CUDA Computing Workshop

Ken-ichi Nomura, Ph.D.
knomura@usc.edu

Center for High-Performance Computing and Communications

University of Southern California
Outline

• Introduction
• CUDA Programming
• CUDA Examples
  – Pi Calculation with Monte Carlo Method
  – Scalar-vector Multiplication with CUBLAS
• Hands-on
What is GPU?

Graphics Processing Unit (GPU)
General-Purpose computation on Graphics Processing Units (GPGPU)

Tesla K20
Kepler GK110
Streaming Multiprocessor (SMX)

Kepler GK110

192 single- & 64 double-precision units
32 special function, 32 load/store units
What is CUDA?

- **Compute Unified Device Architecture (CUDA)**
- CUDA is a parallel computing platform and programming model that enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

**Basic approaches:**

1. Write own parallel codes using a familiar programming language such as C, C++, Fortran, etc.
2. Use GPU-accelerated library if your code uses CPU-only libraries such as cuBLAS, cuFFT etc.
3. Use GPU-supported software like Matlab, NAMD, gromacs, etc.
CPU vs GPU

GPU is specialized for highly compute-intensive, data parallel computation.

- more transistors are devoted to data processing rather than data caching and flow control

Reduces flow control (e.g. if-statement) and data dependency.

- Expose concurrency!
GPU Node Specifications

- **CPU**: 2 x Intel Xeon @ 2.40GHz, 16 (=2 x 8) cores
- **Memory**: 64 GB
- **GPU**: 2 x Tesla K20 (GK110)
  - Peak performance: 1.17 Tflops (double), 3.52 Tflops (float)
  - Number of CUDA cores: 2496 (13 SMX x 192 core/SMX)
  - CUDA Capability: 3.5
  - Onboard Memory: DDR5 5GB
  - Memory Bandwidth (ECC off): 208GBtyes/sec
- **Network**: InfiniBand FDR
  - 56 Gbps (4 x 14 Gbps)
GPU Software

CUDA driver:  5.0  /usr/usc/cuda/5.0
Compilers:
  GNU 4.4.6,  /usr/bin/gcc
  PGI 13.3,  /usr/usc/pgi/13.3
  Intel 12.1.1,  /usr/usc/intel/12.1.1
Utilities:
  nvprof, cuda-gdb, cuda-memcheck
MPI library:
  OpenMPI 1.6.4  /usr/usc/openmpi/1.6.4
CUDA-enabled Applications:
  NAMD 2.9  /usr/usc/NAMD/2.9-gpu
  Gromacs 4.6.1  /usr/usc/gromacs/4.6.1
  Matlab 2013a  /usr/usc/matlab/2013a
Outline

• Introduction
• CUDA Programming
• CUDA Examples
  – Pi Calculation with Monte Carlo Method
  – Scalar-vector Multiplication with CUBLAS
• Hands-on Session
CUDA Multithread Programming Model

A CUDA program = host + kernel

Host code runs on CPU and kernel code runs on GPU

When invoking a kernel code, specify the number of threads per block and the number of blocks per the grid

**Grid**: a 1 or 2D array of thread blocks

**Block**: a 1, 2 or 3D array of threads

Thread blocks are executed in parallel
Why Grid and Blocks?

Streaming Multiprocessor | CUDA Programming Model
CUDA Memory Model

Each thread can:
• Read/write per-thread registers
• Read/write per-thread local memory
• Read/write per-block shared memory
• Read/write per-grid global memory
• Read only per-grid constant memory

Host code can:
• Read/write per-grid global memory
• Read/write per-grid constant memory
Build-in variables & type

- `3dim` 3-element (x,y,z) integer array
- `dim3 gridDim`
  Dimensions of the grid in blocks
  (gridDim.z unused)
- `dim3 blockDim`
  Dimensions of the block in threads
- `dim3 blockIdx`
  Block index within the grid
- `dim3 threadIdx`
  Thread index within the block

```c
int tid = threadIdx.x + blockDim.x*blockIdx.x;
```
CUDA Kernel

• Definition
  ```c
  __global__ void kernel_fun(argument_list)
  ```

• Invocation
  ```c
  kernel_fun <<<execution configuration>>> (arguments)
  ```
  – Range specifies set of values for thread grid

  ```c
  host_fun() {
    ...
    dim3 dimGrid(4,2,1)
    dim3 dimBlock(2,2,2)
    kernel_fun <<<dimGrid, dimBlock>>> (arguments)
    ...
  }
  ```

  3-element struct accessed by dimGrid.x, dimGrid.y, dimGrid.z
Function Qualifiers

• __host__: functions run on CPU (default)

• __global__: functions run on the GPU but called from host function, providing an interface the host and the GPU.

• __device__: functions run on the GPU, called by codes running on the GPU.
CUDA APIs: Memory Allocation & Free

cudaMalloc(devPtr, size)
Allocates object in the device global memory
Requires two arguments:
devPtr: Address of a pointer to the allocated object
size: Size of the allocated object

cudaFree(devPtr)
Frees object from device global memory
Argument: Pointer to freed object

cudaMalloc((void **) &sumDev, size);
cudaFree(sumDev);
CUDA APIs: Data Transfer

cudaMemcpy(dest, src, size, cmd)

- Arguments
  dest: pointer of an array to receive data
  src: pointer of an array to send data
  size: # of bytes to transfer
  cmd: transfer direction

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice
CUDA APIs: Set Value

cudaMemset(devPtr, value, count)

Fills \textit{count} bytes of memory space pointed by \textit{devPtr} with a constant \textit{value}.

Arguments:
\textbf{devPtr}: Pointer to device memory
\textbf{value}: Integer value to set for each byte of specified memory
\textbf{count}: Size in bytes to set
Outline

• Introduction
• CUDA Programming
• CUDA Examples
  – Pi Calculation with Monte Carlo Method
  – Scalar-vector Multiplication with CUBLAS
• Hands-on Session
PI Calculation with Monte Carlo (MC) Method

Circle: \( S_1 = \pi r^2 \)

Square: \( S_2 = (2r)^2 = 4r^2 \)

\[
\frac{S_1}{S_2} = \frac{\pi r^2}{4r^2} \rightarrow \pi = \frac{4S_1}{S_2}
\]
Monte Carlo Integration

A numerical integration technique using random numbers

Generate many randomly distributed points within the square

The area of the circle can be approximately obtained from the ratio of points inside of the circle and the total number of points.
#include <stdio.h>
#include <stdlib.h>

#define TRIALS_PER_THREAD 4096
#define PI 3.1415926535 // known value of pi

float pi_mc_h(long trials) {
    float x, y;
    long points_in_circle=0;
    for(long i = 0; i < trials; i++) {
        x = rand() / (float) RAND_MAX;
        y = rand() / (float) RAND_MAX;
        points_in_circle += (x*x + y*y <= 1.0f);
    }
    return 4.0f * points_in_circle/trials;
}

int main (int argc, char *argv[]) {
    float pi_cpu = pi_mc_h(TRIALS_PER_THREAD);
    printf("CPU estimate of PI = %f [error of %f]\n", pi_cpu, pi_cpu - PI);
    return 0;
}
CURAND Library

Problem: rand() can not be called from a kernel!
   ➔ Use CUDA random number generator

CURAND can generate random numbers on the host (CPU) memory or the device (GPU) memory.

To generate random numbers on the GPU memory,
1. Include curand_kernel.h
2. Allocate a memory space on device to store CURAND state.
3. Initialize the state with a “seed”
4. Generate random number sequences
PI Calculation (GPU): Header Part

```c
#include <stdio.h>
#include <stdlib.h>
#include <curand_kernel.h>  // CURAND lib header file

#define TRIALS_PER_THREAD 2048
#define BLOCKS 256
#define THREADS 256
#define PI 3.1415926535  // known value of pi
```
int main(int argc, char *argv[]) {
    float host[BLOCKS * THREADS];
    float *dev;
    curandState *devStates;

cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
cudaMalloc((void **) &devStates, BLOCKS*THREADS*sizeof(curandState));
pi_mc<<<BLOCKS, THREADS>>>(dev, devStates);
cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float), cudaMemcpyDeviceToHost);

float pi_gpu=0.0;
for(int i = 0; i < BLOCKS * THREADS; i++) pi_gpu += host[i];
pi_gpu /= (BLOCKS * THREADS);

printf("CUDA estimate of PI = %f [error of %f ]\n",
        pi_gpu, pi_gpu - PI);
cudaFree(dev);
cudaFree(devStates);
return 0;
}
__global__ void pi_mc(float *estimate, curandState *states) {
    unsigned int tid = threadIdx.x + blockDim.x*blockIdx.x;
    int points_in_circle = 0;
    float x, y;

    // Initialize CURAND
    curand_init(tid, 0, 0, &states[tid]);

    for(int i = 0; i < TRIALS_PER_THREAD; i++) {
        x = curand_uniform(&states[tid]);
        y = curand_uniform(&states[tid]);
        // count if x & y is in the circle.
        points_in_circle += (x*x + y*y <= 1.0f);
    }
    estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD;
}
PI Calculation: Compile & Run

```
[hpc-login2 pi_mc]$ source /usr/usc/cuda/5.0/setup.csh
[hpc-login2 pi_mc]$ ls
pi_mc.cu
[hpc-login2 pi_mc]$ nvcc -arch=sm_35 pi_mc.cu -o pi_mc
[hpc-login2 pi_mc]$ qsub -I -d .
-l nodes=1:ppn=16:gpus=2,walltime=30:00

[hpc3136 pi_mc]$ ./pi_mc
# of trials per thread = 2048, # of blocks = 16, # of threads/block = 16.
GPU pi calculated in 0.230000 s.
CUDA estimate of PI = 3.140327 [error of -0.001265 ]
CPU pi calculated in 0.020000 s.
CPU estimate of PI = 3.139565 [error of -0.002028 ]
```
Scalar-Vector Multiplication with CUBLAS

CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms to perform vector-matrix computations) using CUDA.

CUBLAS helper functions to create and terminate CUBLAS objects in GPU space, and transfer data between host and devices.

SAXPY (Single-precision real Alpha X Plus Y) is a Level 1 (vector) operation in the BLAS library.

SAXPY is a combination of scalar multiplication and vector addition, and a common operation in vector processors.

\[ Y_i \leftarrow \alpha X_i + Y_i \]
CUBLAS Basic APIs

cublasInit (void) & cublasShutdown (void)
Call before and after any other CUBLAS API call

cublasAlloc (int \( n \), int elemSize, void **devicePtr)
Allocate a memory space on GPU. \( n \) is the number of elements with elemSize bytes. devicePtr is a pointer to the array.

cublasFree (const void *devicePtr)
Free the GPU memory space referenced by devicePtr.

```c
float *x_dev, *y_dev;
cublasAlloc(N, sizeof(float), (void**) &x_dev);
cublasAlloc(N, sizeof(float), (void**) &y_dev);
```
CUBLAS APIs: Set & Get Vector

cublasSetVector(int \( n \), int \( elemSize \), const void *\( cpumem \), int \(incx\), void *\( gpumem \), int \(incy\))

Copies \( n \) elements from a vector \( cpumem \) on the CPU memory to a vector \( gpumem \) on the GPU memory.

\( cpumem \): pointer of an array to send data
\( gpumem \): pointer of an array to receive data
\( elemSize \): Each element size in byte
\( incx, incy \): Storage spacing size (= 1 for a vector)

cublasGetVector(int \( n \), int \( elemSize \), const void *\( gpumem \), int \(incx\), void *\( cpumem \), int \(incy\))

Copies \( n \) elements from a vector \( gpumem \) on the GPU memory to a vector \( cpumem \) on the CPU memory.
CUBLAS APIs: cublasSaxpy

cublasSaxpy (int n, float alpha, const float *x, int incx, float *y, int incy)

- Multiplies single precision vector x by single precision scalar alpha and adds the result to single precision vector y.
- Overwrites single precision y with single precision alpha * x + y.

http://www.netlib.org/blas/saxpy.f
```c
#include <stdlib.h>
#include <stdio.h>
#include <cublas.h>

int main(void) {
    int N = 4;

    float *x_h, *y_h;
    x_h = (float *)malloc(N * sizeof(float));
    y_h = (float *)malloc(N * sizeof(float));

    printf("initial x_h & y_h: \n");
    for (int i = 0; i < N; i++) {
        x_h[i] = i;
        y_h[i] = 0.1f;
        printf("y_h[%d] = %f, x_h[%d] = %f\n", i, y_h[i], i, x_h[i]);
    }
    cublasInit();
    float *x_dev, *y_dev;
    cublasAlloc(N, sizeof(float), (void**) &x_dev);
    cublasAlloc(N, sizeof(float), (void**) &y_dev);
```
cublasSetVector(N, sizeof(float), x_h, 1, x_dev, 1);
cublasSetVector(N, sizeof(float), y_h, 1, y_dev, 1);

// Perform SAXPY on x & y
  cublasSaxpy(N, 2.0, x_dev, 1, y_dev, 1);
cublasGetVector(N, sizeof(float), y_dev, 1, y_h, 1);

  printf("final x_h & y_h: \n");
  for(int i=0; i<N; i++)
      printf("y_h[%d] = %f, x_h[%d] = %f\n", i, y_h[i], i, x_h[i]);

  free(x_h); free(y_h);
cublasFree(x_dev); cublasFree(y_dev);
cublasShutdown();
return 0;
CUBLAS SAXPY: Compile & Run

[saxpy]$ source /usr/usc/cuda/5.0/setup.csh
[saxpy]$ ls

cublas_saxpy.cu

[saxpy]$ nvcc -lcublas cublas_saxpy.cu -o cublas_saxpy

[saxpy]$ qsub -I -d . -l nodes=1:ppn=16:gpus=2,walltime=30:00
CUBLAS SAXPY: Output

```
[saxpy]$ ./cuda_saxpy

initial x_h & y_h:
y_h[0] = 0.100000, x_h[0] = 0.000000
y_h[1] = 0.100000, x_h[1] = 1.000000
y_h[2] = 0.100000, x_h[2] = 2.000000
y_h[3] = 0.100000, x_h[3] = 3.000000

final x_h & y_h:
y_h[0] = 0.100000, x_h[0] = 0.000000
y_h[1] = 2.100000, x_h[1] = 1.000000
```
Links

CUDA Developer Zone
https://developer.nvidia.com/category/zone/cuda-zone

CUDA Samples
http://docs.nvidia.com/cuda/index.html

CURAND

CUBLAS

Pi calculation with MC
http://coitweb.uncc.edu/~abw/SIGCSE2011Workshop/MonteCarloPi.pdf

CUDA Lecture Note
http://cacs.usc.edu/education/cs653/10CUDA.pdf
Any Questions?
Hands-on Exercise I
CUDA Samples
CUDA Samples

The list of sample codes included in the NVIDIA CUDA Toolkit.

- Go to [http://docs.nvidia.com/cuda/cuda-samples/index.html](http://docs.nvidia.com/cuda/cuda-samples/index.html)
- Download `deviceQuery.tar.gz` and copy it to your home directory.
deviceQuery: Compile

[laptop] ssh knomura@hpc-login2.usc.edu
[hpc-login2]$ source /usr/usc/cuda/5.0/setup.csh
[hpc-login2]$ tar xvfz deviceQuery.tar.gz
[hpc-login2]$ cd 1.Utilities/deviceQuery/
[deviceQuery]$ setenv CUDA_PATH /usr/usc/cuda/5.0/
[deviceQuery]$ make
[deviceQuery]$ ls

deviceQuery* deviceQuery.cpp* deviceQuery.o Makefile
# Submit PBS Script to Use GPUs with CUDA 5.0

[deviceQuery]$

vi run.sh

```csh
#!/bin/csh

#PBS -l walltime=30:00
#PBS -l nodes=1:ppn=16:gpus=2
#PBS -d .

source /usr/usc/cuda/5.0/setup.csh
./deviceQuery
```

[deviceQuery]$

qsub run.sh
deviceQuery: Output

/deviceQuery Starting...

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

Detected 2 CUDA Capable device(s)

Device 0: "Tesla K20m"
  CUDA Driver Version / Runtime Version  5.0 / 5.0
  CUDA Capability Major/Minor version number:  3.5
  Total amount of global memory:  4800 MBytes (5032706048 bytes)
  (13) Multiprocessors x (192) CUDA Cores/MP:  2496 CUDA Cores
  GPU Clock rate:  706 MHz (0.71 GHz)
  Memory Clock rate:  2600 Mhz
  Memory Bus Width:  320-bit
  L2 Cache Size:  1310720 bytes
  ....
Hands-on Exercise II
PI Calculation Benchmark
(CPU vs GPU)
PI Calculation Benchmark

In pi_mc.cu,

```c
#define TRIALS_PER_THREAD 2048
#define BLOCKS 16
#define THREADS 16
```

Change BLOCKS&THREADS = 16, 64, 256, 512 to see how the accuracy and the computation time change.

# of trials per thread = 2048, # of blocks = 16, # of threads/block = 16.
GPU pi calculated in 0.220000 s.
CUDA estimate of PI = 3.136795 [error of -0.004798 ]
CPU pi calculated in 0.020000 s.
CPU estimate of PI = 3.139565 [error of -0.002028 ]
Compile & Submit PBS Script

```
[hpcl-login2:~]$ source /usr/usc/cuda/5.0/setup.csh
[hpcl-login2:~]$ nvcc -arch=sm_35 pi_mc.cu -o pi_mc

#!/bin/csh
#PBS -l walltime=30:00
#PBS -l nodes=1:ppn=16:gpus=2
#PBS -d .
source /usr/usc/cuda/5.0/setup.csh
./pi_mc

[hpcl-login2:~]$ qsub run.sh
```
Programing Tips

• `printf()` can be called from CUDA kernel

```c
#ifdef __global__
void pi_mc(double *estimate, curandState *states) {
    unsigned int tid = threadIdx.x + blockDim.x*blockIdx.x;
    int points_in_circle = 0;
    double x, y;
    printf("tid = %d \n", tid);

    curand_init(tid, 0, 0, &states[tid]); // Initialize CURAND

    ...
}
```

Very useful to debug your code!