Introduction to GPGPU Programming

Pragnesh Patel
pragnesh@utk.edu
Remote Data Analysis and Visualization Center
National Institute for Computational Sciences
University of Tennessee
Outline:

- Motivation
- History
- GPU architecture
- GPU programming model
- CUDA C
- CUDA tools
- OpenCL in brief
- Other useful GPU tools
- Summary
- References
Motivation:
Graphics Processing Unit

- Designed to rapidly manipulate and alter memory in such a way so as to accelerate the building of images in a frame buffer intended for output to a display.
- The term was popularized by NVIDIA in 1999.
- GeForce 256: The World’s first GPU.
GPU:

- GPU = Graphics Processing Unit
  - Chip in computer video cards, PlayStation 3, XBOX etc..
  - Two major vendors: NVIDIA and AMD

NVIDIA GPUs:

- Desktop GPUs
  - GeForce series for CPU
- Mobile GPUs
  - GeForce series for Mobile
- Workstation/HPC GPUs
  - Quadro NVS, Tesla, Kepler
NVIDIA GPUs:

- Supports CUDA and OpenCL
- Fermi (Tesla version)
  - Upto 512 cores
  - Upto 6GB memory
  - Upto 665 GFLOPS – Double precision
  - Caches included: L1 per multiprocessor, L2 shared
- Kepler in 2012
- Maxwell in 2014
AMD GPUs:

- Desktop GPUs
  - Radeon series
- Mobile GPUs
  - Mobility Radeon
- Workstation GPUs
  - FirePro, FireStream
- Supports OpenCL (no CUDA)
Modern GPUs are very efficient at
- Manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for algorithms where processing of large block of data is done in parallel.

GPUs are massively multithreaded manycore chips.
- NVIDIA tesla products have upto 512 cores.
- Over 665 GFLOPS sustained performance (double floating point)
- 6GB of Memory
- Memory bandwidth upto 177 GBytes/sec.

Users across science and engineering disciplines are achieving very good speedups on GPUs.
Typical Supercomputer:

- Large amount of nodes.
  - Distributed memory
  - Multicore processors (e.g. 12 cores per node Kraken)
- Fast interconnect.
- Programming models
  - MPI
  - Hybrid (Pthreads, OpenMP with MPI)
Accelerated Supercomputer:

- Accelerated HPC floating workloads using GPUs.
  - Peak FP performance 10x vs CPU.
  - Memory bandwidth 20x vs CPU.
  - Parallelism, of the order of 500 cores, thousands of threads.

- GPUs are accelerators.
  - Has its own fast memory.
  - Separate card connected to CPU node Via PCI-E bus.
Other accelerators:

- **Intel**
  - Intel MIC (Many Integrated Core)
  - ~ 50 X86 vector cores
  - OpenMP, OpenCL, Intel parallel building blocks etc...
  - First commercial product (Knights corner) in 2012.

- **Other** (FPGA and DSP based system etc...)

Reference: http://intel.com
*Not* for all applications:

- SIMD(Single Program, Multiple Data) are best.
- Operations need to be sufficient size to overcome overhead.
- Think millions of operations.
- Data transfer could be bottleneck (between CPU memory and GPU memory)
How it is different from CPU:

- GPU is specialized for compute-intensive, highly parallel computation—exactly what graphics rendering is about.
- GPU devotes more transistors to data processing rather than data caching and flow control.
Trying it together:

Processing flow on CUDA:

1. Copy processing data
2. Instruct the processing
3. Execute parallel in each core
4. Copy the result
GPU architecture:

- High level block diagram of NVIDIA GPU chip.

Reference: http://nvidia.com
GPU memory model:

- Multilevel levels of memory hierarchy

Reference: http://nvidia.com
Thread: is a ready for execution/running instance of a kernel. Each thread has its own instruction address counter and register state.

Warp: is a group of 32 parallel threads.

Block: is a group of Warps. A block is executed on one multiprocessor. Every block has its own shared memory and registers in the multiprocessor.

Grid: is a group of Blocks.

Host: is the CPU in CUDA applications.

Device: is the GPU in CUDA applications.
GPU memory model:

- GPU has much more aggressive memory subsystem.

Reference: http://nvidia.com
GPU programming:

How to use GPUs

- Use existing GPU software
- Use available libraries for GPUs
- Program GPU with directives
- Program native GPU code
Use existing GPU software:

- NAMD, GROMACS, GPU-HMMER, TeraChem
- Pros
  - No implementation headaches for end users.
- Cons
  - Existing applications do not cover all science areas.
  - Often include limited number of algorithms/models.
  - For many applications the GPU version is still immature.
GPU programming model:

- GPU accelerator is called **device**, CPU is **host**.
- GPU code (kernel) is launched and executed on the device by several threads.
- Threads grouped into thread blocks.
- Program code is written from single thread's point of view.
  - Each thread can diverge and execute a unique code path (can cause performance issues).
Thread Hierarchy:

- **Threads:**
  - 3D IDs, unique in block
- **Blocks:**
  - 3D IDs, unique in grid
- **Dimensions are set at kernel launch.**
- **Built-in variables for device code:**
  - threadIdx, blockIdx
  - blockDim, gridDim

*Since CUDA 4*
Introduction to CUDA:

- **Compute Unified Device Architecture**
- CUDA is a C/C++ language extension for GPU programming.
  - PGI has developed similar FORTRAN 2003 extension.
- Two APIs: Runtime and Driver
CUDA software stack:

- **CUDA Optimized Libraries:** math.h, FFT, BLAS, ...
- **NVIDIA C Compiler**
- **NVIDIA Assembly for Computing (PTX)**
- **CUDA Driver**
- **Profiler**
- **CPU Host Code**
- **Standard C Compiler**
- **CPU**
- **GPU**
- **Integrated CPU + GPU C Source Code**

Reference: http://nvidia.com
Introduction to CUDA C:

- Qualifiers
  - global, device, shared, local, constant...
- Built-in variables
  - threadIdx, blockIdx
- Intrinsics
  - __syncthreads,
- Runtime API
  - Memory, device execution management.
- Kernel launch

```c
__device__ float array[128];
__global__ void kern(float *data)
{
  __shared__ float buffer[32];
  ....
  buffer[threadIdx.x] = data[i];
  ....
  __syncthreads;
  ....
}
float *d_data;
cudaMalloc((void **)&d_data, bytes);
kern<<<1024, 128>>>(d_data);
```
Vector add CPU:

```c
void add(int *a, int *b, int *c)
{
    int i;
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```
CUDA memory management:

//include header files
#include <cuda.h>
#include <cutil.h>
//include kernels
#include "vector_add_kernel.cu"

static const int N = 100000;
int main(int argc, char** argv) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;
    //Memory allocation on device
    cudaMalloc(&dev_a, N*sizeof(int));
    cudaMalloc(&dev_b, N*sizeof(int));
    cudaMalloc(&dev_c, N*sizeof(int));
    //Memory copy host to device
    cudaMemcpy(dev_a, a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, N*sizeof(int), cudaMemcpyHostToDevice);
    //Call Kernel (in next slide)
    //Copy result from GPU to CPU
    cudaMemcpy(c, dev_c, N*sizeof(int), cudaMemcpyDeviceToHost);
    //Free memory
    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
Vector launch kernel:

```c
add<<<1,1>>>(dev_a, dev_b, dev_c);//serial
```

```c
add<<<N,1>>>(dev_a, dev_b, dev_c);//parallel
```

//Only the first parameter interest us right now. The first parameter ask Cuda to execute the function on N parallel blocks.
Vector add kernel function:

```c
///vector_add_kernel.cu serial
static const int N=100000;
__global__ void add(int *a, int *b, int * c)
{
    for(int i=0;i<N;i++){
        c[i] = a[i] + b[i];
    }
}
```
Vector add kernel function:

///vector_add_kernel.cu parallel
__global__ void add(int *a, int *b, int* c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]
}
Launching Kernels:

- Modified C function call syntax:
  
  ```
  kernel<<<dim3 dG, dim3 dB>>>(...)
  ```

- Execution Configuration ("<<< >>>")

- **dG** - dimension and size of grid in blocks Two-dimensional: x and y
  Blocks launched in the grid: dG.x * dG.y

- **dB** - dimension and size of blocks in threads: Three-dimensional: x, y, and z
  Threads per block: dB.x * dB.y * dB.z

- Unspecified dim3 fields initialize to 1
Compiling CUDA code:

- Compilation tools are a part of CUDA SDK.
- `nvcc` compiler translates code written in CUDA into PTX.
- `nvcc` separates the code for host and device.
  - Host code is compiled with regular C/C++ compiler.
- More information: 
  [http://www.nics.tennessee.edu/~ksharkey/tutorials/](http://www.nics.tennessee.edu/~ksharkey/tutorials/)
Compiling CUDA code:

On Keeneland:

```bash
>module load PE-intel
>module load cuda/4.1

> nvcc -ccbin $CC -o gpu.out gpucode.cu
```

On Nautilus:

```bash
>module load PE-gnu
>module load cuda/4.0RC2

> nvcc -ccbin $CC -o gpu.out gpucode.cu
```

- More information:
  [http://www.nics.tennessee.edu/~ksharkey/tutorials/](http://www.nics.tennessee.edu/~ksharkey/tutorials/)
Running CUDA code:

On Keeneland:

> qsub -I -l nodes=1:ppn=1:gpus=3,walltime=00:30:00
> ./gpu.out

On Nautilus:

> qsub -I -l ncpus=1,gpus=1,walltime=00:30:00
> ./gpu.out

- More information:
  
  https://wiki-rdav.nics.tennessee.edu/index.php/Using_the_Nvidia_GPUson_Nautilus

  AND

  http://keeneland.gatech.edu/support/quick-start#runningjobs
Hybrid:

- CUDA with MultiGPU
- CUDA + OpenMP
- CUDA + MPI
- CUDA + OpenMP + MPI
CUDA libraries:

- MAGMA
- CUBLAS
- CULA
- CUFFT
- CUSPARSE
- THRUST
- Optix

- Easy to use in your programs.
- Algorithms in libraries are usually efficient.
CUDA debuggers and profilers:

- Debuggers:
  - Allinea DDT
  - CUDA-GDB
  - Totalview
  - Cuda-memcheck

- Profilers:
  - Tau
  - NVIDIA visual profiler
CUDA and OpenCL:

- NVIDIA: CUDA
  - Use compiler to build kernels
  - C language extensions (nvcc)
    - Also a low-level driver-only API

- Open-free standard.
- Builds kernel at runtime.
- API only, no new compiler-API calls to execute kernel
Directive based GPU code:

- Two main products
  - PGI accelerators
  - HMPP (CAPS enterprise)
- Normal C or Fortran code with directives to guide compiler in creating a GPU version.
- Backend supporting CUDA, OpenCL and even normal CPUs.
Pros
- Same code base as CPU version
- Less time consuming
- Portability is better due to different backends.

Cons
- Generated code may not be as fast as hand-tuned CUDA.
OpenACC:

- Describes a collection of compiler directives to specify loops and regions of code in standard C, C++, and Fortran.
- Allow programmer to create high-level host+ accelerators programs without the need to explicitly initialize the device, manage data or program transfers.
- Backed by PGI, CAPS, Cray and NVIDIA
- Part of OpenMP 4.0?
Ocelot:

- Aim to compile CUDA programs so that they can be run on architectures other than NVIDIA GPUs.
- It is a modular dynamic compilation framework for heterogeneous system, providing various backend targets for CUDA programs and analysis modules for the PTX virtual instruction set.
- Proliferation of Heterogeneous computing.
- Ocelot currently allows CUDA programs to be executed on NVIDIA GPUs, AMD GPUs, and x86-CPUs at full speed without recompilation.
Jacket:

- It combines the speed of CUDA and the graphics of the GPU with the user friendliness.
- Provides GPU library for C, C++, Fortran, Python and MATLAB.
- Provides GPU counterparts to CPU data types, such as real and complex double, single, uint32, int32, logical, etc. Any variable residing in the host (CPU) memory can be cast to Jacket's GPU data types.
- It’s memory management system allocates and manages memory for these variables on the GPU automatically, behind-the-scenes. Any functions called on GPU data will execute on the GPU automatically without any extra programming.
- For more information: http://www.accelereyes.com/
Jacket example:

**CPU:**

\[
\begin{align*}
X &= \text{double}(\text{magic}(3)); \\
Y &= \text{ones}(3, \text{'double'}); \\
A &= X \times Y
\end{align*}
\]

**GPU:**

```
addpath <jacket_root>/engine
X = gdouble(magic(3));
Y = gones(3, \text{'double'});
A = X \times Y
```
Programming languages and GPGPU:

- pyCUDA, pyOpenCL
- MATLAB with CUDA toolbox
- CUDA FORTRAN
- ROpenCL, RCUDA
- Haskell, Perl etc…
Resources:

- http://gpgpu.org/
- http://www.gputechconf.com/page/home.html#
- http://keeneland.gatech.edu/software/cuda
Rules for fast GPU codes:

- Get the data on the GPU (and keep it there! If possible)
  - PCIe x16 v2.0 bus: 8GiB/s in a single direction
  - GPUs: ~180 GiB/s
- Give the GPU enough work to do
- Reuse and locate data to avoid global memory bottlenecks
- Corollary: Avoid `malloc/free`
Summary:

- Accelerated supercomputers emerging.
- GPUs offer tremendous potential to accelerate scientific applications.
- Newer generations GPUs getting easier to program.
- Challenges still remain in using them efficiently.
- Still a few cliffs:
  - HOST-GPU transfer
  - Careful memory access
  - Lots of parallelism
  - Thread divergence
Accelerated Supercomputer:

- Challenges remain
  - Applicability: Can you solve your algorithm efficiently using a GPU?
  - Programmability: Effort of code writing that uses a GPU efficiently.
  - Portability: Incompatibilities between vendors
  - Availability: Are you able gain access to large scale system?
  - Scalability: Can you scale the GPU software efficiently to several nodes?
References:

- https://nimrodteam.org/meetings/team_mtg_8_10/nimrod_gpu.pdf
- http://people.maths.ox.ac.uk/~gilesm/hpc/NVIDIA/NVIDIA_CUDA_Tutorial_No_NDA_Apr08.pdf
- http://www.nics.tennessee.edu/~ksharkey/tutorials/
- http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/08-opencl.pdf
- Introduction GPU computing by Sebastian von alfthan
- Supercomputing for the Masses: Killer-Apps, Parallel Mappings, Scalability and Application Lifespan by Rob Farber
- The PTX GPU Assembly Simulator and Interpreter By N.M. Stiffler, Zheming Jin, Ibrahim Savran
In this tutorial session, we covered

- GPU architecture
- GPU programming model
- CUDA C
- CUDA tools
- OpenCL in brief
- Other useful GPU tools
- References
Thank You !!!