Introduction to Parallel Programming & Cluster Computing

GPGPU: Number Crunching in Your Graphics Card

Josh Alexander, Henry Neeman - University of Oklahoma
Ivan Babic, Mobeen Ludin, Kristin Muterspaw, Charlie Peck - Earlham College
Michial Green, Tom Murphy - Contra Costa College

OSCER/OU - August, 2012
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

- 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 (some);
- they’re hard to program (all, at least for now);
- your code on them may not be 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**.

[Image 1](http://www.amd.com/us-en/assets/content_type/DigitalMedia/46928a_01_ATI-FirePro_V8700_angled_low_res.gif)

[Image 2](http://images.nvidia.com/products/quadro_fx_5800/Quadro_FX5800_low_3qtr.png)


[Image 4](http://www.overclockers.ua/news/cpu/106612-Knights-Ferry.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 produce 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.
- For example, in 2006 – 2007, GPUs sold at a rate of about 80 million cards per year, generating about $20 billion per year in revenue.


- This means that the GPU companies have been able to recoup the huge fixed costs.
GPUs Do Arithmetic

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

Source: NVIDIA
What’s Different?

Figure: nvidia.com

NCSI Intro Parallel: GPGPU
August, 2012
Memory Hierarchy? You Bet!

Host

DRAM

Chipset

CPU

Device

DRAM

Global Memory

Local Memory

GPU

Multiprocessor

Multiprocessor

Multiprocessor

Registers

Shared Memory

Source: NVIDIA

NCSI Intro Parallel: GPGPU
August, 2012
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 Inc (PGI) Fortran and C compilers with accelerator directives; PGI CUDA Fortran (Fortran 90 equivalent of NVIDIA’s CUDA C).
- OpenMP version 4.0 may include directives for accelerators.
- Others are popping up or in development now ….
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) has released a Fortran implementation of CUDA available in their Fortran compiler.
```c
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
    for (i=0, i<N; i++) a_h[i] = 100.f + i;
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```

Source: NVIDIA
NCSI Intro Parallel: GPGPU
August, 2012

Friday, August 3, 12
CUDA Programming

- Create kernel that will execute on the card
- Allocate memory on the CPU and populate
- Allocate memory on the card and copy data from CPU to it
- Determine how the kernel will lay down on the card
- Execute the kernel on the card
- Copy the results from the card’s memory to the CPU’s
CUDA Example Part 1

// example1.cpp : Defines the entry point for the console application.

#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);
}
OpenMP 4.0 Accelerator Directives

- OpenMP’s 4.0 standard is very much in discussion (and flux).
- It *may* end up with accelerator directives.
- It’s too soon to say what the details will be, if it happens at all.
- But, if it happens, then codes amenable to accelerator directives will be able to get substantial speedups with very modest coding effort.
OpenMP 4.0 Accelerator Example

```fortran
!$omp 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
$omp end acc_region
```

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

http://www.cse.scitech.ac.uk/events/GPU_2010/12_Hart.pdf
Digging Deeper:
CUDA on NVIDIA
NVIDIA Tesla C2050 Card Specs

- 448 GPU cores
- 1.15 GHz
- Single precision floating point performance: 1030.4 GFLOPs (2 single precision flops per clock per core)
- Double precision floating point performance: 515.2 GFLOPs (1 double precision flop per clock per core)
- Internal RAM: 3 GB DDR5
- Internal RAM speed: 144 GB/sec (compared 21-25 GB/sec for regular RAM)
- Has to be plugged into a PCIe slot (at most 8 GB/sec per GPU card)
NVIDIA Tesla S2050 Server Specs

- 4 C2050 cards inside a 1U server (looks like a Sooner node)
- 1.15 GHz
- Single Precision (SP) floating point performance: 4121.6 GFLOPs
- Double Precision (DP) floating point performance: 2060.8 GFLOPs
- Internal RAM: 12 GB total (3 GB per GPU card)
- Internal RAM speed: 576 GB/sec aggregate
- Has to be plugged into two PCIe slots (at most 16 GB/sec for 4 GPU cards)
## Compare x86 vs S2050

Let’s compare the best dual socket x86 server today vs S2050.

<table>
<thead>
<tr>
<th></th>
<th>Dual socket, AMD 2.3 GHz 12-core</th>
<th>NVIDIA Tesla S2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak DP FLOPs</td>
<td>220.8 GFLOPs DP</td>
<td>2060.8 GFLOPs DP (9.3x)</td>
</tr>
<tr>
<td>Peak SP FLOPS</td>
<td>441.6 GFLOPs SP</td>
<td>4121.6 GFLOPs SP (9.3x)</td>
</tr>
<tr>
<td>Peak RAM BW</td>
<td>25 GB/sec</td>
<td>576 GB/sec (23x)</td>
</tr>
<tr>
<td>Peak PCIe BW</td>
<td>N/A</td>
<td>16 GB/sec</td>
</tr>
<tr>
<td>Needs x86 server to attach to?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Power/Heat</td>
<td>~450 W</td>
<td>~900 W + ~400 W (~2.9x)</td>
</tr>
<tr>
<td>Code portable?</td>
<td>Yes</td>
<td>No (CUDA)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes (PGI, OpenCL)</td>
</tr>
</tbody>
</table>
## Compare x86 vs S2050

Here are some interesting measures:

<table>
<thead>
<tr>
<th></th>
<th>Dual socket, AMD 2.3 GHz 12-core</th>
<th>NVIDIA Tesla S2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP GFLOPs/Watt</td>
<td>~0.5 GFLOPs/Watt</td>
<td>~1.6 GFLOPs/Watt (~3x)</td>
</tr>
<tr>
<td>SP GFLOPS/Watt</td>
<td>~1 GFLOPs/Watt</td>
<td>~3.2 GFLOPs/Watt (~3x)</td>
</tr>
<tr>
<td>DP GFLOPs/sq ft</td>
<td>~590 GFLOPs/sq ft</td>
<td>~2750 GFLOPs/sq ft (4.7x)</td>
</tr>
<tr>
<td>SP GFLOPs/sq ft</td>
<td>~1180 GFLOPs/sq ft</td>
<td>~5500 GFLOPs/sq ft (4.7x)</td>
</tr>
<tr>
<td>Racks per PFLOP DP</td>
<td>142 racks/PFLOP DP</td>
<td>32 racks/PFLOP DP (23%)</td>
</tr>
<tr>
<td>Racks per PFLOP SP</td>
<td>71 racks/PFLOP SP</td>
<td>16 racks/PFLOP SP (23%)</td>
</tr>
</tbody>
</table>
What Are the Downsides?

- You have to rewrite your code into CUDA or OpenCL or PGI accelerator directives (or someday maybe OpenMP).
  - 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.

- BUT: Many groups are coming out with GPGPU code development tools that may help a lot, such as:
  - Fortran-to-CUDA-C converter (NCAR)
  - CUDA C automatic optimizer (memory, threading etc)
  - OpenMP-to-CUDA converter
  - CUDA-to-x86 converter (CUDA code on non-CUDA system)
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
- GDDR5 GPU card RAM: 144 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?

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

Under the Hood

10-Series Architecture

- 240 thread processors execute kernel threads
- 30 multiprocessors, each contains
  - 8 thread processors
  - One double-precision unit
- Shared memory enables thread cooperation

Source: NVIDIA

NCSI Intro Parallel: GPGPU
August, 2012
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 masse. (This is like what happens at the end of an OpenMP loop.)
In CUDA, a grid is a group of (thread) blocks, with no synchronization at all among the blocks.
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
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 a 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 S2050 server contains 4 C2050 GPU cards, and if a single host has two PCIe slots, then both of the PCIe plugs of the S2050 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 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
Thanks for your attention!
Questions?