Introduction to GPU Programming

Volodymyr (Vlad) Kindratenko
Innovative Systems Laboratory @ NCSA
Institute for Advanced Computing Applications and Technologies (IACAT)
Tutorial Goals

• Become familiar with NVIDIA GPU architecture
• Become familiar with the NVIDIA GPU application development flow
• Be able to write and run simple NVIDIA GPU kernels in CUDA
• Be aware of performance limiting factors and understand performance tuning strategies
Schedule

• Day 1
  – 2:30-3:45 part I
  – 3:45-4:15 break
  – 4:15-5:30 part II

• Day 2
  – 2:30-3:45 part III
  – 3:45-4:15 break
  – 4:15-5:30 part IV
Part I

• Introduction
• Hands-on: getting started with NCSA GPU cluster
• Hands-on: anatomy of a GPU application
Introduction

• Why use Graphics Processing Units (GPUs) for general-purpose computing

• Modern GPU architecture
  – NVIDIA

• GPU programming overview
  – Libraries
  – CUDA C
  – OpenCL
  – PGI x64+GPU
Why GPUs?
Raw Performance Trends

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt

Graph is courtesy of NVIDIA
Why GPUs?
Memory Bandwidth Trends

Graph is courtesy of NVIDIA

V. Kindratenko, Introduction to GPU Programming (part I), December 2010, The American University in Cairo, Egypt
GPU vs. CPU Silicon Use

Graph is courtesy of NVIDIA

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
NVIDIA GPU Architecture

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of:
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals
  - A multithreaded instruction unit
  - On-chip shared memory
- GDDR3 SDRAM
- PCIe interface

Figure is courtesy of NVIDIA
NVIDIA GeForce9400M G GPU

- 16 streaming processors arranged as 2 streaming multiprocessors
- At 0.8 GHz this provides
  - 54 GFLOPS in single-precision (SP)
- 128-bit interface to off-chip GDDR3 memory
  - 21 GB/s bandwidth

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
NVIDIA Tesla C1060 GPU

- 240 streaming processors arranged as 30 streaming multiprocessors
- At 1.3 GHz this provides
  - 1 TFLOPS SP
  - 86.4 GFLOPS DP
- 512-bit interface to off-chip GDDR3 memory
  - 102 GB/s bandwidth
NVIDIA Tesla S1070 Computing Server

- 4 T10 GPUs
GPU Use/Programming

• GPU libraries
  – NVIDIA’s CUDA BLAS and FFT libraries
  – Many 3rd party libraries

• Low abstraction lightweight GPU programming toolkits
  – CUDA C
  – OpenCL

• High abstraction compiler-based tools
  – PGI x64+GPU
CUDA C APIs

- higher-level API called the **CUDA runtime API**
  - myKernel<<<grid size>>>(args);

- low-level API called the **CUDA driver API**
  - cuModuleLoad( &module, binfile );
  - cuModuleGetFunction( &func, module, "mykernel" );
  - ...
  - cuParamSetv( func, 0, &args, 48 );
  - ...
  - cuParamsetSize( func, 48 );
  - cuFuncSetBlockSize( func, ts[0], ts[1], 1 );
  - cuLaunchGrid( func, gs[0], gs[1] );
Getting Started with NCSA GPU Cluster

• Cluster architecture overview
• How to login and check out a node
• How to compile and run an existing application
NCSA AC GPU Cluster
GPU Cluster Architecture

- Servers: 32
  - CPU cores: 128
- Accelerator Units: 32
  - GPUs: 128

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
GPU Cluster Node Architecture

- HP xw9400 workstation
  - 2216 AMD Opteron 2.4 GHz dual socket dual core
  - 8 GB DDR2
  - InfiniBand QDR
- S1070 1U GPU Computing Server
  - 1.3 GHz Tesla T10 processors
  - 4x4 GB GDDR3 SDRAM
Accessing the GPU Cluster

- Use Secure Shell (SSH) client to access AC
  - `ssh USER@ac.ncsa.uiuc.edu` (User: tra1 – tra50; Password: ???)
  - You will see something like this printed out:

```
See machine details and a technical report at:
http://www.ncsa.uiuc.edu/Projects/GPUcluster/
If publishing works supported by AC, you can acknowledge it as an IACAT resource.  http://www.ncsa.illinois.edu/UserInfo/Allocations/Ack.html

All SSH traffic on this system is monitored.  For more information see:
https://bw-wiki.ncsa.uiuc.edu/display/bw/Security+Monitoring+Policy

Machine Description and HOW TO USE.  See:  /usr/local/share/docs/ac.readme
CUDA wrapper readme:  /usr/local/share/docs/cuda_wrapper.readme
These docs also available at:  http://ac.ncsa.uiuc.edu/docs/

########################################################################

Nov 22, 2010
CUDA 3.2 fully deployed.  Release nodes are available here:
An updated SDK is available in /usr/local/cuda/

Questions?  Contact Jeremy Enos jenos@ncsa.uiuc.edu
(for password resets, please contact help@ncsa.uiuc.edu)

11:17:55 up  2:38,  9 users,  load average: 0.10, 0.07, 0.06
Job state_count = Transit:0 Queued:1 Held:0 Waiting:0 Running:49 Exiting:0

[tra50@ac ~]$```

V. Kindratenko, Introduction to GPU Programming (part I), December 2010, The American University in Cairo, Egypt
Installing Tutorial Examples

• Run this sequence to retrieve and install tutorial examples:

```
cd
tar -xvzf cairo2010_tutorial.tgz
cd tutorial
ls
benchmarks src1 src2 src4 src5 src6
```
Accessing the GPU Cluster

You are here

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
Requesting a Cluster Node for Interactive Use

• Run `qstat` to see what other users do

• Run `qsub -I -l walltime=03:00:00` to request a node with a single GPU for 3 hours of interactive use
  – You will see something like this printed out:

```bash
qsub: waiting for job 1183789.acm to start
qsub: job 1183789.acm ready

[tra50@ac10 ~]$ _
```
Requesting a Cluster Node

user 1

user 2

\ldots

user n

ac
(head node)

You are here

ac01
(compute node)

ac02
(compute node)

\ldots

ac32
(compute node)
Some useful utilities installed on AC

• As part of NVIDIA driver
  – nvidia-smi (NVIDIA System Management Interface program)

• As part of NVIDIA CUDA SDK
  – deviceQuery
  – bandwidthTest

• As part of CUDA wrapper
  – wrapper_query
  – showgputime/showallgputime (works from the head node only)
nvidia-smi

Timestamp                          : Mon May 24 14:39:28 2010
Unit 0:
  Product Name            : NVIDIA Tesla S1070-400 Turn-key
  Product ID              : 920-20804-0006
  Serial Number           : 0324708000059
  Firmware Ver            : 3.6
  Intake Temperature      : 22 C
  GPU 0:
    Product Name    : Tesla T10 Processor
    Serial          : 2624258902399
    PCI ID          : 5e710de
    Bridge Port     : 0
    Temperature     : 33 C
  GPU 1:
    Product Name    : Tesla T10 Processor
    Serial          : 2624258902399
    PCI ID          : 5e710de
    Bridge Port     : 2
    Temperature     : 30 C
  Fan Tachs:
    #00: 3566 Status: NORMAL
    #01: 3574 Status: NORMAL
    ...
    #12: 3564 Status: NORMAL
    #13: 3408 Status: NORMAL
PSU:
  Voltage         : 12.01 V
  Current         : 19.14 A
  State           : Normal
LED:
  State           : GREEN

Unit 1:
  Product Name            : NVIDIA Tesla S1070-400 Turn-key
  Product ID              : 920-20804-0006
  Serial Number           : 0324708000059
  Firmware Ver            : 3.6
  Intake Temperature      : 22 C
  GPU 0:
    Product Name    : Tesla T10 Processor
    Serial          : 1930554578325
    PCI ID          : 5e710de
    Bridge Port     : 0
    Temperature     : 33 C
  GPU 1:
    Product Name    : Tesla T10 Processor
    Serial          : 1930554578325
    PCI ID          : 5e710de
    Bridge Port     : 2
    Temperature     : 30 C
  Fan Tachs:
    #00: 3584 Status: NORMAL
    #01: 3570 Status: NORMAL
    ...
    #12: 3572 Status: NORMAL
    #13: 3412 Status: NORMAL
PSU:
  Voltage         : 11.99 V
  Current         : 19.14 A
  State           : Normal
LED:
  State           : GREEN
CUDA Device Query (Runtime API) version (CUDART static linking)
There is 1 device supporting CUDA
Device 0: "Tesla T10 Processor"
- CUDA Driver Version: 3.0
- CUDA Runtime Version: 3.0
- CUDA Capability Major revision number: 1
- CUDA Capability Minor revision number: 3
- Total amount of global memory: 4294770688 bytes
- Number of multiprocessors: 30
- Number of cores: 240
- Total amount of constant memory: 65536 bytes
- Total amount of shared memory per block: 16384 bytes
- Total number of registers available per block: 16384
- Warp size: 32
- Maximum number of threads per block: 512
- Maximum sizes of each dimension of a block: 512 x 512 x 64
- Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
- Maximum memory pitch: 2147483647 bytes
- Texture alignment: 256 bytes
- Clock rate: 1.30 GHz
- Concurrent copy and execution: Yes
- Run time limit on kernels: No
- Integrated: No
- Support host page-locked memory mapping: Yes
- Compute mode: Exclusive (only one host thread at a time can use this device)
cuda_wrapper info:
  version=2
  userID=21783
  pid=-1
  nGPU=1
  physGPU[0]=2
  key_env_var=
  allow_user_passthru=1
  affinity:
    GPU=0,   CPU=0 2
    GPU=1,   CPU=0 2
    GPU=2,   CPU=1 3
    GPU=3,   CPU=1 3
  cudaAPI = Unknown
  walltime = 10.228021 seconds
  gpu_kernel_time = 0.000000 seconds
  gpu_usage = 0.00%

• There are 4 GPUs per cluster node
• When requesting a node, we can specify how many GPUs should be allocated
  – e.g., `\-l nodes=1:ppn=4` in `qsub` resources string will result in all 4 GPUs allocated
• By default, only one GPU per node is allocated

V. Kindratenko, Introduction to GPU Programming (part I), December 2010, The American University in Cairo, Egypt
Compiling and Running an Existing Application

• cd tutorial/src1
  – vecadd.c - reference C implementation
  – vecadd.cu – CUDA implementation

• Compile & run CPU version
  gcc vecadd.c -o vecadd_cpu
  ./vecadd_cpu
    Running CPU vecAdd for 16384 elements
    C[0]=2147483648.00 ...

• Compile & run GPU version
  nvcc vecadd.cu -o vecadd_gpu
  ./vecadd_gpu
    Running GPU vecAdd for 16384 elements
    C[0]=2147483648.00 ...

V. Kindratenko, Introduction to GPU Programming (part I), December 2010, The American University in Cairo, Egypt
nvcc

• Any source file containing CUDA C language extensions must be compiled with nvcc
• nvcc is a compiler driver that invokes many other tools to accomplish the job
• Basic nvcc usage
  – nvcc <filename>.cu [-o <executable>]
    • Builds release mode
  – nvcc -deviceemu <filename>.cu
    • Builds device emulation mode (all code runs on CPU)
  – -g flag allows to build debug mode for gdb debugger
  – nvcc --version

V. Kindratenko, Introduction to GPU Programming (part I), December 2010, The American University in Cairo, Egypt
Anatomy of a GPU Application

• Host side
  – Allocate memory on the GPU (device memory)
  – Copy data to the device memory
  – Launch GPU kernel and wait until it is done
  – Copy results back to the host memory

• Device side
  – Execute the GPU kernel
void vecAdd(int N, float* A, float* B, float* C) {
    for (int i = 0; i < N; i++) C[i] = A[i] + B[i];
}

int main(int argc, char **argv) {
    int N = 16384; // default vector size

    float *A = (float*)malloc(N * sizeof(float));
    float *B = (float*)malloc(N * sizeof(float));
    float *C = (float*)malloc(N * sizeof(float));

    vecAdd(N, A, B, C); // call compute kernel

    free(A); free(B); free(C);
}
Adding GPU support

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
Memory Spaces

- CPU and GPU have separate memory spaces
  - Data between these spaces is moved across the PCIe bus
  - Use functions to allocate/set/copy memory on GPU
- Host (CPU) manages device (GPU) memory
  - cudaMemcpy(void** pointer, size_t nbytes)
  - cudaFree(void* pointer)
  - cudaMemcpy(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction);
    - returns after the copy is complete
    - blocks CPU thread until all bytes have been copied
    - does not start copying until previous CUDA calls complete
    - enum cudaMemcpyKind
      - cudaMemcpyHostToDevice
      - cudaMemcpyDeviceToHost
      - cudaMemcpyDeviceToDevice
Adding GPU support

```c
int main(int argc, char **argv)
{
    int N = 16384;  // default vector size

    float *A = (float*)malloc(N * sizeof(float));
    float *B = (float*)malloc(N * sizeof(float));
    float *C = (float*)malloc(N * sizeof(float));

    float *devPtrA, *devPtrB, *devPtrC;

    cudaMalloc((void**)&devPtrA, N * sizeof(float));
    cudaMalloc((void**)&devPtrB, N * sizeof(float));
    cudaMalloc((void**)&devPtrC, N * sizeof(float));

    cudaMemcpy(devPtrA, A, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrC, C, N * sizeof(float), cudaMemcpyHostToDevice);
}
```

Memory allocation on the GPU card

Copy data from the CPU (host) memory to the GPU (device) memory

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
Adding GPU support

```c
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);

cudaMemcpy(C, devPtrC, N * sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);

free(A);
free(B);
free(C);
```

Kernel invocation

Copy results from device memory to the host memory

Device memory de-allocation

V. Kindratenko, *Introduction to GPU Programming (part I)*, December 2010, The American University in Cairo, Egypt
GPU Kernel

• CPU version
  
  ```c
  void vecAdd(int N, float* A, float* B, float* C)
  {
    for (int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
  }
  ```

• GPU version
  
  ```c
  __global__ void vecAdd(float* A, float* B, float* C)
  {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
  }
  ```