Introduction to CUDA

CUDA Basics

S. Sundar and M. Panchatcharam

August 9, 2014

Outline

CUDA Architecture

CUDA

Compute Unified Device Architecture

- Before, CUDA architecture GPU used for gaming purposes
- GPU partitioned computing resources into vertex and pixel shaders
- After, CUDA, GPU excels computation in addition to performing well at traditional graphics tasks

CUDA Architecture

- NVIDIA introduced CUDA along with G80
- CUDA 1.0 was the first compute capability with G80
- Latest Release CUDA 6.0
- Developer Version CUDA 6.5
- CUDA has many compute capability, For example, 1.0, 1.1, 1.2,1.3, 2.0,2.1, 3.0, 3.5 and 5.0
- Don't confuse with version and compute capability
- compute capability 1.0 is now obsolete

CUDA Architecture

- Compute capability 1.x - Tesla architecture or G80 and GT200
- Compute capability 2.x - Fermi architecture
- Compute capability 3.x - Kepler architecture
- Compute capability 5.x - Maxwell architecture

Compute Capability 1.x

Let us look how does it work

- Using `cudaGetDeviceProperties()`, you can get your system's compute capability
- Cycle 0: Allocates separate memory for CPU and GPU, CPU fills the first buffer
- Cycle 1: CPU invokes CUDA kernel (a GPU task) on the GPU. CPU fetches next data where as GPU processes the received data. CPU is ready to fill the next buffer
- Cycle 2: CPU fills the buffer and invokes kernel. CPU checks whether kernel from cycle 1 which was processing buffer 0 has completed
- Cycle N: Repeat Cycle 2, alternating between which buffer reads and writes on the CPU with the buffer being processed on the GPU

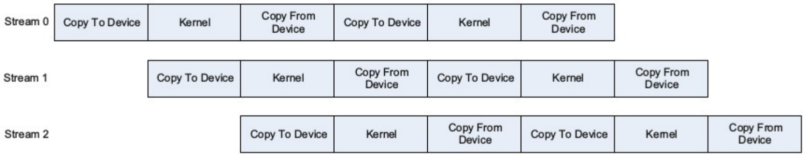
Compute Capability 1.3

- GT200 belongs to CC 1.3
- Supports Double precision
- Fast single precision
- Single precision works faster, where as double precision slows down

Compute Capability 2.0

- Fermi hardware belongs to CC 2.0
- 16 K to 48K of L1 cache memory on each SP
- Shared L2 cache for all SMs
- ECC (in Tesla)-Error correcting code
- Tess supports dual copy engines
- Shared memory 48 K per SM

Dual copy engines



Compute Capability 2.1

- 48 CUDA cores per SM instead of 32 per SM in CC 2.0
- Eight Single-precision, SFU instead of four in CC 2.0
- Dual warp dispatcher
- Superscalar approach
- Hardware uses instruction level parallelism (ILP) within each thread
- ILP differs from TLP (Thread Level Parallelism)

Compute Capability 3.0

- Kepler architecture belongs to this group
- 192 CUDA cores per SMX
- 32 single-precision, SFU
- Quad warp scheduler
- Number of instruction issued at once by scheduler = 2

Compute Capability 5.0

- Maxwell architecture
- 1.5 to 2 times faster than Kepler architecture
- 8 to 10 times performance increase
- 2 MB L2 cache
- Memory bus 128 bit instead of 192 bit as in Kepler
- Instead of SM, now SMM
- SMM units is partitioned so that each of the four warp scheduler controls isolated floating point 32 CUDA cores
- Kepler share resources, where as Maxwell does not during load/store, SFU
- SMM allows fine-grain allocation of resources than SMX
- 128 CUDA core SMM = 90% of 192 CUDA core SMX

Summary Table

Specifications	1.x	2.0	2.1	3.0	3.5	5.0
No. of Cores	8	32	48	192	192	128
No. of SFU	2	4	8	32	32	32
No. of Warp Schedulers	1	2	2	4	4	4
No. of Instr. per scheduler	1	1	2	2	2	2

Grids, Blocks, Threads

- NVIDIA use a variant of SIMD, called SPMD (Single Program, Multiple Data)
- The heart of the parallel programming in GPU is the idea of thread
- Single flow of execution through the program
- Think of the cotton thread and warp in a garment
- In the same way threads of cotton are woven into cloth, threads used together make up a parallel program
- Threads grouped into warps, blocks and grids

Threads

Threads

A thread is the fundamental building block of a parallel program

- Are you familiar with multicore programming?
- No? No problem. You are using a single thread in any serial code
- Thinking in terms of lots of threads is hard
- Like it or not, to improve program speed requires us to think in terms of parallel design

Threads

- GPU supports huge numbers of threads, fine-grained parallelism
- CPU also supports threads but based on coarse-grained parallelism
- GPUs are designed for running a large number of tasks
- CPUs and GPUs have stall conditions
- GPU handle with high frequency

Threads

Look at the simple piece of code

```
void Function()
{
  for(int i=0;i<128;i++)
  {
      a[i]=b[i]*c[i];
  }
}
```

Translate this to 128 threads in CUDA, where each thread executes `a[i]=b[i]*c[i];`

Threads

- None of `b[i]` and `c[i]` depends on other
- Independent loop, easy to parallelize
- In a quad core CPU, each core handles 32 indices
- core 1 handles 0-31 indices, core 2 : 32-63 indices, Core 3: 64-95 and Core 4:96-127
- CUDA translate this loop by creating kernel execution

Threads

CUDA kernel function conceptually looks identical to the loop body without structure

```
__global__ void kernel(int * a, int *b, int *c)
{
    a[i]=b[i]*c[i];
}
```

- Lost the loop and loop control variable i
- __ global__ added to C, that tells the CPU to generate GPU code
- CPU and GPU separates memory spaces
- CPU cannot GPU parameters and vice versa

Threads

- Note, i is no longer defined
- CUDA provides a special parameter, different for each thread: thread ID

```
__global__ void kernel(int * a, int *b, int *c)
{
    int i=threadIdx.x;
    a[i]=b[i]*c[i];
}
```

Threads

- Thread 0 returns 0, Thread 1 returns 1, Thread 127 returns 127
- Each thread does exactly two reads from memory, one multiply and one store
- Code execution is identical, but data changes

Warps

Warps

Threads are grouped into 32 thread groups called warps. Half warp is a group of 16 threads

- 128 threads translated to 4 warps
- At first, extract the thread ID and then calculate the address in the arrays and issue a memory fetch request
- Next, multiply which requires both operands so the thread is suspended
- When all 32 threads in that block of 32 threads are suspended, the hardware switches to another warp

Warps

Kernel

Kernel

CUDA defines an extension to the C language used to invoke a kernel.
Kernel is just a name for a function that executes on the GPU
`kernel<<<numBlocks,numThreads>>>(param1,param2,...)`

- Let us discuss about blocks later
- numThreads is the number of number of threads required to launch into the kernel
- 512 threads per block in GT200 and 1024 threads per block in Fermi/Kepler
- Parameters can be passed via registers or constant memory
- For 128 threads with three parameters, we use 3 x 128 = 384 registers

Kernel

- Each SM has 8192 registers
- If we run one block of thread per SM, we have 64 registers (8192/128)
- Running one block of 128 threads per SM is a very bad idea
- Let us use Blocks

Blocks

Blocks

Group of threads are called blocks

- 512 threads per block
- Number of threads varies depending on architecture
- The first parameter in kernel function is the number of blocks
- `kernel<<<2,128>>>(param1,param2,...)` has 2 blocks and 2 x 128 threads
- Kernel function is executed 2×128 times each with different thread

How to calculate threadID?

- i=blockIdx.x*blockDim.x+threadIdx.x

```
__global__ void kernel(int * a, int *b, int *c)
{
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    a[i]=b[i]*c[i];
}
```

Blocks

- First block has `blockIdx.x=0`, so `i=threadIdx.x`
- For `blockIdx.x=1`, `blockDim.x=128`, so `i>128`
- Notice: Small error happened due to adding one more block
- We have 256 threads, but we did not change size of the array
- Accessing beyond the array gives error
- Change the kernel as
- `kernel<<<2,64>>>(param1,param2,...)`

Blocks

- Total number of blocks in GT200 is 65536 blocks
- Each block has 512 threads and in total 33554432 (around 3.5 million) threads
- Fermi architecture has 1024 threads per block, in total 64 million threads
- With 64 million threads, you can process upto 64 million elements

Threads

- Thread 0 returns 0, Thread 1 returns 1, Thread 127 returns 127
- Each thread does exactly two reads from memory, one multiply and one store
- Code execution is identical, but data changes

Grids

Grid

A grid is simply a set of blocks where you have an X and Y axis, a 2D mapping

- Thread index is calculated using $Y \times X \times T$
- The number of threads in a block should always be a multiple of the warp size

2D Threads

- 2D array in terms of blocks means you get two thread indexes
- `i=blockIdx.x*blockDim.x+threadIdx.x;`
- `j=blockIdx.y*blockDim.y+threadIdx.y;`
- `a[i][j]=1.0;`
- CUDA runtime specifies the X and Y axis using `blockDim.x` and `blockDim.y`
- `dim3 numThreads(16,8);`
- `dim3 numBlocks(2,2);`
- `kernel<<<numBlocks,numThreads>>>(param1,param2,...)`

Register

- GPU has thousands of registers per SM
- Major difference between CPU and GPU are how they map registers
- To run a new task the CPU needs to do a context switch
- Context switch takes several hundred CPU cycles
- GPU uses threads to hide memory fetch and instruction execution latency
- GPU dedicates real registers to each thread and every thread

Shared Memory

- Shared memory is effectively a user-controlled L1 cache
- L1 cache and shared memory share a 64 K memory segment per SM
- Shared Memory speed is driven by the core and clock rate
- Each thread shares the data using shared memory (See Figure)

Constant Memory

- Constant memory is a form of virtual addressing of global memory
- It is a read only memory
- Size of Constant memory is 64 K

Global Memory

- Global memory is writable from both GPU and the CPU
- GPU cards transfer data to and fro without the help of CPU
- Memory from GPU is accessible to the CPU host processor in one of the three ways
  - Explicitly with a blocking transfer
  - Explicity with a nonblocking transfer
  - Implicitly using zero memory copy

Texture Memory

- Texture memory can be used in two ways
- Caching on compute 1.x and 3.x hardware
- Hardware-based manipulation of memory reads
- Texture memory is optimized for locality
- It expects data to be provided to adjacent threads

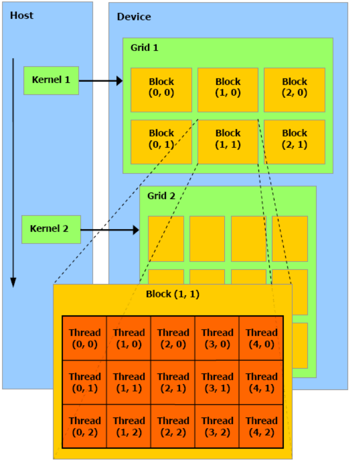
ECC

ECC

Error correction code. ECC memory provides for automatic error detection and correction

- ECC is must for data centers
- Electrical devices emits small amount of radiation
- Radiation can change the contents of memory cells
- This may lead to unaccepted level error if not properly packed, especially computing center
- ECC detects and corrects single-bit upset conditions that you can find in large data centers
- ECC is available in Tesla products

CUDA Threads



Installing the SDK

Required

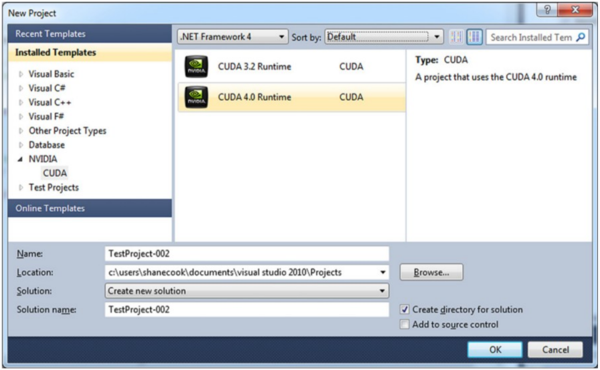
- Microsoft visual studio 2005, 2008 or 2010
- Download and install latest NVIDIA drivers

Install in this order

- NVIDIA development drivers
- CUDA toolkit
- CUDA SDK
- GPU computing SDK
- Parallel Nsight debugger

Creating Project

- Go to File $\longrightarrow$ New $\longleftrightarrow$ Wizard
- The wizard will create a single project containing the kernel.cu



Linux

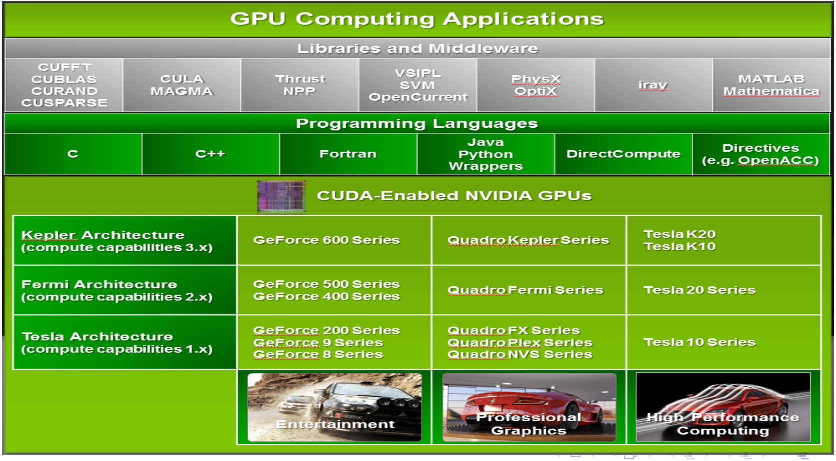
To install in Ubuntu, follow these instructions Open terminal

- `sudo init 3`
- `gedit /etc/sudoers`
- `username ALL=(ALL) ALL`
- `sudo chmod +w /etc/default/grub`
- `sudo gedit /etc/default/grub`

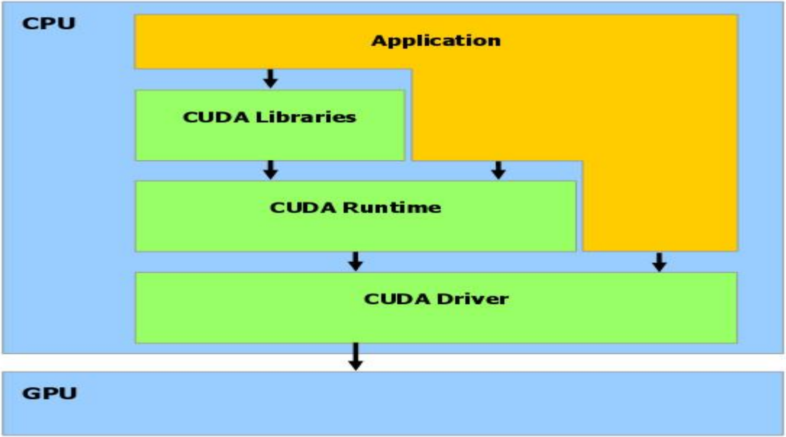
Ubuntu

- # GRUB_CMDLINE_LINUX_DEFAULT1/4"quiet splash"
- GRUB_CMDLINE_LINUX_DEFAULT1/4"text"
- sudo update-grub
- Download the driver and cd Downloads
- sudo sh NVIDIA-version.run
- sudo sh <sdk-version.run
- export PATH=/usr/local/cuda/bin:$ PATH
- export LD_LIBRARY_PATH=/usr/local/cuda/lib:$ LD_LIBRARY_PATH

CUDA Applications



CUDA Stack


CPU
Application
CUDA Libraries
CUDA Runtime
CUDA Driver
GPU

CUDA Languages

CUDA can be programmed using

- C, C++
- Fortran
- Java, Python
- Matlab
- Mathematica

Almost all computing software packages supports GPU now

CUDA C

- Even thought CUDA is supported in many commercial software, the first introduction was CUDA C
- All other software are 90% variant of CUDA C
- Because CUDA is an extension of C language
- Next Lecture : CUDA C
- Ready to practice CUDA C?

Are you Ready to Program?

THANK YOU