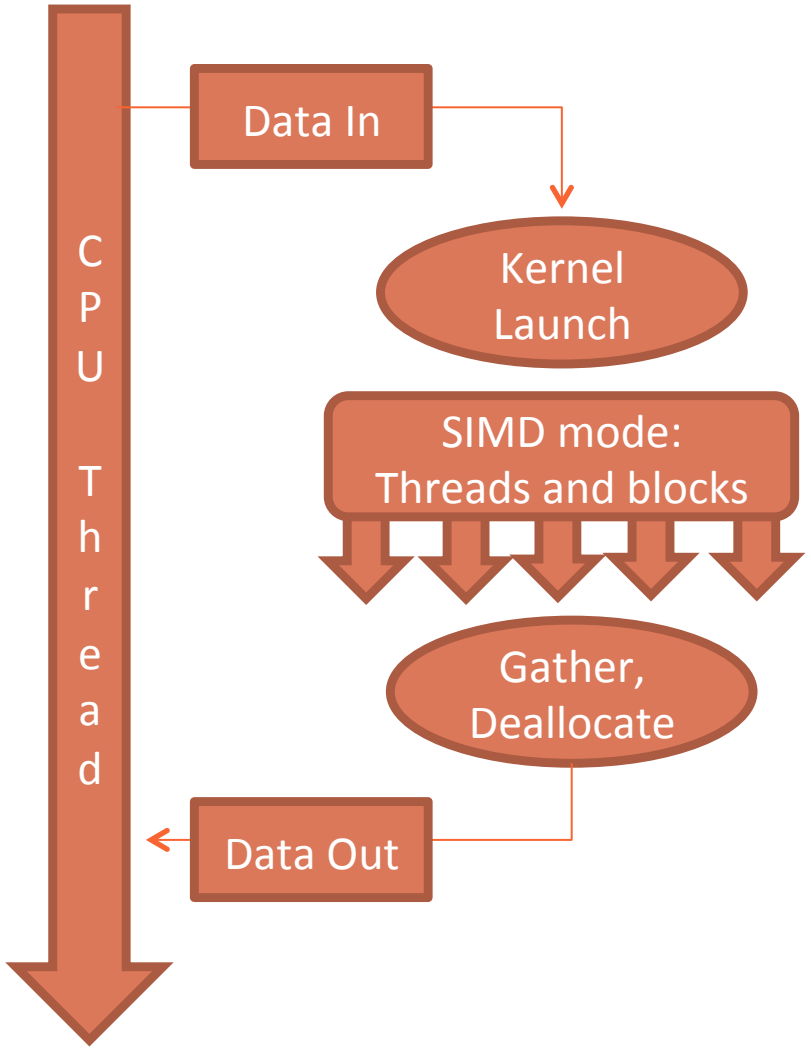
cudaMalloc (void **devPtr, size_t size)
Allocate memory on the device.

cudaMemcpyAsync (void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream=0)
Copies data between host and device.

LSU

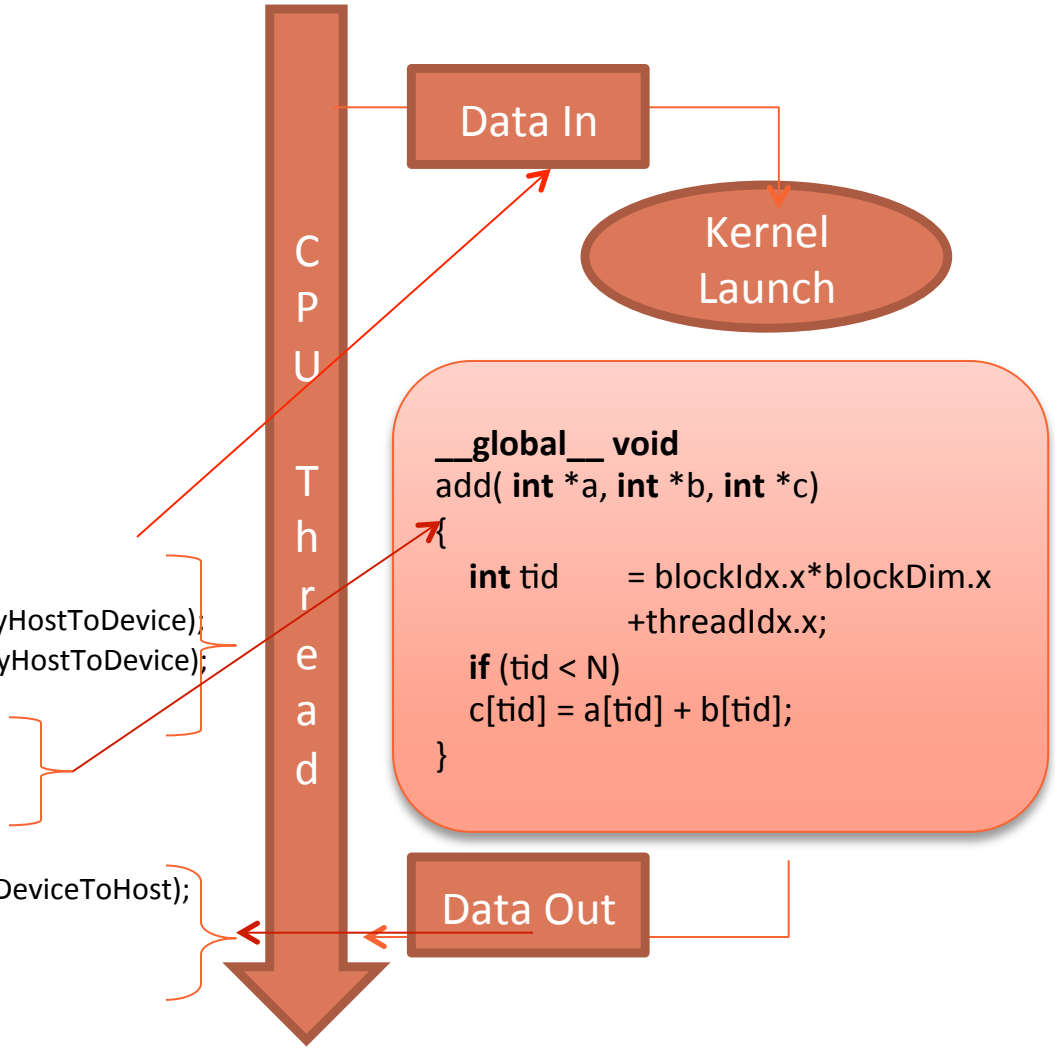
Examples: Vector add

```
int main( void ) {
    int a[n], b[n], c[n];
    int *a_d, *b_d, *c_d;
    size_t nbytes=n*sizeof(int);

// allocate the memory on the GPU
    cudaMalloc( (void**)&a_d, nbytes );
    cudaMalloc( (void**)&b_d, nbytes );
    cudaMalloc( (void**)&c_d, nbytes ) ;

// fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<n; i++) {
    a[i] = -I; b[i] = i * i;}

// copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( a_d, a, N*sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy( b_d, b, N*sizeof(int),  cudaMemcpyHostToDevice);

// launch kernel
    add<<<n,1>>>( a_d, b_d, c_d );

// copy the array 'c' back to the CPU
   cudaMemcpy( c, c_d, n*sizeof(int), cudaMemcpyDeviceToHost);

// free the memory allocated on the GPU
   cudaFree( a_d ); cudaFree( b_d ); cudaFree( c_d );
return 0;}
```



Examples: Vector add

```
int main( void ) {
    int a[n], b[n], c[n];
    int *a_d, *b_d, *c_d;
    size_t nbytes=n*sizeof(int);

// allocate the memory on the GPU
    cudaMalloc( (void**)&a_d, nbytes );
    cudaMalloc( (void**)&b_d, nbytes );
    cudaMalloc( (void**)&c_d, nbytes ) ;

// fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<n; i++) {
    a[i] = -I; b[i] = i * i;}

// copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( a_d, a, N*sizeof(int), cudaMemcpyHostToDevice);
   cudaMemcpy( b_d, b, N*sizeof(int),  cudaMemcpyHostToDevice);

// launch kernel
    add<<<n,1>>>( a_d, b_d, c_d );

// copy the array 'c' back to the CPU
   cudaMemcpy( c, c_d, n*sizeof(int), cudaMemcpyDeviceToHost);

// free the memory allocated on the GPU
   cudaFree( a_d ); cudaFree( b_d ); cudaFree( c_d );
return 0;}
```



Examples: Matadd

```
__global__ void gpuadd (int *a_d, int *b_d, int *c_d)
{
   int col = blockIdx.x*blockDim.x + threadIdx.x;
   int row = blockIdx.y*blockDim.y + threadIdx.y;
   int itd = row*N+col;
   c_d[itd] = a_d[itd]+b_d[itd];

}
```

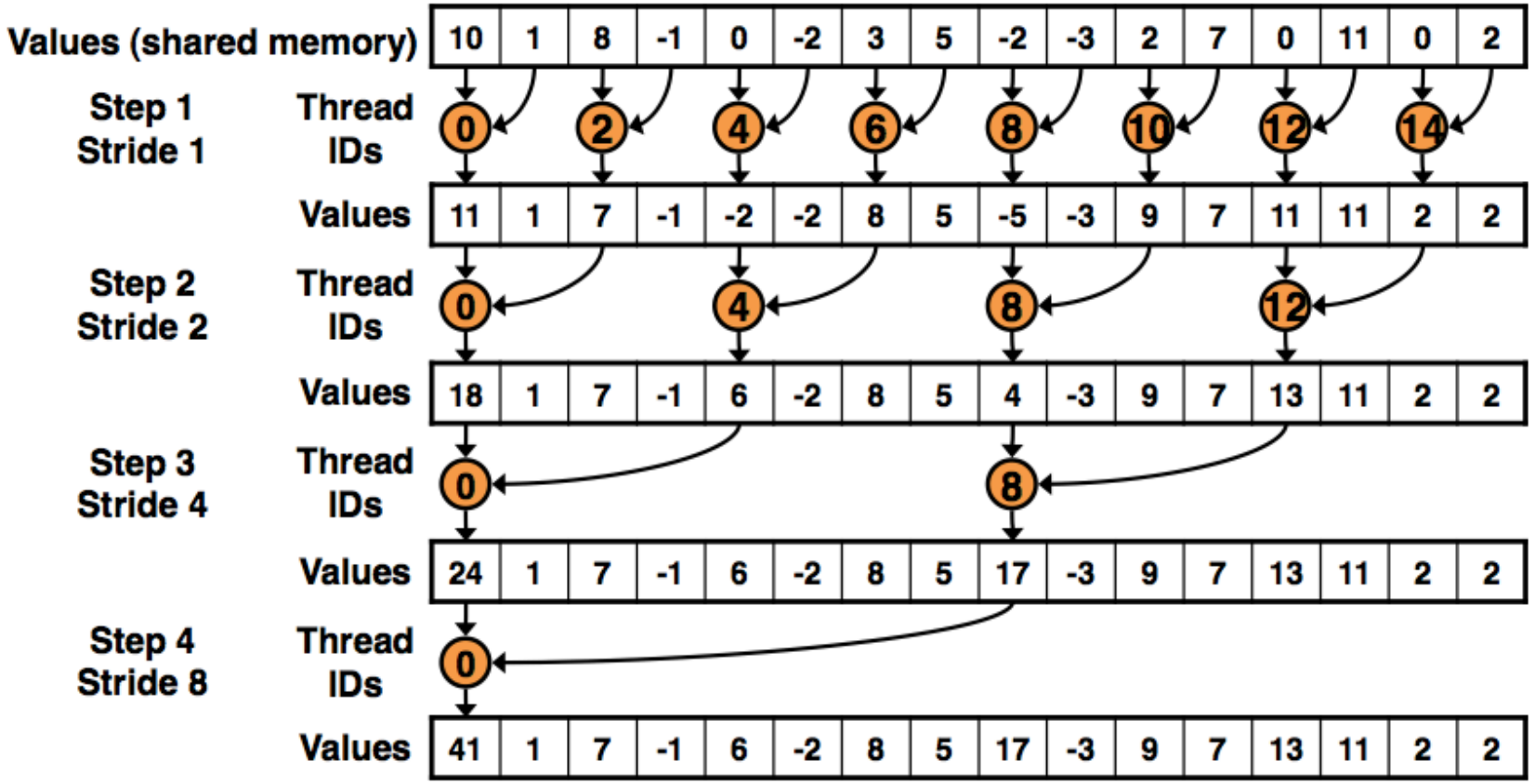
LSU

Examples: Matmul sdk code walkthrough

http://www.hpcwire.com/hpcwire/2008-10-30/compilers_and_more_optimizing_gpu_kernels.html

LSU

Examples: Reduction 1: Interleaved addressing with divergent branching



Examples: Reduction 1: Interleaved addressing with divergent branching

```
__global__ void reduce1(int *g_idata, int *g_odata) {

   extern __shared__ int sdata[];

// each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

   sdata[tid] = g_idata[i];

   __syncthreads();

// do reduction in shared mem
   for(unsigned int s=1; s < blockDim.x; s *= 2) {
   if (tid % (2*s) == 0) {
   sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
   }

// write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
   }
```

Examples: Reduction 1: Interleaved addressing with divergent branching

```
__global__ void reduce1(int *g_idata, int *g_odata) {

   extern __shared__ int sdata[];

// each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

   sdata[tid] = g_idata[i];

   __syncthreads();

// do reduction in shared mem
   for(unsigned int s=1; s < blockDim.x; s *= 2) {
   if (tid % (2*s) == 0) {
   sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
   }

// write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
   }
```

If statement based on threadid makes the loop divergent. Modulo operator on device is slow

Examples: Reduction 2: Interleaved addressing without divergent branching

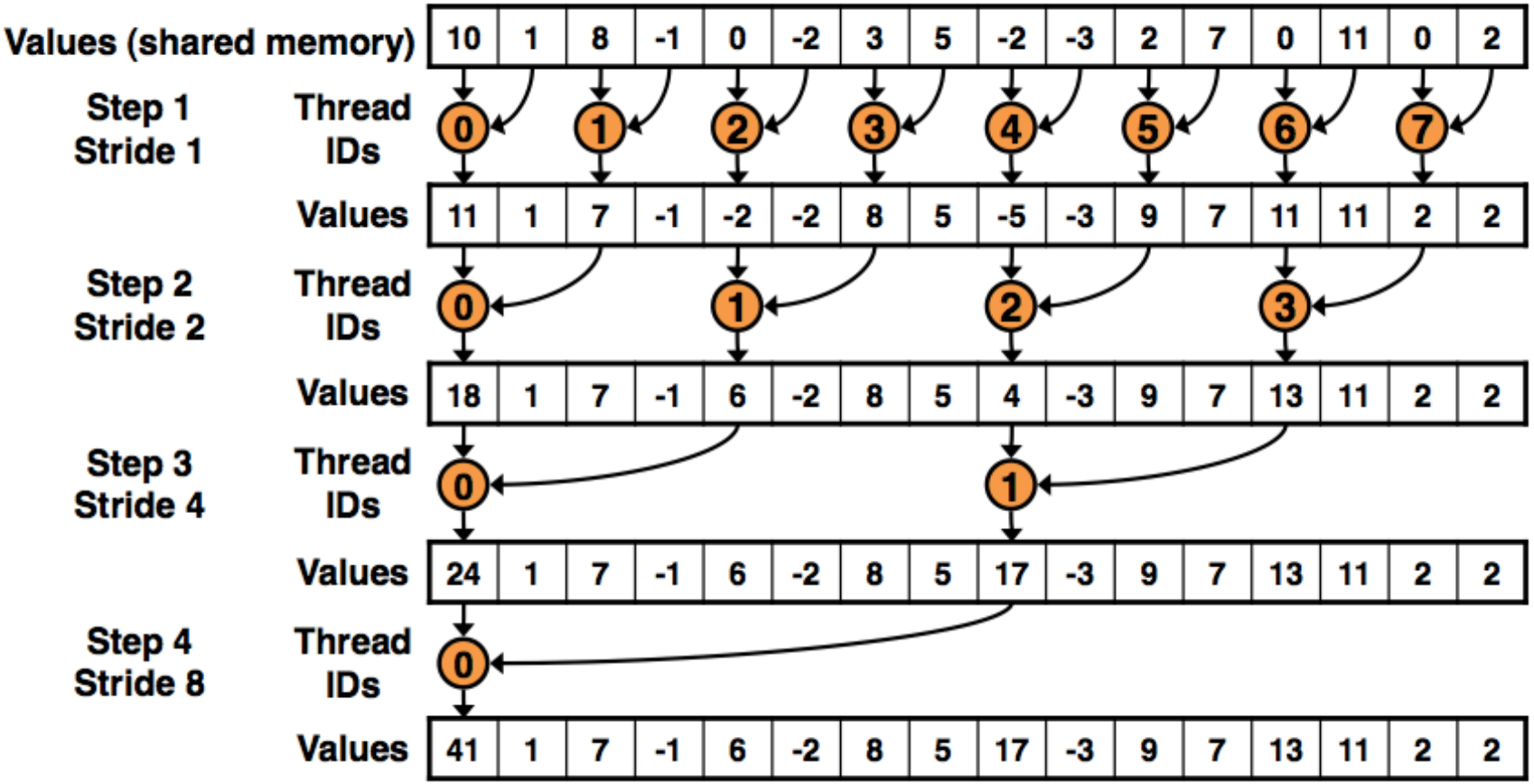
```
for (unsigned int s=1; s < blockDim.x; s *= 2)
{ if (tid % (2*s) == 0) {
   sdata[tid] += sdata[tid + s];
}
 __syncthreads();
}
```

```
for (unsigned int s=1; s < blockDim.x; s *= 2)
{ int index = 2 * s * tid;
  if (index < blockDim.x) {
  sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

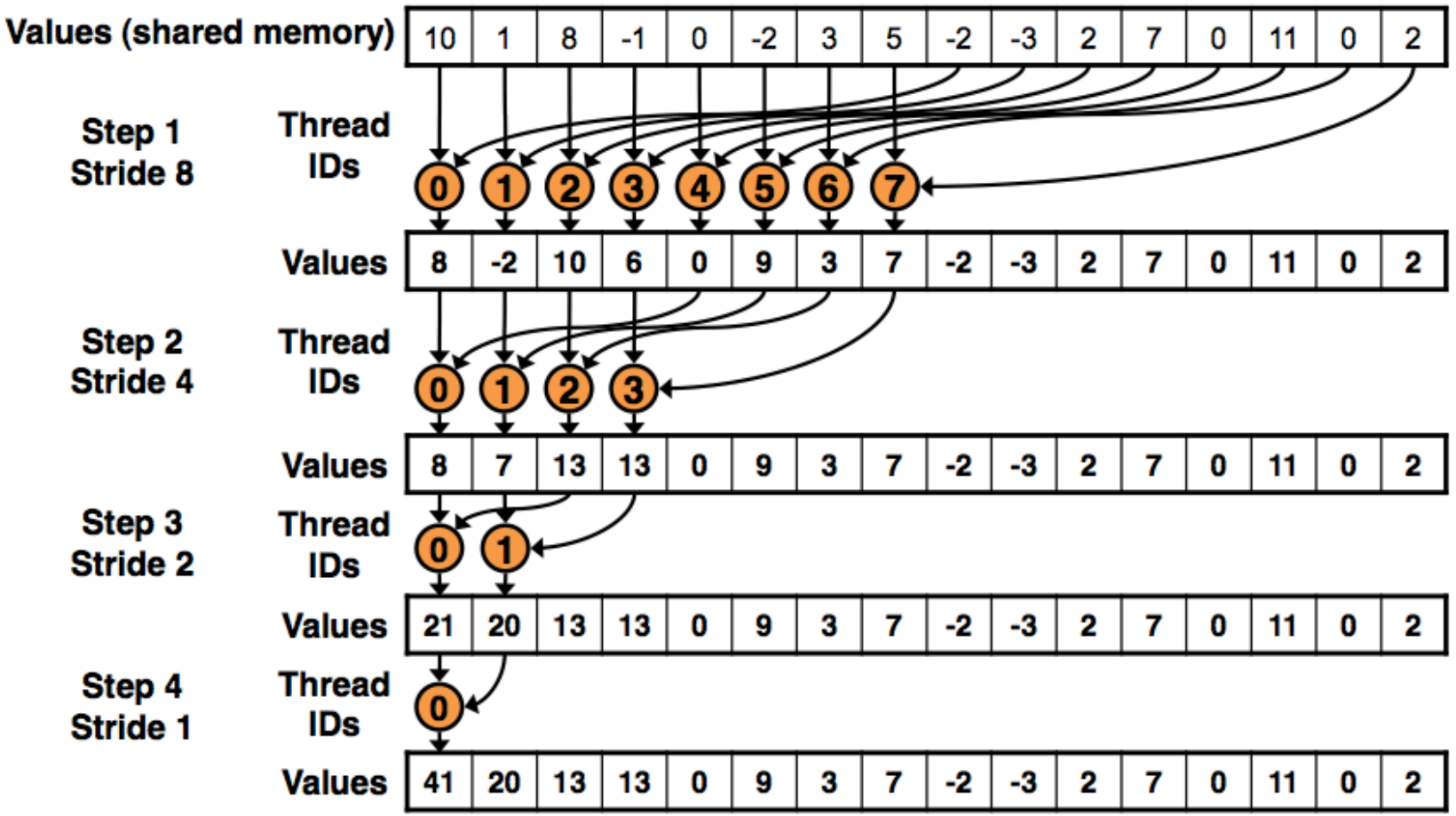
Just replace divergent branch in inner loop:
With strided index and non-divergent branch:

LSU

Examples: Reduction 2: Interleaved addressing without divergent branching



Examples: Reduction 3: Sequential adressing



Examples: Reduction 3: Sequential adressing

```
for (unsigned int s=1; s < blockDim.x; s *= 2)
{   int index = 2 * s * tid;
    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }  __syncthreads();
}
```

Just replace strided indexing in inner loop with reversed loop and threadID-based indexing

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1)
{ if (tid < s) {
   sdata[tid] += sdata[tid + s];
   }__syncthreads();
}
```

LSU

Examples: Reduction

Performance for 16M element reduction

	Bandwidth(M2090, Kepler20xm)	
Kernel 1:	8.98 GB/s	15.1571 GB/s
Kernel 2:	12.3959 GB/s	17.4061 GB/s
Kernel 3:	17.4884 GB/s	53.4588 GB/s

http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

LSU

Examples: Applications

Exercise 1

1. Modifying vector add code to calculate the norm of avector
2. Matmul program to for matrix multiplication
3. Modify reduction code to use reduction kernel 3
4. Using cuda-memcheck to check for memory errors
5. Using nvprof profiler

x86 based CPU architecture

GPU architecture