CUDA BASIC
Contents

- Why GPU Computing?
- CUDA Software Overview
- CUDA Programming Model
  - GPU Overview
  - How to Compute on GPU
  - Kernel Execution
  - Memory Hierarchy
  - Compilation
  - Asynchronous Concurrent Execution
Why GPU Computing

31-10-2016
Why GPU Computing

▷ GPU are made for gaming
  – highly parallel architecture: display thousands of pixels simultaneously
CPU vs GPU: Transistors Repartition

GPU comes from graphics rendering:
- Single Instruction Multiple Data (compute-intensive & few control)
- Throughput oriented (thousands of pixels simultaneously)
CPU vs GPU: Transistors Repartition

GPU comes from graphics rendering:
- Single Instruction Multiple Data (compute-intensive & few control)
- Throughput oriented (thousands of pixels simultaneously)
- GPU equivalent to a CPU core: fetch, decode, load operand, execute?

~ 2-3 GHz

CPU Core

< 1 GHz

CUDA Core

Streaming Multiprocessor (SM)
CPU vs GPU : Transistors Repartition

GPU comes from graphics rendering :
- Single Instruction Multiple Data (compute-intensive & few control)
- Throughput oriented (thousands of pixels simultaneously)
- 1 SM comparable to 1 CPU core : fetch, decode, load operand, execute
- Connected through PCI-Express
OPPORTUNITY

▶ Since years ~2000, frequency scaling is over... We are now scaling cores!

▶ GPU is a massively multi-threaded many-cores architecture
  – Thousands of threads executed in parallel
    • Kepler (K80) able to run 2048*13=26624 threads in parallel
    • Pascal 1024*56=57334 threads

▶ GPU is a relatively cheap commodity component
  – Big Market, low production prices... comparable to CPU

▶ GPU is fast
CPU vs GPU: Theoretical Computational Peak
CPU vs GPU: Theoretical Bandwidth

Theoretical Peak GB/s

- GeForce GPU
- Tesla GPU
- Intel CPU

2003 2005 2007 2009 2011 2013 2015
Parallelism and Performance

GPU Speed-up
- x100-1000 performance gain?

Amdahl’s law:
- P: proportion made parallel
- 1-P: proportion kept serial
- N: amount of processors

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

« Debunking the 100X GPU vs. CPU myth »
« Closing the Ninja Performance Gap through Traditional Programming and Compiler Technology »
How to GPU Computing

- **CUDA**
  - runtime API (high level)
  - driver API (low level)

- **OpenCl**
  - same level as CUDA driver API

- **OpenAcc**
  - directives based (OpenMP like)
CUDA Software Overview
Software Overview


• Package contains:
  – CUDA Driver
  – CUDA SDK (Software Development Kit)
  – CUDA Toolkit:
    • GPU accelerated libraries : cuFFT, cuBLAS, cuSPARSE, cuRAND, NPP, Thrust, CUDA Math Library
    • Debugging
      • cuda-gdb
      • cuda-memcheck for memory errors, race conditions, bad memory accesses, etc
      • nsight Visual Studio Edition or Eclipse Edition
    • Profiling
      • nvprof command-line profiling
      • nvvp visual profiler
Software Overview

▶ ONLINE CUDA Documentation:

- GPU Management & Deployment Documentation:  
  http://docs.nvidia.com/deploy/index.html

- CUDA Toolkit Documentation  
  http://docs.nvidia.com/cuda/index.html

▶ OFFLINE CUDA Documentation:

- All documentations can be found in directory:  
  /opt/cuda/(CUDA_VERSION)/doc/pdf/

▶ Start with “CUDA C Programming Guide”
Extended C

► Declspecs
  - global, device, shared, local, constant

`__device__` float filter[N];
`__global__` void convolve (float *image) {
  `__shared__` float region[M];
  ...
}

► Keywords
  - threadIdx, blockIdx

region[threadIdx] = image[i];
`__syncthreads()`
...
image[j] = result;
}

► Intrinsics
  - `__syncthreads`

► Runtime API
  - Memory, symbol, execution management

  // Allocate GPU memory
  void *myimage = cudaMalloc(bytes)

► Function launch
  // 100 blocks, 10 threads per block
  convolve<<<100, 10>>>(myimage);
CUDA Programming Model

31-10-2016
GPU Overview

31-10-2016
A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU (or host)
  - Has its own DRAM (device memory)
  - Runs many threads in parallel

- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
    - more threads => better to hide latency (memory accesses, operations)
Hardware Overview

**Device** contains
- Multiprocessors
- Memory
- Host access interface

**Multiprocessors** contains
- ALUs
- Registers
- Shared Memory
- Access to Local Memory
- Access to Global Memory
Memory Overview

- **Global memory**
  - Main memory to communicate (R/W data) between host and device
  - Contents visible to all threads
  - No cache (HW < 2.0)
  - Cache L1/L2 (HW >= 2.0)

- **Texture and constant memories**
  - Constants initialized by host
  - Contents visible to all threads
  - Cache available
deviceQuery

Device 0: "Tesla K30"
CUDA Driver Version / Runtime Version: 6.5 / 6.5
CUDA Capability Major/Minor version number: 3.7
Total amount of global memory: 11520 MB (12979136668 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores
GPU Clock Rate: 824 MHz (0.82 GHz)
Memory Clock rate: 2595 MHz
Memory Bus Width: 384-bit
L2 Cache Size: 1572864 bytes
Maximum Texture Dimension Size (x,y,z): 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size (num layers): 1D=(16384), 2648 layers
Maximum Layered 2D Texture Size (num layers): 2D=(16384, 16384), 2648 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 65536
Warp size: 32
Maximum number of threads per multiprocessor: 2048
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 2 copy engine(s)
Run time limit on kernels: No
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Enabled
Device supports Unified Addressing (UVA): Yes
Device PCI Bus ID / PCI location ID: 4 / 0
Compute Mode: < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
How to Compute on GPU
How to Compute on GPU

5 steps to offload computation on the GPU:

1. Memory allocation
2. Data transfer
3. Execution (computation)
4. Data transfer
5. Free memory

HOST CODE
CPU calls to GPU driver
Memory Allocation

**cudaMalloc()**
- Allocates object in the device Global Memory
- Requires two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object

**cudaFree()**
- Frees object from device Global Memory
  - Pointer to freed object
Memory Allocation Example

- Allocate a 1024 * 1024 single precision float matrix:

```c
float* MyMatrixOnDevice;
int size = 1024 * 1024 * sizeof(float);

cudaMalloc( (void**) &MyMatrixOnDevice, size);
```

- `cudaFree(MyMatrixOnDevice);`
Data Transfer

**cudaMemcpy()**
- memory data transfer
- Requires 4 parameters
  1. Pointer to destination
  2. Pointer to source
  3. Number of bytes copied
  4. Type of transfer
    - Host to Host
    - Host to Device
    - Device to Host
    - Device to Device

- Asynchronous variant supported
Data Transfer Example

- Transfer of a 1024 * 1024 single precision float array:
  - host_ptr: pointer on host memory
  - device_ptr: pointer on GPU memory

- Send data from CPU to the GPU:

  \[
  \text{cudaMemcpy}(\text{device\_ptr}, \text{host\_ptr}, \text{size}, \text{cudaMemcpy}\text{HostTo}\text{Device});
  \]

- Send data from GPU to the CPU:

  \[
  \text{cudaMemcpy}(\text{host\_ptr}, \text{device\_ptr}, \text{size}, \text{cudaMemcpy}\text{DeviceTo}\text{Host});
  \]
Kernel Execution
How to Execute Code on GPU

▶ Three steps:
  1. define a kernel which will be executed by threads
  2. define a grid of threads
  3. call the kernel with the grid of threads

▶ OpenMP Analogy:

```c
#pragma omp parallel for
for(int i=0; i<size; i++){
    A[i] = i;
}
```

```
//definition of the grid of threads
dim3 dimBlock()
dim3 dimGrid()

//kernel call
mykernel<<<dimGrid, dimBlock>>>(A)
```

```
//definition of the kernel
__global__ mykernel(float *A){
    int i = ...
    A[i] = i;
}
```
CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on</th>
<th>Callable from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> void myKernel()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>device</strong> float myDeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>host</strong> float myHostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- Kernel functions must return void

- For functions executed on the device:
  - Recursion possible
  - No static variable declarations inside the function
  - No variable number of arguments
Example

```c
__device__ int myDeviceFunc(int a, int b){
    return a+b;
}

__global__ void myKernel(int size_x, int *in, int * out){
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    out[index] = myDeviceFunc(in[index], 3);
}
```
Thread Batching: Grids and Blocks

- A kernel will be executed by multiple threads organized in a grid of thread blocks
  - All threads share data memory space

- The \texttt{dim3} type allows to specify:
  - the number of threads in a block
  - the number of blocks in a grid

- Exemple:
  - kernel 1:
    - \texttt{dim3 dimblock(5,3,1)}
    - \texttt{dim3 dimgrid(3,2,1)}
  - kernel 2:
    - \texttt{dim3 dimblock(?,?,?)}
    - \texttt{dim3 dimgrid(4,3,1)}
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can be identified in a kernel

- Block keywords
  - `threadIdx.\{x,y,z\}` defines the thread index inside the block
  - `blockDim.\{x,y,z\}` defines the block dimensions

- Grid keywords
  - `blockIdx.\{x,y,z\}` defines the block index inside the grid
  - `gridDim.\{x,y,z\}` defines the grid dimension

- Simplifies memory addressing when processing multidimensional data
Example 1D

```c
__global__ void myKernel(int size_x, int * out) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < size_x) {
        out[index] = index;
    }
}
```

dim3 dimBlock; dim3 dimGrid;
dimBlock.x = 4;
dimGrid.x = (size_x + dimBlock.x - 1) / dimBlock.x;
myKernel<<<dimGrid, dimBlock>>>(size_x, d_out);

```
<table>
<thead>
<tr>
<th>block 0</th>
<th>block 1</th>
<th>block 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
<td>0 1 2 3</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
</table>

out
Example 2D

```c
__global__ void myKernel(int size_x, int size_y, int * out){

int idx = threadIdx.x + blockIdx.x * blockDim.x;
int idy = threadIdx.y + blockIdx.y * blockDim.y;

if( (idx < size_x) && (idy < size_y) ){
    out[idx + idy *size_x] = idx + idy *size_x;
}
}

dim3 dimBlock; dim3 dimGrid;
dimBlock.x = 32;
dimBlock.y =   4;

dimGrid.x = (size_x + dimBlock.x - 1) / dimBlock.x;
dimGrid.y = (size_y + dimBlock.y - 1) / dimBlock.y;

myKernel<<<dimGrid, dimBlock>>>(size_x, size_y, d_out);
```
Memory Hierarchy
Memory Hierarchy

Streaming Multiprocessor N

... 

Streaming Multiprocessor 1

CUDA core
CUDA core
CUDA core
CUDA core
... 

Registers

Shared Memory

Device Memory
Memory Hierarchy

- A thread runs on 1 CUDA core
- A thread has access to:
  - registers
  - shared memory
  - device memory
Memory Hierarchy

- A thread block runs on 1 streaming multiprocessor
- Threads in a same block communicate through:
  - shared memory
  - device memory
Memory Hierarchy

- A multiprocessor can run multiple thread block
- Threads in different blocks communicate through:
  - device memory

device memory data lifetime = GPU thread (context) lifetime
Memory Hierarchy

- Read-Only Cached Memory
  - Constant Memory
  - Texture Memory

- Logical Memory Space
  - Local Memory: Registers spilled to Global Memory
Memory Hierarchy

(Device) Grid

Block (0, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Local Memory
- Global Memory

Block (1, 0)
- Shared Memory
- Registers
- Thread (0, 0)
- Local Memory
- Global Memory
Compilation

31-10-2016
Compilation

Any source file containing CUDA language extensions must be compiled with `nvcc`
- *.cu files

`nvcc` is a compiler driver
- Works by invoking all the necessary tools and compilers like nvvm, g++, ...

`nvcc` can output:
- Either C code
  - That must then be compiled with the rest of the application using another tool
- Or object code directly
Compilation

C/C++ CUDA Application

NVCC

PTX Code

CPU Code

Virtual

PTX to Target Compiler

Physical

K80

... GPU
Compiling for the Right Architecture

Specify the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

```
Stage 1 (nvvm)
```

```
x.ptx
```

```
Stage 2 (ptxas)
```

```
x.cubin
```

```
nvcc -arch=
```

virtuall compute architecture
Compiling for the Right Architecture

Specify the name of the NVIDIA GPU to assemble and optimize PTX for.

nvcc

-x=cu (device code part)

Stage 1 (nvvm)

Stage 2 (ptxas)

x.ptx

x.cubin

virtual compute architecture

-real compute architecture

-code=

-code=
## Virtual Architecture Feature List

<table>
<thead>
<tr>
<th>Feature</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>compute_20</code></td>
<td>Basic features</td>
</tr>
<tr>
<td>(deprecated)</td>
<td>+ Fermi support</td>
</tr>
<tr>
<td><code>compute_30</code></td>
<td>+ Kepler support</td>
</tr>
<tr>
<td>and <code>compute_32</code></td>
<td>+ Unified memory programming</td>
</tr>
<tr>
<td><code>compute_35</code></td>
<td>+ Dynamic parallelism support</td>
</tr>
<tr>
<td><code>compute_50</code></td>
<td>+ Maxwell support</td>
</tr>
<tr>
<td>, <code>compute_52</code></td>
<td></td>
</tr>
<tr>
<td>and <code>compute_53</code></td>
<td></td>
</tr>
<tr>
<td>sm_20 (deprecated)</td>
<td>Basic features  + Fermi support</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>sm_30 and sm_32</td>
<td>+ Kepler support  + Unified memory programming</td>
</tr>
<tr>
<td>sm_35</td>
<td>+ Dynamic parallelism support</td>
</tr>
<tr>
<td>sm_50, sm_52, and sm_53</td>
<td>+ Maxwell support</td>
</tr>
</tbody>
</table>
Compiling for the Right Architecture

nvcc

-arch=compute_XX

virtual compute architecture

x.cu (device code part)

Stage 1 (nvvm)

x.ptx

Stage 2 (ptxas)

x.cubin

cuda device driver

execute

-code=sm_YY

(real compute architecture)

(YY must be superior or equal to XX)
Compiling for the Right Architecture

nvcc

-arch=\texttt{compute\_XX}

\texttt{x.cu} (device code part)

\texttt{x.ptx}

-arch=\texttt{compute\_YY} \quad \text{(YY must be superior or equal to XX)}

\texttt{Stage 1 (nvvm)}

\texttt{Stage 2 (ptxas)}

\texttt{x.cubin}

virtual compute architecture

real compute architecture

cuda device driver
Compiling for the Right Architecture

▶ Fat binary (multiple architecture binaries in one)

nvcc x.cu -arch=compute_30 -code=compute_30,sm_30,sm_35

- generates exact code for **two Kepler variants**, plus **PTX code for use by JIT** in case a next-generation GPU is encountered

▶ Defining multiple architectures

nvcc -gencode=arch=XX1,code=YY1 -gencode=arch=XX2,code=YY2 ...

▶ Shortcuts

```
nvcc x.cu -arch=sm_35
=> nvcc x.cu -arch=compute_35 -code=sm_35,compute_35
```

```
nvcc x.cu -arch=compute_35
=> nvcc x.c.u -arch=compute_35 -code=compute_35
```
Asynchronous Concurrent Execution

31-10-2016
Asynchronous Concurrent Execution

- A CUDA application can execute several tasks concurrently:
  - Computation on the host
  - Computation on the device
  - Memory transfers from the host to the device
  - Memory transfers from the device to the host
  - Memory transfers within the memory of a given device
  - Memory transfers among devices
Asynchronous Concurrent Execution

Concurrent Execution between Host and Device:

- Kernel launches
- Memory copies within a single device's memory
- Memory copies from host to device of a memory block of 64 KB or less
- Memory copies performed by functions that are suffixed with Async
- Memory set function calls

Return control to the host thread before the device completes the requested task.
Example: Concurrent Execution between Host and Device

cudaMemcpy(..., cudaMemcpyHostToDevice);

kernel1 <<<100, 512 >>>(...)

do work on host

cudaMemcpy(..., cudaMemcpyDeviceToHost);
Example: Concurrent Execution between Host and Device

2 possibilities:

- or

HtoD 1 | kernel 1 | DtoH 1

work on host

HtoD 1 | kernel 1 | DtoH 1

work on host
Asynchronous Concurrent Execution

Concurrent Execution on GPU:

- A device can run several kernels simultaneously

- Kernel executions and data transfers
  - require page locked memory on the host

- A host to device and a device to host data transfer
  - require page locked memory on the host
Concurrent Execution on GPU: Streams

- CUDA stream:
  - a stream is set of different commands which will be executed sequentially on the GPU (memory copies, kernels)
  - commands from different streams are performed concurrently
    - no specific order

- Stream are declare with a specific type:
  - cudaStream_t

- Creation:
  - cudaStreamCreate()

- Destruction:
  - cudaStreamDestroy()
    - synchronize streams
Example: 1 stream

cudaStream_t stream [1];

//stream creation
cudaStreamCreate (& stream [0]);

float * hostPtr;
cudaMallocHost (& hostPtr , 2 * size ); //page-locked memory allocation

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream [0]);
kernel1 <<<100, 512 , 0, stream [0]>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream [0]);

cudaMemcpyAsync ( ... , cudaMemcpyHostToDevice , stream [0]);
kernel2 <<<100, 512 , 0, stream [0]>>>(...)
cudaMemcpyAsync ( ... , cudaMemcpyDeviceToHost , stream [0]);

do work on host

///stream destruction
cudaStreamDestroy ( stream [0]);
Example: 1 stream

1 synchronous stream:

```
stream 0

HtoD 1  kernel 1  DtoH 1  HtoD 2  kernel 2  DtoH 2

host

work on host

time
```
Example: 2 streams

cudaStream_t stream [2];

//streams creation
for ( int i = 0; i < 2; ++i)
    cudaStreamCreate (& stream [i]);

float * hostPtr;
cudaMallocHost (& hostPtr, 2 * size); //page-locked memory allocation

cudaMemcpyAsync (... , cudaMemcpyHostToDevice , stream [0]);
kernel1 <<<100, 512 , 0, stream [0]>>>(...)
cudaMemcpyAsync (... , cudaMemcpyDeviceToHost , stream [0]);

cudaMemcpyAsync (... , cudaMemcpyHostToDevice , stream [1]);
kernel2 <<<100, 512 , 0, stream [1]>>>(...)
cudaMemcpyAsync (... , cudaMemcpyDeviceToHost , stream [1]);

do work on host

//streams destruction
for ( int i = 0; i < 2; ++i)
    cudaStreamDestroy ( stream [i]);
Example: 2 streams

2 streams with asynchronous data transfers:

- **stream 0**
  - HtoD 1
  - kernel 1
  -DtoH 1

- **stream 1**
  - HtoD 2
  - kernel 2
  -DtoH 2

- **host**
  - work on host
Default Stream

- By default all operations are in the default stream

- All operations in this stream are synchronous:
  - commands are made in order on the GPU

- This implied synchronization can be deactivated with the compilation option: --default-stream-per-thread
Explicit Synchronization

- `cudaDeviceSynchronize()`: synchronization of all operations on the device
- `cudaStreamSynchronize()`: synchronization of all operations of a stream
  - previous commands are done
- `cudaStreamWaitEvent()`: synchronization of a stream based on a event
  - ensure that operations from another stream are made
- `cudaStreamQuery()`: test the status of a stream
  - if all operation are done
Thanks

For more information please contact:
Georges-Emmanuel Moulard
M+ 33 6 85529054
georges-emmanuel.moulard@atos.net

Atos, the Atos logo, Atos Consulting, Atos Worldgrid, Worldline, BlueKiwi, Canopy the Open Cloud Company, Yunano, Zero Email, Zero Email Certified and The Zero Email Company are registered trademarks of Atos. April 2015. © 2015 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

31-10-2016