Overview of GPU Architecture and CUDA Programming Models
Processor’s Trend

- CPU: Increase # of cores, Increase vector width (SSE)
- GPU: caches, Superscalar
- CPGPU
- Embedded systems
GPU Architecture Trend

- Fixed pipelines → programmable cores → unified programmable cores → more cores → GPGPU support
Overview of GPU (Tesla) Architecture

- Streaming Multiprocessor
- Streaming Multiprocessor
- Streaming Multiprocessor
- Interconnection Network
- Global Memory (Device memory)

SIMD Execution Unit

- PC
- I-Cache
- Decoder
- Shared Memory
- Stream Processor
- Stream Processor
- Stream Processor
Parallel Programming Models Based on Memory Spaces

Shared memory

Distributed memory

CUDA

PCI-E

Georgia Tech

comparch
CUDA Programming Model

Host

Global Memory
Constant Memory
Texture Memory

Block (0, 0)
- Thread (0, 0)
- Thread (1, 0)
- Thread (2, 0)
- Thread (3, 0)

Block (1, 0)
- Thread (0, 0)
- Thread (1, 0)

Shared Memory

Registers

Local Memory

Shared Memory

Registers

Local Memory

Block (0, 0)

Block (1, 1)

Thread (0, 0) is connected to
Thread (1, 0) and
Thread (2, 0)

Thread (1, 0) is connected to
Thread (2, 0)

Thread (2, 0) is connected to
Thread (3, 0)

Thread (0, 1) is connected to
Thread (1, 1)

Thread (1, 1) is connected to
Thread (2, 1)

Thread (2, 1) is connected to
Thread (3, 1)

Thread (0, 2) is connected to
Thread (1, 2)

Thread (1, 2) is connected to
Thread (2, 2)

Thread (2, 2) is connected to
Thread (3, 2)

Block (1, 0)

Grid

Block (0, 0) is connected to
Block (1, 0)

Block (1, 0) is connected to
Block (2, 0)

Block (2, 0) is connected to
Block (1, 1)

Block (1, 1) is connected to
Block (2, 1)

Block (2, 1) is connected to
Block (2, 0)

(0, 0) is connected to
(1, 0)

(1, 0) is connected to
(0, 0)

(0, 1) is connected to
(1, 1)

(1, 1) is connected to
(0, 0)

(2, 0) is connected to
(1, 1)

(1, 1) is connected to
(2, 0)

(0, 2) is connected to
(1, 2)

(1, 2) is connected to
(0, 2)

(2, 2) is connected to
(1, 2)

(1, 2) is connected to
(2, 2)
Kernel Creation

• A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50);    // 5000 thread blocks
dim3 DimBlock(4, 8, 8);   // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```
Elementwise Matrix Addition

- Let’s assume $N=16$, $blockDim=4 \rightarrow 4$ blocks

- $blockIdx.x = 0$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $Idx = 0,1,2,3$

- $blockIdx.x = 1$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $Idx = 4,5,6,7$

- $blockIdx.x = 2$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $Idx = 8,9,10,11$

- $blockIdx.x = 3$
  - $blockDim.x = 4$
  - $threadIdx.x = 0,1,2,3$
  - $Idx = 12,13,14,15$
**Elementwise Matrix Addition**

**CPU Program**

```c
void add matrix ( float *a, float *b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
    }
}

int main () {
    add matrix (a, b, c, N);
}
```

**GPU Program**

```c
__global__ add_matrix ( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize) ;
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
Warp (CUDA’s Term)

- **Warp is the basic unit of execution**
  - A group of threads (e.g. 32 threads for the Tesla GPU architecture)

Warp Execution

Programmer specifies the # threads

Finite number of streaming processors

SIMD Execution Unit

Sources ready

Inst 1
Inst 2
Inst 3

One warp

One warp

One warp
# OpenCL vs. CUDA

<table>
<thead>
<tr>
<th></th>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution Model</strong></td>
<td>Work-groups/work-items</td>
<td>Block/Thread</td>
</tr>
<tr>
<td><strong>Memory model</strong></td>
<td>Global/constant/local/private</td>
<td>Global/constant/shared/local + Texture</td>
</tr>
<tr>
<td><strong>Memory consistency</strong></td>
<td>Weak consistency</td>
<td>Weak consistency</td>
</tr>
<tr>
<td><strong>Synchronization</strong></td>
<td>Synchronization using a work-group barrier (between work-items)</td>
<td>Using synch_threads Between threads</td>
</tr>
</tbody>
</table>
GPGPU Friendly Features in Fermi

- Improve double precision
- ECC support
- Cache hierarchy
- More shared memory
- Fast atomic operations
- Dual warp scheduler (superscalar)
- Concurrent kernel execution
- Fully pipelined integer unit (fast INT operations)
OCCUPANCY
**Occupancy**

- Shows how many blocks are assigned to the SM
- Programmer specifies the number of threads per block

---

**SM**

- **Registers**
- **Shared Memory**

100% Occupancy

**SM**

- **Registers**
- **Shared Memory**

Only one block is allocated

- Register requirements per block
- Shared memory requirements per block
Higher Occupancy

- Better processor utilization
- Hide the memory latency

Processor is not utilized

Warp 1: Processor is not utilized
Warp 2: Processor is not utilized
Warp 3: Processor is not utilized
Warp 4: Better utilization!
Warp 5: Better utilization!
High Occupancy ≠ High Performance

- Programmers try to optimize programs for occupancy
- No performance improvement from increased occupancy