CS454 Topics in Advanced Computer Science
Introduction to NVIDIA CUDA
Chengyu Sun
California State University, Los Angeles

Super Computer on a Chip

<table>
<thead>
<tr>
<th>GPU Model</th>
<th>Number of Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 275, 280, 285</td>
<td>240</td>
</tr>
<tr>
<td>GeForce GTX 260</td>
<td>192</td>
</tr>
<tr>
<td>GeForce GTX 250</td>
<td>128</td>
</tr>
<tr>
<td>GeForce 9800 GTX</td>
<td>128</td>
</tr>
<tr>
<td>GeForce 9800 GT</td>
<td>112</td>
</tr>
<tr>
<td>GeForce 9600 GSO</td>
<td>96</td>
</tr>
<tr>
<td>GeForce 9600 GT</td>
<td>64</td>
</tr>
<tr>
<td>GeForce 8800 GTX</td>
<td>128</td>
</tr>
<tr>
<td>GeForce 8800 GT</td>
<td>112</td>
</tr>
</tbody>
</table>

GPU vs. CPU

Old approach: map the computation task to some graphics operation
- DirectX
- OpenGL

New approach: use a general-purposed API for GPU
- NVIDIA CUDA
- ATI Steam SDK

Why GPU Is Faster

Use GPU for General-Purposed Computing

Vector Add

```
Input : A[], B[], N
Output: C[]

for ( int i = 0 ; i < N ; ++i )
{
    C[i] = A[i] + B[i];
}
```
Vector Add in CUDA

```c
// kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    // kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

About the Example

- CUDA is a C API with some extended syntax
- Kernel: a C function that can be executed N times by N different threads in parallel
- Each thread has a unique id threadIdx

ThreadIdx and Thread Blocks

- Threads can be created in 1D, 2D, or 3D block
- Example:

  ```c
  KernelFunction<<<1,N>>>( arguments );
  ````
  
  Create a 1D block of threads, and each thread can be identified by threadIdx.x

Matrix Add

```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    // dimBlock is identified by threadIdx.x
    dim3 dimBlock(N, N);
    MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

Thread Grid

- Multiple thread blocks form a thread grid
- Each block in the grid can be identified by blockIdx
- Example:

  ```c
  dim3 dimGrid(3,2);
  dim3 dimBlock(4,3);
  KernelFunction<<<dimGrid, dimBlock>>>( arguments );
  ```

Thread Grid Example ...

- A grid has 2x3 blocks
... Thread Grid Example

- A block has 3x4 threads

```
Block (1, 1)

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

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

Thread (0, 2) Thread (1, 2) Thread (2, 2) Thread (3, 2)
```

Matrix Add Using Thread Grid

```c
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}
```

```c
int main()
{
    dim3 dimBlock(16, 16);
    dim3 dimGrid((N + dimBlock.x – 1) / dimBlock.x, (N + dimBlock.y – 1) / dimBlock.y);
    MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

About CUDA Threads

- Overhead of creating a thread is extremely low
- There can be more threads than the actual number of processors

About Thread Blocks

- All threads in a block are expected to reside on the same processor
  - Up to 512 threads per block
- Threads within the same block share memory and can be synchronized
- Thread blocks are required to be executed independently in any order

CUDA Program Execution ...

- Host + Device

```
Host + Device
```

... CUDA Program Execution

- Host + Device

```
Host + Device
```

```c
C-Program Sequential Execution
```

```c
Device
```

```c
Block (0, 0) Block (1, 0) Block (2, 0)
Block (0, 1) Block (1, 1) Block (2, 1)
Block (0, 2) Block (1, 2) Block (2, 2)
```

```c
Serial code
```

```c
Parallel Kernel Invocation
```

```c
Serial code
```

```c
Device
```

```c
Block (0, 0) Block (1, 0) Block (2, 0)
Block (0, 1) Block (1, 1) Block (2, 1)
Block (0, 2) Block (1, 2) Block (2, 2)
```
Example: VecAdd.c

- Copy vectors from host memory to device memory
- Copy results from device memory to host memory

Memory

- Host memory
- Device memory
  - Global
    - Not cached, slow
  - Per-block shared
    - On chip, very fast
  - Per-thread local
    - Not cached, as slow as global memory

Example: MatMul1.c

- Use device global memory
- Kernel invocation
- Kernel

Matrix Multiplication Using Shared Memory ...

... Matrix Multiplication Using Shared Memory

- Only $A_{sub}$ and $B_{sub}$ are needed to calculate $C_{sub}$
- Each element in $A_{sub}$ and $B_{sub}$ is used multiple times during the calculation of $C_{sub}$
- How do we get $A_{sub}$ and $B_{sub}$ into shared memory??

Example: MatMul2.c

- __shared__
- __syncthread()
Summary

- Kernel
- Thread grid
- Program execution
- Memories
- Barrier synchronization

References

- NVIDIA CUDA Programming Guide
  Version 2.3