L2: Introduction to CUDA

January 14, 2009

Outline

- Overview of the CUDA Programming Model for NVIDIA systems
- Motivation for programming model
- Presentation of syntax
- Simple working example (also on website)
- **Reading:** GPU Gems 2, Ch. 31;
  CUDA 2.0 Manual, particularly Chapters 2 and 4

This lecture includes slides provided by:
Wen-mei Hwu (UIUC) and David Kirk (NVIDIA)
see http://courses.ece.uiuc.edu/ece498/al1/

and Austin Robison (NVIDIA)

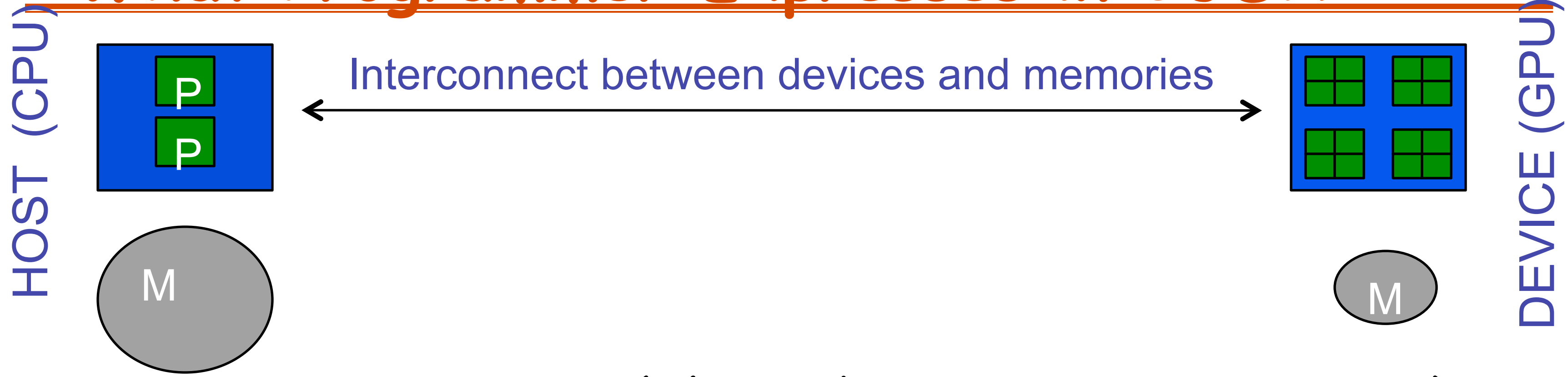
CUDA (Compute Unified Device Architecture)

- ***Data-parallel*** programming interface to GPU
  - Data to be operated on is discretized into independent partition of memory
  - Each thread performs roughly same computation to different partition of data
  - When appropriate, easy to express and very efficient parallelization
- Programmer expresses
  - Thread programs to be launched on GPU, and how to launch
  - Data organization and movement between host and GPU
  - Synchronization, memory management, testing, …
- CUDA is one of first to support ***heterogeneous*** architectures (more later in the semester)
- CUDA environment
  - Compiler, run-time utilities, libraries, emulation, performance

Today's Lecture

- Goal is to enable writing CUDA programs right away
  - Not efficient ones – need to explain architecture and mapping for that (soon)
  - Not correct ones – need to discuss how to reason about correctness (also soon)
  - Limited discussion of why these constructs are used or comparison with other programming models (more as semester progresses)
  - Limited discussion of how to use CUDA environment (more next week)
  - No discussion of how to debug. We'll cover that as best we can during the semester.

What Programmer Expresses in CUDA



- Computation partitioning (where does computation occur?)
  - Declarations on functions __host__, __global__, __device__
  - Mapping of thread programs to device: compute <<<gs, bs>>>(<args>)
- Data partitioning (where does data reside, who may access it and how?)
  - Declarations on data __shared__, __device__, __constant__, …
- Data management and orchestration
  - Copying to/from host: *e.g.*, cudaMemcpy(h_obj,d_obj, cudaMemcpyDevicetoHost)
- Concurrency management
  - *E.g.* __synchthreads()

Minimal Extensions to C + API

- Declspecs
  - global, device, shared, local, constant

```
__device__ float filter[N];

__global__ void convolve (float *image)
{

  __shared__ float region[M];
  ...
```

- Keywords
  - threadIdx, blockIdx

```
region[threadIdx] = image[i];
```

- Intrinsics
  - __syncthreads

```
  __syncthreads()
  ...

  image[j] = result;
}
```

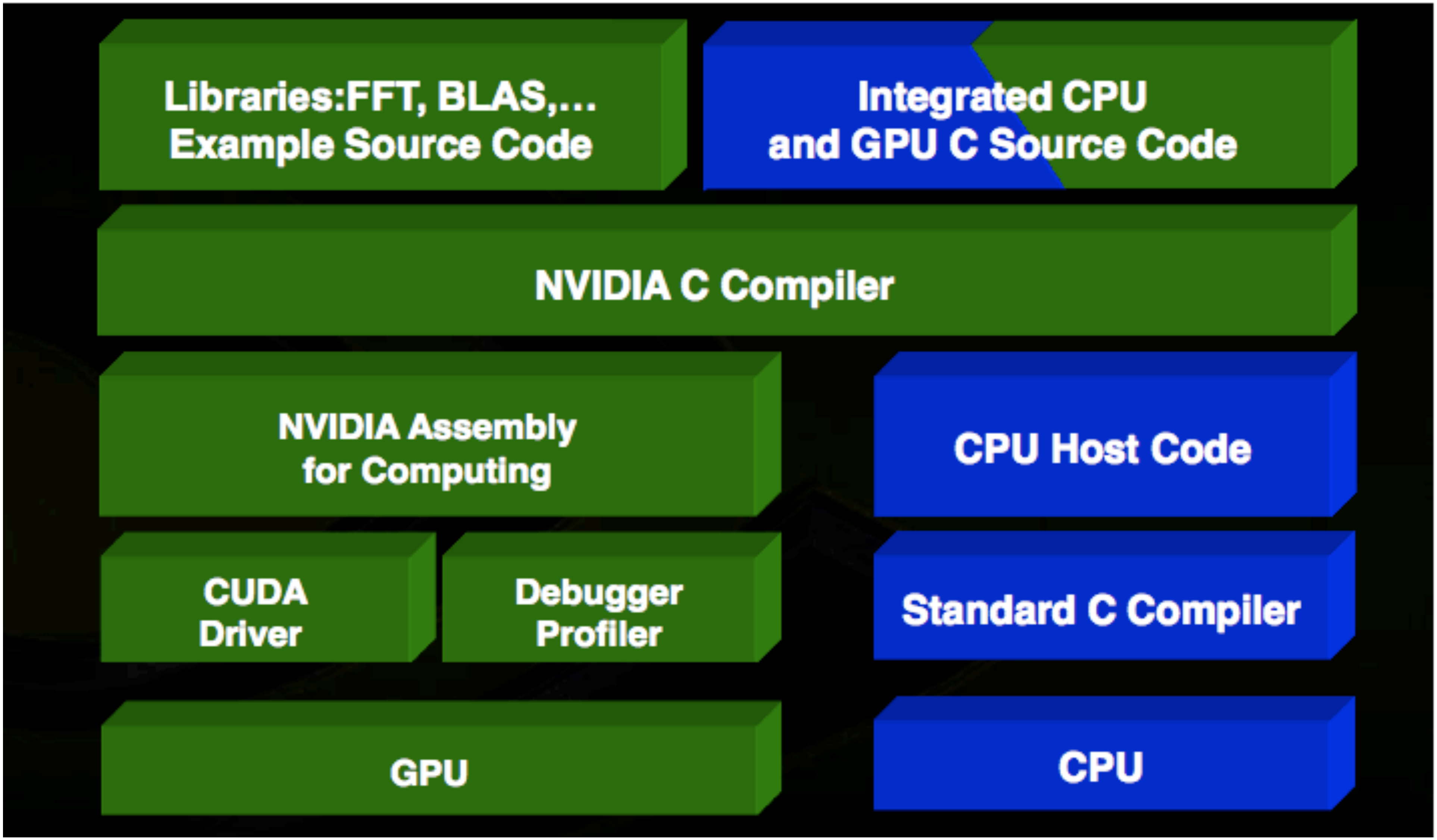
- Runtime API
  - Memory, symbol, execution management

```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

- Function launch

```
// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Software Developer's Kit (SDK)



Slide source: Austin Robison (NVIDIA)

NVCC Compiler's Role: Partition Code and Compile for Device

mycode.cu

```
int main_data;
__shared__  int sdata;

Main() { }
__host__  hfunc () {
   int hdata;
 <<<gfunc(g,b,m)>>>();
}

__global__  gfunc() {
   int gdata;
}

__device__  dfunc() {
   int ddata;
}
```

Host Only

Interface

Device Only

Compiled by native compiler: gcc, icc, cc

```
int main_data;

Main() {}
__host__  hfunc () {
   int hdata;
<<<gfunc(g,b,m)>>>
();
}
```

Compiled by nvcc compiler

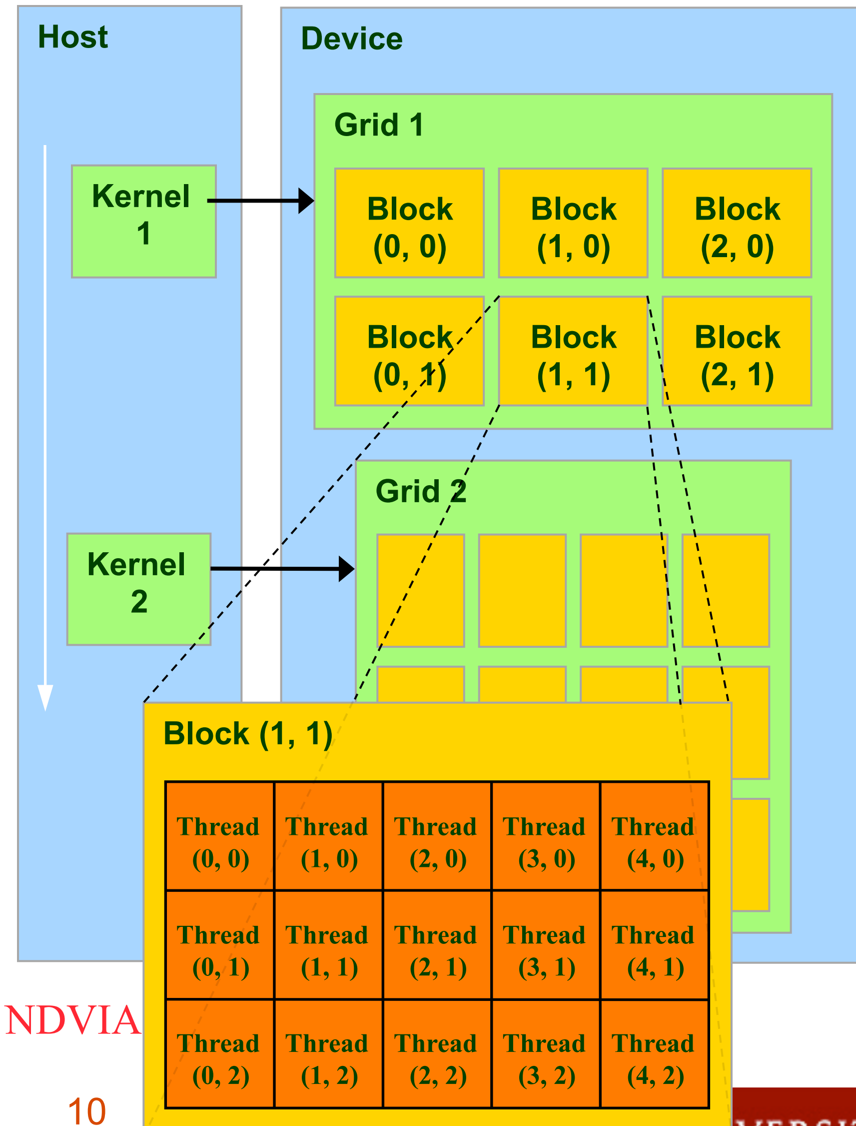
```
__shared__ sdata;

__global__ gfunc() {
   int gdata;
}

__device__ dfunc() {
   int ddata;
}
```

CUDA Programming Model:
A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

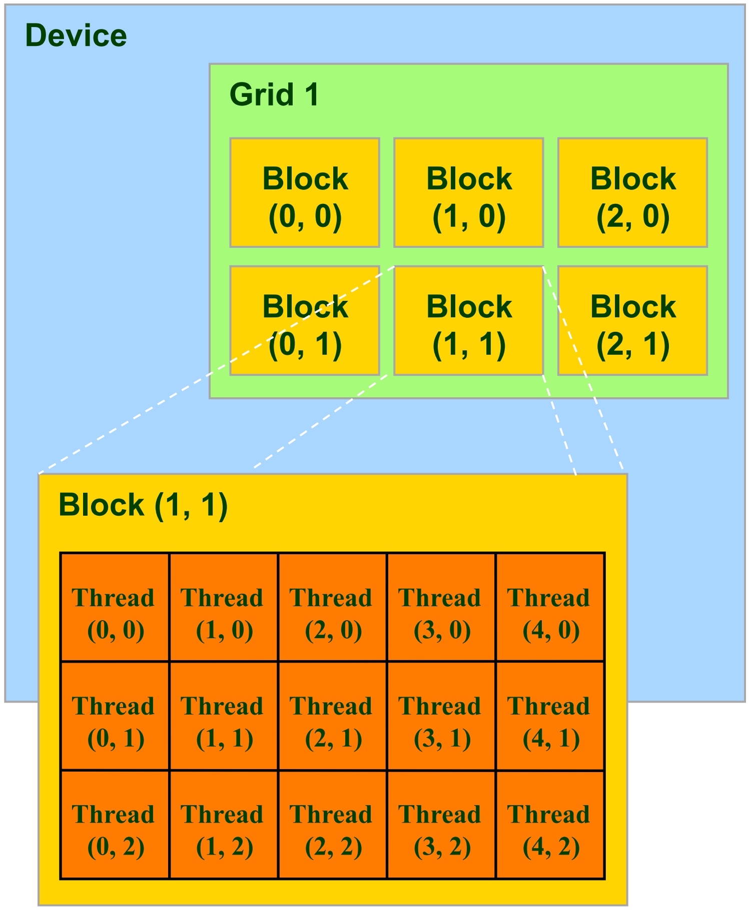
- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared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



Courtesy: NDVIA

Block and Thread IDs

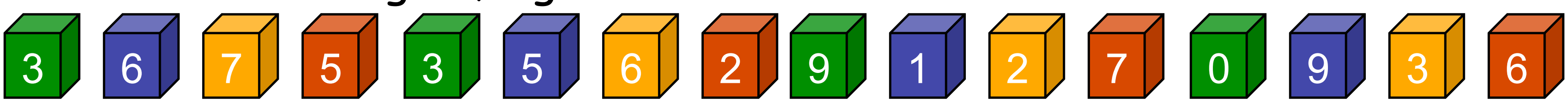
- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Courtesy: NDVIA

Simple working code example

- Goal for this example:
  - Really simple but illustrative of key concepts
  - Fits in one file with simple compile command
  - Can absorb during lecture
- What does it do?
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does "6" appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid



threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

} Known as a cyclic data distribution

CUDA Pseudo-Code

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

GLOBAL FUNCTION:

Thread scans subset of array elements

Call ***device*** function to compare with "6"

Compute local result

DEVICE FUNCTION:

Compare current element and "6"

Return 1 if same, else 0

Main Program: Preliminaries

MAIN PROGRAM:

Initialization

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  …
}
```

Main Program: Invoke Global Function

MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute
        (int *in_arr, int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  /* initialization */ …
  outer_compute(in_array, out_array);
   …
}
```

Main Program: Calculate Output & Print Result

MAIN PROGRAM:

Initialization (OMIT)

- Allocate memory on host for input and output
- Assign random numbers to input array

Call ***host*** function

Calculate final output from per-thread output

Print result

```
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute
        (int *in_arr, int *out_arr);
int main(int argc, char **argv)
{
  int *in_array, *out_array;
  int sum = 0;
  /* initialization */ …
  outer_compute(in_array, out_array);
  for (int i=0; i<BLOCKSIZE; i++) {
    sum+=out_array[i];
  }
  printf ("Result = %d\n",sum);
}
```

Host Function: Preliminaries & Allocation

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int *h_in_array, int *h_out_array) {

  int *d_in_array, *d_out_array;

   cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));

  cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));

  ...
}
```

Host Function: Copy Data To/From Host

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int *h_in_array, int *h_out_array) {
 int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));
 cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));
 cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),
        cudaMemcpyHostToDevice);
 … do computation ...
 cudaMemcpy(h_out_array,d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);
}
```

Host Function: Setup & Call Global Function

HOST FUNCTION:

Allocate memory on device for copy of ***input*** and ***output***

Copy input to ***device***

Set up grid/block

Call ***global*** function

Copy ***device*** output to host

```
__host__ void outer_compute (int
*h_in_array, int *h_out_array) {
 int *d_in_array, *d_out_array;

  cudaMalloc((void **) &d_in_array,
        SIZE*sizeof(int));
 cudaMalloc((void **) &d_out_array,
        BLOCKSIZE*sizeof(int));
 cudaMemcpy(d_in_array, h_in_array,
        SIZE*sizeof(int),
        cudaMemcpyHostToDevice);
 msize = (SIZE+BLOCKSIZE) *
        sizeof (int);
 compute<<<1,BLOCKSIZE,msize)>>>
        (d_in_array, d_out_array);
 cudaMemcpy(h_out_array, d_out_array,
        BLOCKSIZE*sizeof(int),
        cudaMemcpyDeviceToHost);
}
```

Global Function

GLOBAL FUNCTION:

Thread scans subset of array elements

Call *device* function to compare with "6"

Compute local result

```
__global__ void compute(int *d_in,int *d_out) {
  d_out[threadIdx.x] = 0;
  for (int i=0; i<SIZE/BLOCKSIZE; i++)
  {
     int val = d_in[i*BLOCKSIZE + threadIdx.x];
     d_out[threadIdx.x] += compare(val, 6);
  }
}
```

Device Function

DEVICE FUNCTION:

Compare current element and "6"

Return 1 if same, else 0

```
__device__ int
  compare(int a, int b) {
  if (a == b) return 1;
  return 0;
}
```

Reductions

- This type of computation is called a ***parallel reduction***
  - Operation is applied to large data structure
  - Computed result represents the aggregate solution across the large data structure
  - Large data structure → computed result (perhaps single number) **[dimensionality reduced]**
- Why might parallel reductions be well-suited to GPUs?
- What if we tried to compute the final sum on the GPUs? (next class and assignment)

Standard Parallel Construct

- Sometimes called "embarassingly parallel" or "pleasingly parallel"
- Each thread is completely independent of the others
- Final result copied to CPU
- Another example, adding two matrices:
  - A more careful examination of decomposing computation into grids and thread blocks

Another Example: Adding Two Matrices

CPU C program

```
void add_matrix_cpu(float *a, float *b,
  float *c, int N)
{
int i, j, index;
for (i=0;i<N;i++) {
    for (j=0;j<N;j++) {
        index =i+j*N;
        c[index]=a[index]+b[index];
     }
  }
}

void main() {
    .....
    add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu(float *a,
  float *b, float *c, intN)
{
    int i =blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index =i+j*N;
    if( i <N && j <N)
      c[index]=a[index]+b[index];
}

void main() {
    dim3 dimBlock(blocksize,blocksize);
    dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);

   add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b
   ,c,N);
}
```

Example source: Austin Robison, NVIDIA

Closer Inspection of Computation and Data Partitioning

- Define 2-d set of blocks, and 2-d set of threads per block

```
dim3 dimBlock(blocksize,blocksize);
dim3 dimGrid(N/dimBlock.x,N/dimBlock.y);
```

- Each thread identifies what element of the matrix it operates on

```
int i=blockIdx.x*blockDim.x+threadIdx.x;
int j=blockIdx.y*blockDim.y+threadIdx.y;
int index =i+j*N;
if( i <N && j <N)
  c[index]=a[index]+b[index];
```

Summary of Lecture

- Introduction to CUDA
- Essentially, a few extensions to C + API supporting heterogeneous data-parallel CPU+GPU execution
  - Computation partitioning
  - Data partititioning (parts of this implied by decomposition into threads)
  - Data organization and management
  - Concurrency management
- Compiler nvcc takes as input a .cu program and produces
  - C Code for host processor (CPU), compiled by native C compiler
  - Code for device processor (GPU), compiled by nvcc compiler
- Two examples
  - Parallel reduction
  - Embarassingly/Pleasingly parallel computation

Next Week

- A few more details to prepare you for your first assignment
  - More on synchronization for reductions
  - More on decomposing into grids and thread blocks
  - More on run-time library
    - Especially constructs to test for correct execution
  - A little on debugging