Parallel Programming & Cluster Computing

GPGPU: Number Crunching in Your Graphics Card

Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College
Andrew Fitz Gibbon, Earlham College
Josh Alexander, University of Oklahoma

Oklahoma Supercomputing Symposium 2009
University of Oklahoma, Tuesday October 6 2009

Outline

- What is GPGPU?
- GPU Programming
- Digging Deeper: CUDA on NVIDIA
- CUDA Thread Hierarchy and Memory Hierarchy
- CUDA Example: Matrix-Matrix Multiply

What is GPGPU?

Accelerators

No, not this ....

http://gizmodo.com/5032891/nissans-eco-gas-pedal-fights-back-to-help-you-save-gas

Accelerators

- In HPC, an accelerator is hardware component whose role is to speed up some aspect of the computing workload.
- In the olden days (1980s), supercomputers sometimes had ***array processors***, which did vector operations on arrays, and PCs sometimes had ***floating point accelerators***: little chips that did the floating point calculations in hardware rather than software.
- More recently, ***Field Programmable Gate Arrays*** (FPGAs) allow reprogramming deep into the hardware.

Why Accelerators are Good

Accelerators are good because:

- they make your code run faster.

Why Accelerators are Bad

Accelerators are bad because:

- they're expensive;
- they're hard to program;
- your code on them isn't portable to other accelerators, so the labor you invest in programming them has a very short half-life.

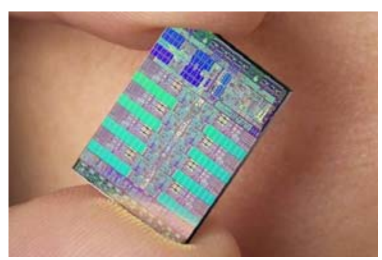
The King of the Accelerators

The undisputed champion of accelerators is:

the **graphics processing unit.**

http://www.amd.com/us-en/assets/content_type/DigitalMedia/46928a_01_ATI-FirePro_V8700_angled_low_res.gif

http://images.nvidia.com/products/quadro_fx_5800/Quadro_FX5800_low_3qtr.png



http://www.gamecyte.com/wp-content/uploads/2009/01/ibm-sony-toshiba-cell.jpg

Why GPU?

- ***Graphics Processing Units*** (GPUs) were originally designed to accelerate graphics tasks like image rendering.
- They became very very popular with videogamers, because they've produced better and better images, and lightning fast.
- And, prices have been extremely good, ranging from three figures at the low end to four figures at the high end.

GPUs are Popular

- Chips are expensive to design (hundreds of millions of $$$), expensive to build the factory for (billions of $$$), but cheap to produce.
- In 2006 – 2007, GPUs sold at a rate of about 80 million cards per year, generating about $20 billion per year in revenue.
  http://www.xbitlabs.com/news/video/display/20080404234228_Shipments_of_Discrete_Graphics_Cards_on_the_Rise_but_Prices_Down_Jon_Peddie_Research.html
- This means that the GPU companies have been able to recoup the huge fix costs.

GPU Do Arithmetic

- GPUs mostly do stuff like rendering images.
- This is done through mostly floating point arithmetic – the same stuff people use supercomputing for!

GPU Programming

Hard to Program?

- In the olden days – that is, until just the last few years – programming GPUs meant either:
  - using a graphics standard like OpenGL (which is mostly meant for rendering), or
  - getting fairly deep into the graphics rendering pipeline.
- To use a GPU to do general purpose number crunching, you had to make your number crunching pretend to be graphics.
- This was hard. So most people didn't bother.

Easy to Program?

More recently, GPU manufacturers have worked hard to make GPUs easier to use for general purpose computing.

This is known as ***General Purpose Graphics Processing Units***.

How to Program a GPU

- Proprietary programming language or extensions
  - NVIDIA: CUDA (C/C++)
  - AMD/ATI: StreamSDK/Brook+ (C/C++)
- OpenCL (Open Computing Language): an industry standard for doing number crunching on GPUs.
- Portland Group Fortran and C compilers with accelerator directives.

NVIDIA CUDA

- NVIDIA proprietary
- Formerly known as "Compute Unified Device Architecture"
- Extensions to C to allow better control of GPU capabilities
- Modest extensions but major rewriting of the code
- Portland Group Inc (PGI) recently announced a Fortran version available in their compiler

CUDA Example Part 1

```
// example1.cpp : Defines the entry point for the console applicati
    on.
//

#include "stdafx.h"

#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx] * a[idx];
}
```

http://llpanorama.wordpress.com/2008/05/21/my-first-cuda-program/

CUDA Example Part 2

```
// main routine that executes on the host
int main(void)
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
}
```

AMD/ATI Brook+

- AMD/ATI proprietary
- Formerly known as "Close to Metal" (CTM)
- Extensions to C to allow better control of GPU capabilities
- No Fortran version available

Brook+ Example Part 1

```
float4 matmult_kernel (int y, int x, int k,
                       float4 M0[], float4 M1[])
{
    float4 total = 0;
    for (int c = 0; c < k / 4; c++)
    {
        total += M0[y][c] * M1[x][c];
    }
    return total;
}
```

http://developer.amd.com/gpu_assets/Stream_Computing_Overview.pdf

Brook+ Example Part 2

```
void matmult (float4 A[], float4 B’[], float4 C[])
{
    for (int i = 0; i < n; i++)
    {
        for (j = 0; j < m / 4; j+)
        {
            launch_thread{
                C[i][j] =
                    matmult_kernel(j, i, k, A, B’);}
        }
    }
    sync_threads{}
}
```

OpenCL

- Open Computing Language
- Open standard developed by the Khronos Group, which is a consortium of many companies (including NVIDIA, AMD and Intel, but also lots of others)
- Initial version of OpenCL standard released in Dec 2008.
- Many companies will create their own implementations.
- Apple expects to be first to market, with an OpenCL implementation included in Mac OS X v10.6 ("Snow Leopard"), expected in 2009.

OpenCL Example Part 1

```
// create a compute context with GPU device
context = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL, NULL,
   NULL);
// create a work-queue
queue = clCreateWorkQueue(context, NULL, NULL, 0);
// allocate the buffer memory objects
memobjs[0] =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   sizeof(float)*2*num_entries, srcA);
memobjs[1] =
    clCreateBuffer(context, CL_MEM_READ_WRITE,
                   sizeof(float)*2*num_entries, NULL);
// create the compute program
program =
    clCreateProgramFromSource(context, 1, &fft1D_1024_kernel_src, NULL);
// build the compute program executable
clBuildProgramExecutable(program, false, NULL, NULL);
// create the compute kernel
kernel = clCreateKernel(program, "fft1D_1024");
```

OpenCL Example Part 2

```
// create N-D range object with work-item dimensions
global_work_size[0] = n;
local_work_size[0] = 64;
range = clCreateNDRangeContainer(context, 0, 1, global_work_size,
   local_work_size);
// set the args values
clSetKernelArg(kernel, 0, (void *)&memobjs[0], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 1, (void *)&memobjs[1], sizeof(cl_mem), NULL);
clSetKernelArg(kernel, 2, NULL,
    sizeof(float)*(local_work_size[0]+1)*16, NULL);
clSetKernelArg(kernel, 3, NULL,
    sizeof(float)*(local_work_size[0]+1)*16, NULL);
// execute kernel
clExecuteKernel(queue, kernel, NULL, range, NULL, 0, NULL);
```

OpenCL Example Part 3

```
// This kernel computes FFT of length 1024. The 1024 length FFT
// is decomposed into calls to a radix 16 function, another
// radix 16 function and then a radix 4 function
kernel void fft1D_1024 (
    global float2 *in, __global float2 *out,
    local float *sMemx, __local float *sMemy)
{
    int tid = get_local_id(0);
    int blockIdx = get_group_id(0) * 1024 + tid;
    float2 data[16];
    // starting index of data to/from global memory
    in = in + blockIdx;
    out = out + blockIdx;
    globalLoads(data, in, 64); // coalesced global reads
```

OpenCL Example Part 4

```
    fftRadix16Pass(data);      // in-place radix-16 pass
    twiddleFactorMul(data, tid, 1024, 0);
    // local shuffle using local memory
    localShuffle(data, sMemx, sMemy, tid,
        (((tid & 15) * 65) + (tid >> 4)));
    fftRadix16Pass(data);               // in-place radix-16 pass
    twiddleFactorMul(data, tid, 64, 4); // twiddle factor multiplication
    localShuffle(data, sMemx, sMemy, tid,
        (((tid >> 4) * 64) + (tid & 15)));
    // four radix-4 function calls
    fftRadix4Pass(data);
    fftRadix4Pass(data + 4);
    fftRadix4Pass(data + 8);
    fftRadix4Pass(data + 12);
    // coalesced global writes
    globalStores(data, out, 64);
}
```

Portland Group Accelerator Directives

- Proprietary directives in Fortran and C
- Similar to OpenMP in structure
- Currently in beta release
- If the compiler doesn't understand these directives, it ignores them, so the same code can work with an accelerator or without, and with the PGI compilers or other compilers.
- In principle, this will be able to work on a variety of accelerators, but the first instance will be NVIDIA; PGI recently announced a deal with AMD/ATI.
- The directives tell the compiler what parts of the code happen in the accelerator; the rest happens in the regular hardware.

PGI Accelerator Example

```
!$acc region
    do k = 1,n1
        do i = 1,n3
            c(i,k) = 0.0
            do j = 1,n2
                c(i,k) = c(i,k) +
&                        a(i,j) * b(j,k)
            enddo
        enddo
    enddo
!$acc end region
```

http://www.pgroup.com/resources/accel.htm

Digging Deeper: CUDA on NVIDIA

NVIDIA Tesla

- NVIDIA now offers a GPU platform named Tesla.
- It consists of their highest end graphics card, minus the video out connector.
- This cuts the cost of the GPU card roughly in half: Quadro FX 5800 is ~$3000, Tesla C1060 is ~$1500.

http://images.nvidia.com/products/tesla_c1060/Tesla_c1060_3qtr_low.png

NVIDIA Tesla C1060 Card Specs

- 240 GPU cores
- 1.296 GHz
- Single precision floating point performance: 933 GFLOPs (3 single precision flops per clock per core)
- Double precision floating point performance: 78 GFLOPs (0.25 double precision flops per clock per core)
- Internal RAM: 4 GB
- Internal RAM speed: 102 GB/sec (compared 21-25 GB/sec for regular RAM)
- Has to be plugged into a PCIe slot (at most 8 GB/sec)

NVIDIA Tesla S1070 Server Specs

- 4 C1060 cards inside a 1U server (looks like a Sooner node)
- Available in both 1.296 GHz and 1.44 GHz
- Single Precision (SP) floating point performance:
  3732 GFLOPs (1.296 GHz) or 4147 GFLOPs (1.44 GHz)
- Double Precision (DP) floating point performance:
  311 GFLOPs (1.296 GHz) or 345 GFLOPs (1.44 GHz)
- Internal RAM: 16 GB total (4 GB per GPU card)
- Internal RAM speed: 408 GB/sec aggregate
- Has to be plugged into two PCIe slots (at most 16 GB/sec)

Compare x86 vs S1070

Let's compare the best dual socket x86 server today vs S1070.

	Dual socket, Intel 2.66 hex core	**NVIDIA Tesla S1070**
Peak DP FLOPs	128 GFLOPs DP	345 GFLOPs DP (2.7x)
Peak SP FLOPS	256 GFLOPs SP	4147 GFLOPs SP (16.2x)
Peak RAM BW	17 GB/sec	408 GB/sec (24x)
Peak PCIe BW	N/A	16 GB/sec
Needs x86 server to attach to?	No	Yes
Power/Heat	~400 W	~800 W + ~400 W (3x)
Code portable?	Yes	No (CUDA) Yes (PGI, OpenCL)

Compare x86 vs S1070

Here are some interesting measures:

	Dual socket, Intel 2.66 hex core	NVIDIA Tesla S1070
DP GFLOPs/Watt	~0.3 GFLOPs/Watt	~0.3 GFLOPs/Watt (same)
SP GFLOPS/Watt	0.64 GFLOPs/Watt	~3.5 GFLOPs (~5x)
DP GFLOPs/sq ft	~340 GFLOPs/sq ft	~460 GFLOPs/sq ft (1.3x)
SP GFLOPs/sq ft	~680 GFLOPs/sq ft	~5500 GFLOPs/sq ft (8x)
Racks per PFLOP DP	244 racks/PFLOP DP	181 racks/PFLOP (3/4) DP
Racks per PFLOP SP	122 racks/PFLOP SP	15 racks/PFLOP (1/8) SP

OU's Sooner is 65 TFLOPs SP, which is **1 rack** of S1070.

What Are the Downsides?

- You have to rewrite your code into CUDA or OpenCL or PGI accelerator directives.
  - CUDA: Proprietary, but maybe portable soon
  - OpenCL: portable but cumbersome
  - PGI accelerator directives: not clear whether you can have most of the code live inside the GPUs.

Programming for Performance

The biggest single performance bottleneck on GPU cards today is the PCIe slot:

- PCIe 2.0 x16: 8 GB/sec
- 1600 MHz Front Side Bus: 25 GB/sec
- GDDR3 GPU card RAM: 102 GB/sec per card

Your goal:

- At startup, move the data from x86 server RAM into GPU RAM.
- Do almost all the work inside the GPU.
- Use the x86 server only for I/O and message passing, to minimize the amount of data moved through the PCIe slot.

Does CUDA Help?

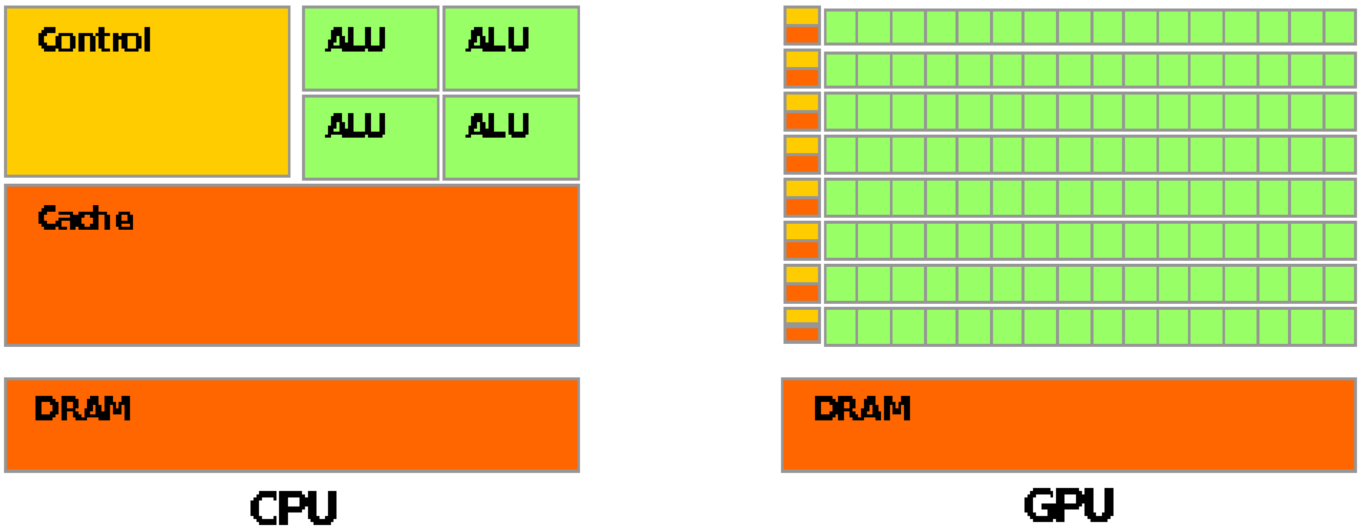
Example Applications	URL	Speedup
Seismic Database	http://www.headwave.com	66x – 100x
Mobile Phone Antenna Simulation	http://www.accelware.com	45x
Molecular Dynamics	http://www.ks.uiuc.edu/Research/vmd	21x – 100x
Neuron Simulation	http://www.evolvedmachines.com	100x
MRI Processing	http://bic-test.beckman.uiuc.edu	245x – 415x
Atmospheric Cloud Simulation	http://www.cs.clemson.edu/~jesteel/clouds.html	50x

http://www.nvidia.com/object/IO_43499.html

CUDA Thread Hierarchy and Memory Hierarchy

Some of these slides provided by Paul Gray, University of Northern Iowa

CPU vs GPU Layout



Source: NVIDIA CUDA Programming Guide

Buzzword: Kernel

In CUDA, a ***kernel*** is code (typically a function) that can be run inside the GPU.

Typically, the kernel code operates in lock-step on the stream processors inside the GPU.

Buzzword: Thread

In CUDA, a ***thread*** is an execution of a kernel with a given index.

Each thread uses its index to access a specific subset of the elements of a target array, such that the collection of all threads cooperatively processes the entire data set.

So these are very much like threads in the OpenMP or pthreads sense – they even have shared variables and private variables.

Buzzword: Block

In CUDA, a ***block*** is a group of threads.

- Just like OpenMP threads, these could execute concurrently or independently, and in no particular order.
- Threads can be coordinated somewhat, using the _syncthreads() function as a barrier, making all threads stop at a certain point in the kernel before moving on en mass. (This is like what happens at the end of an OpenMP loop.)

Buzzword: Grid

In CUDA, a ***grid*** is a group of (thread) blocks, with no synchronization at all among the blocks.

NVIDIA GPU Hierarchy

- ***Grids*** map to GPUs
- ***Blocks*** map to the MultiProcessors (MP)
  - Blocks are never split across MPs, but an MP can have multiple blocks
- ***Threads*** map to Stream Processors (SP)
- ***Warps*** are groups of (32) threads that execute simultaneously

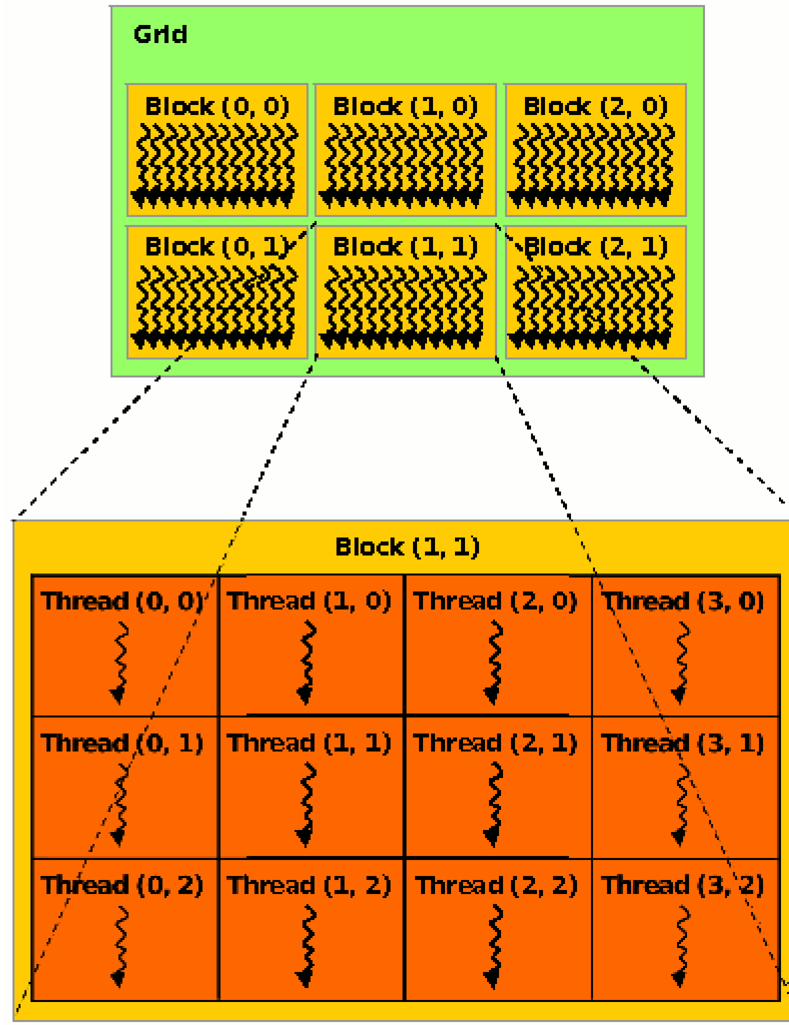


Image Source: NVIDIA CUDA Programming Guide

CUDA 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.

So, you can express your collection of blocks, and your collection of threads within a block, as a 1D array, a 2D array or a 3D array.

These can be helpful when thinking of your data as 2D or 3D.

`__global__` Keyword

- In CUDA, if a function is declared with the `__global__` keyword, that means that it's intended to be executed inside the GPU.
- In CUDA, the term for the GPU is ***device***, and the term for the x86 server is ***host***.
- So, a kernel runs on a device, while the main function and so on run on the host.
- Note that a host can play host to multiple devices; for example, an S1070 server contains 4 C1060 GPU cards, and if a single host has two PCIe slots, then both of the PCIe plugs of the S1070 can be plugged into that same host.

Copying Data from Host to Device

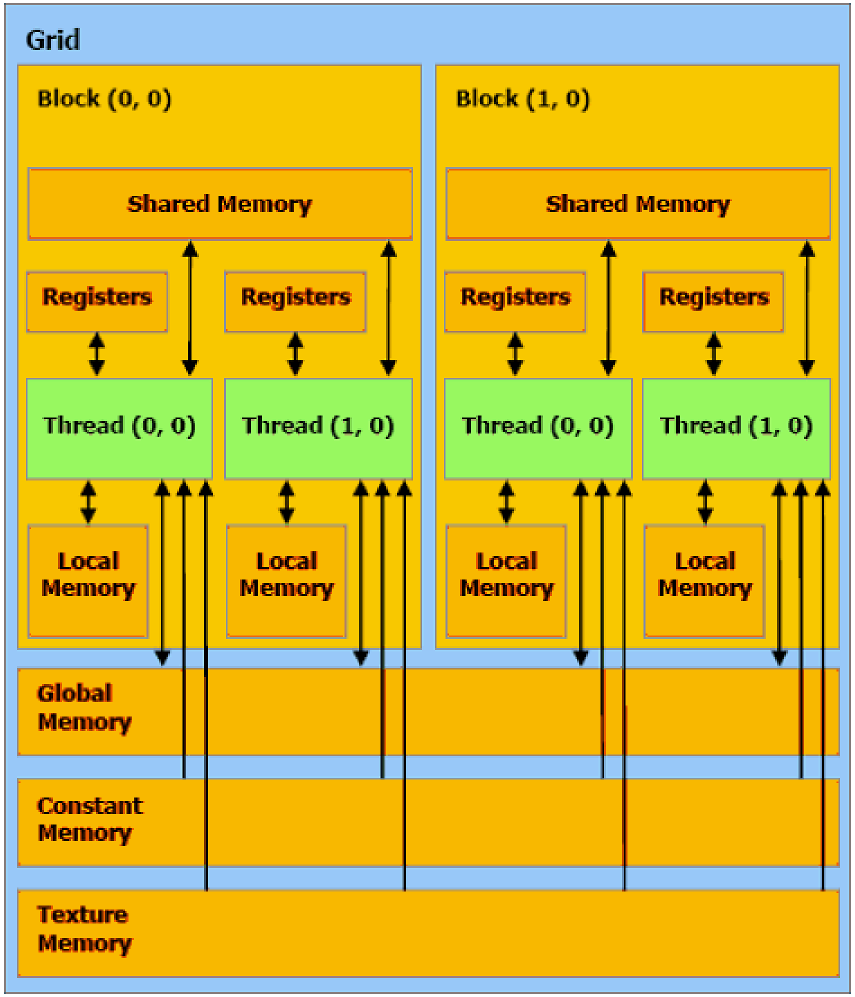
If data need to move from the host (where presumably the data are initially input or generated), then a copy has to exist in both places.

Typically, what's copied are arrays, though of course you can also copy a scalar (the address of which is treated as an array of length 1).

CUDA Memory Hierarchy #1

CUDA has a hierarchy of several kinds of memory:

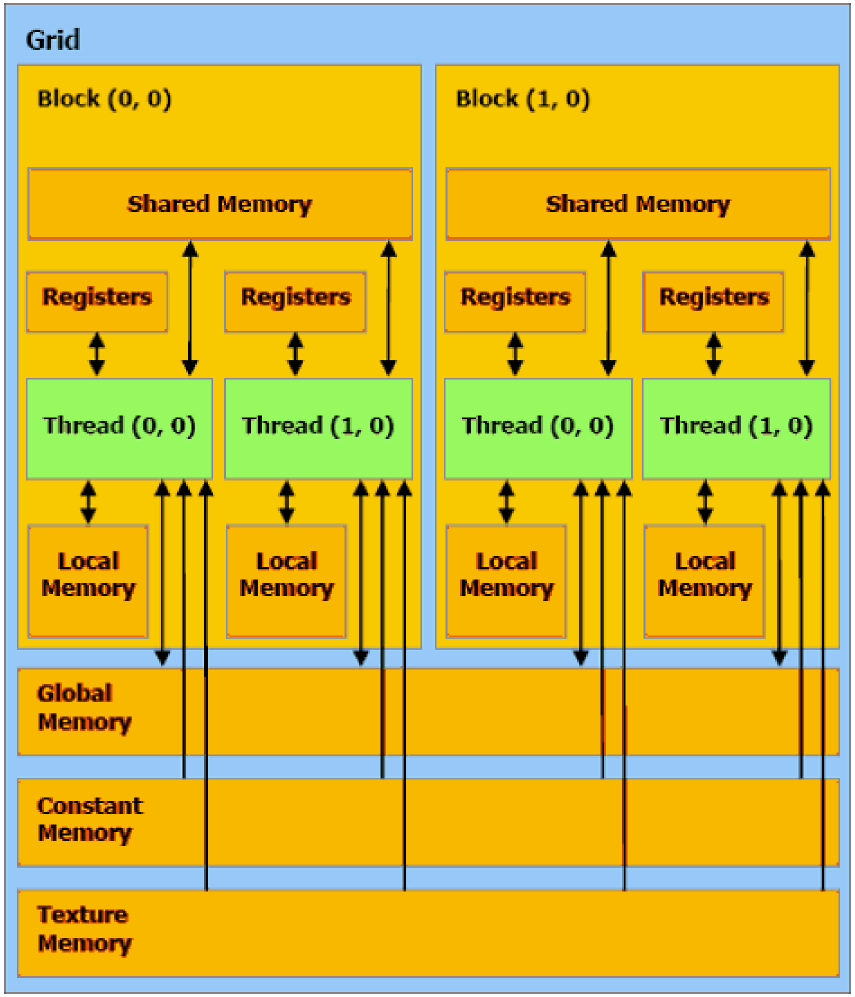
- Host memory (x86 server)
- Device memory (GPU)
  - ***Global***: visible to all threads in all blocks – largest, slowest
  - ***Shared***: visible to all threads in a particular block – medium size, medium speed
  - ***Local***: visible only to a particular thread – smallest, fastest



CUDA Memory Hierarchy #2

CUDA has a hierarchy of several kinds of memory:

- Host memory (x86 server)
- Device memory (GPU)
  - ***Constant***: visible to all threads in all blocks; read only
  - ***Texture***: visible to all threads in all blocks; read only



CUDA Example: Matrix-Matrix Multiply

http://developer.download.nvidia.com/compute/cuda/sdk/website/Linear_Algebra.html#matrixMul

Matrix-Matrix Multiply Main Part 1

```
float* host_A;
float* host_B;
float* host_B;
float* device_A;
float* device_B;
float* device_C;

host_A = (float*) malloc(mem_size_A);
host_B = (float*) malloc(mem_size_B);
host_C = (float*) malloc(mem_size_C);

cudaMalloc((void**) &device_A, mem_size_A);
cudaMalloc((void**) &device_B, mem_size_B);
cudamalloc((void**) &device_C, mem_size_C);

// Set up the initial values of A and B here.

// Henry says: I've oversimplified this a bit from
// the original example code.
```

Matrix-Matrix Multiply Main Part 2

```
// copy host memory to device
cudaMemcpy(device_A, host_A, mem_size_A,
           cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, mem_size_B,
           cudaMemcpyHostToDevice);
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

// execute the kernel
matrixMul<<< grid, threads >>>(device_C,
                               device_A, device_B, WA, WB);

// copy result from device to host
cudaMemcpy(host_C, device_C, mem_size_C,
           cudaMemcpyDeviceToHost);
```

Matrix Matrix Multiply Kernel Part 1

```
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    float Csub = 0;
```

Matrix Matrix Multiply Kernel Part 2

```
// Loop over all the sub-matrices of A and B
// required to compute the block sub-matrix
for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {

    // Declaration of the shared memory array As used to
    // store the sub-matrix of A
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

    // Declaration of the shared memory array Bs used to
    // store the sub-matrix of B
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    // Load the matrices from device memory
    // to shared memory; each thread loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    __syncthreads();
```

Matrix Matrix Multiply Kernel Part 3

```
        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write the block sub-matrix to device memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Would We Really Do It This Way?

We wouldn’t really do matrix-matrix multiply this way.

NVIDIA has developed a CUDA implementation of the BLAS libraries, which include a highly tuned matrix-matrix multiply routine.

(We’ll learn about BLAS next time.)

There’s also a CUDA FFT library, if your code needs Fast Fourier Transforms.

Thanks for your attention!

Questions?