CUDA/GPGPU Workshop 2012

CUDA/GPGPU Arch&Prog

Chok Yip

Wichita State University
7/11/2012
GPU-Hardware perspective

- GPU as PCI device
  - Original PCI
  - PCIe
- Inside GPU architecture
GPU as PCI device

- Traditional PC connects to devices through 2 nodes (North & South Bridge)
- Southbridge serves as a concentrator for slower I/O devices
- GPU was connected as peripheral devices
GPU as PCI device

- PCI Device are mapped into CPU’s physical address space
- Addresses assigned to the PCI devices at boot time
  - Devices poll for input
PCle Bus

• PCle now serves as backbone
  – Northbridge/Southbridge are PCle switches
  – Some PCle cards are PCI cards with PCI to PCle bridge
GPU-PCI-E

- Devices connect to same switch
  - Each card links to central switch
  - Dedicated bus(links) for PCI-E devices
  - Packet switches messages from virtual channel
GPU Architecture

• GPU (the chip itself) consists of group of Streaming Multiprocessors (SM)
• Inside each SM
  – 32 cores (sharing the same instruction)
  – 64KB shared memory (shared among the 32 cores)
  – 32k 32bit registers
  – 2 warp schedulers (To schedule instructions)
  – 4 special function units
GPU Architecture

- Each core
  - Logic
  - Move, Compare
  - Branch
  - Fused Multiply-Add for single (32bit) and double (64bit) precision
Therefore, one SM can perform
- 32x32bit floating point/clock
- 16x64bit floating point/clock
- 32x32bit Integer/clock

Each SM shares the same instruction

Each GPU Chip have different number of SM

On a high-end consumer card
- GTX 480 has 15 SM
- GTX 680 has 96 SM
- This translates to performance
CPU and GPU Comparison

- **CPU**: Low Latency
  - Fast access to memory through caching
  - Control logic for out-of-order and speculative execution
  - High clockspeed

- **GPU**: High Throughput
  - High latency, high throughput
  - Relatively simple control logic
  - Massively parallel ALUs
CPU and GPU Comparison

- CPU architecture reduces memory access time within each thread to enhance throughput
- GPU architecture increases number of concurrent threads to enhance throughput

Optional:

CPU threads are equivalent to Warps (Warps as in collection of threads)
Each Warp will process 32 CUDA Threads concurrently
Threads and Blocks

• GPU threads are lightweight
  – GPU executes massive number of threads concurrently

• Low branching performance
Threads and Blocks

- CUDA Threads are grouped into blocks
  - This is to optimize the use of memory
- Instruction sent by host to GPU is called a Kernel
  - GPU sees a kernel as a grid of blocks of threads
Execution of Threads and Blocks

Each CUDA thread will execute in one core.

Depending on memory requirements of a kernel, multiple block may execute on each SM.

Each kernel can only be executed by one device (unless programmer’s intervention).
Multiple kernels may be executed at one time.
Execution of Kernel

• The host (CPU) executes a kernel in GPU in 4 steps

CPU allocates and copies data to GPU
On CUDA API:
  cudaMalloc()
cudaMemCpy()
Execution of Kernel

- CPU Sends function parameters and instructions to GPU
- CUDA API:
  
  \[ \text{myFunc} \lll \text{Blocks, Threads} \rrr (\text{parameters}) \]
Execution of Kernel

- GPU executes instruction as scheduled in warps
- Results will need to be copied back to Host memory (RAM) using `cudaMemCpy()`
Questions?

References (including images):
• http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/06-cuda-overview.pdf
CUDA Memory

• Memory Hierarchy of nVidia GPU
  – Speed:
  – Registers > Shared Memory > Constant Cache > Texture Memory > Device Memory

• Memory Sizes:
  – Registers (per SM) 32kb
  – Shared Mem (per SM) 64KB
  – Constant
  – Texture
  – Device Memory (per card) up to 6GB
CUDA Memory

• Recall: each block consists of a set of threads
• Each Thread is given a small piece of Local memory
• Blocks do not span more than one SM
• Each Block owns a shared memory
Memory Hierarchy

- Local memory for each thread is private
  - Lifespan only during thread execution
- Shared Memory is accessible among threads for each block
  - Lifespan only during kernel call
- In CUDA API:
  ```
  __shared__ int a[SIZE];
  ```
Memory Hierarchy

• Lifespan of Global Memory:
  – Global Memory will be reserved until programmer uses `cudaFree()`

• Global Memory is accessible among blocks
  – Higher latency
  – Off-chip memory
  – `cudaMalloc()` allocates memory here
• Note that every GPU board has its own memory
• Programmer in CUDA is responsible for memory allocation and free operations

Chok
A CUDA Example

• CUDA Kernel consists of two parts

1. `void add<<<1, N>>>(int dev_a, int dev_b, int dev_c)`
   
   N tells CUDA how many threads to execute this function on
   
   In this case this generates N threads each block

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

   this is a CUDA Kernel (GPU Code)

• The number of thread blocks in a grid and the number of threads in a thread block is controlled by the programmer passing in variables to the kernel launch command.
A CUDA Example

• With add() running in parallel...let’s do vector addition
• Kernel can refer to its thread’s index with the variable threadIdx.x

• Each thread adds a value from a[] and b[], storing the result in c[]:

  ```c
  __global__ void add( int*a, int*b, int*c ) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
  }
  ```
• By using threadIdx.x to index arrays, each thread handles different indices
A CUDA Example

• We write this code:

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

```c
void add<<<1, N>>>(int dev_a, int dev_b, int dev_c)
```

• N=4, This is what runs in parallel on the device:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[0] = a[0] + b[0];</td>
<td>c[1] = a[1] + b[1];</td>
</tr>
<tr>
<td>Thread 2</td>
<td>Thread 3</td>
</tr>
</tbody>
</table>
__global__ void add( int*a, int*b, int*c ) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}

int main( void ) {
    int a, b, c; // host copies of a, b, c
    int* dev_a, *dev_b, *dev_c; // device copies of a, b, c
    int size = sizeof( int); // we need space for an integer
    // allocate device copies of a, b, c
    cudaMalloc( (void**) &dev_a, size );
    cudaMalloc( (void**) &dev_b, size );
    cudaMalloc( (void**) &dev_c, size );
a = 2;
b = 7;
A CUDA Example

// copy inputs to device
cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice);

// launch add() kernel on GPU, passing parameters
add<<< 1, 1 >>>( dev_a, dev_b, dev_c);

// copy device result back to host copy of c
cudaMemcpy( &c, dev_c, size, cudaMemcpyDeviceToHost);
cudaFree( dev_a);
cudaFree( dev_b);
cudaFree( dev_c);
Return 0;
}
Another Example

• Matrix Multiplication
  – Problem: Recall that each block can hold maximum of 512 threads
    • What if matrix contains more than 512 elements?
  – Solution: Group threads into blocks
Another Example

• In this example, we will attempt to group 16x16 matrices into 16 4x4 tiles

// CUDA Kernel
__global__ void matrixMul(float* C, float* A, float* B, int wA, int wB)
{
    // 1. 2D Thread ID
    int tx = blockIdx.x * TILE_SIZE + threadIdx.x;
    int ty = blockIdx.y * TILE_SIZE + threadIdx.y;
    .
    .
}
Another Example

• Calling the kernel

```c
// setup execution parameters
dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
dim3 grid(WC / threads.x, HC / threads.y);

//BLOCK_SIZE = 16, WC = 16, HC = 16

// execute the kernel
matrixMul<<<grid, threads>>>(d_C, d_A, d_B, WA, WB);
```
CUDA/GPGPU Arch & Prog

THANK YOU