High Performance Computing and GPU Programming

Lecture 2: GPU Core
- GPU Intro Cont.
- Programming Model
- GPU Memory
GPU Intro
• CPU vs. GPU

CPU
SIMD – Single instruction multiple data vector units

GPU
SIMT – Single instruction multiple threads
## GPU Intro

- **CPU vs. GPU**

<table>
<thead>
<tr>
<th></th>
<th>GPU – Tesla K20</th>
<th>CPU – Intel I7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>Main memory</td>
<td>4 (8)</td>
</tr>
<tr>
<td>Memory</td>
<td>5 GB</td>
<td>32 KB L1 cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>256 KB L2 cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 MB L3 shared</td>
</tr>
<tr>
<td>Clock Speed</td>
<td>2.6 GHz</td>
<td>3.2 GHz</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>208 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>FLOPS</td>
<td>$1.17 \times 10^{12}$</td>
<td>$70 \times 10^{9}$</td>
</tr>
</tbody>
</table>
Programming Model
Programming Model

• Three major topics in GPU computing
  – Architecture Management
    • Threads and blocks – How to set-up?
  – Memory Management
    • This is where you get speed!
  – Algorithm Management
    • Optimization and massive parallelism
• SM creates, manages, schedules, and executes threads in groups of parallel threads - Warps
• Warps are not easy
  – Warp size is 32 threads on current GPUs
  – Threads in a warp start together
  – When an SM has a task or block:
    • The threads are split into warps
    • A sort of “scheduling” is done

• Most of the time we have to ignore this
  – Not all problems fit into a multiple of 32!
  – Many papers claim 500x speed-up for matrix operations
    • The cases are for sizes of 32x32, 256x256, 512x512, etc...
// 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 with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
Programming Model

- Memory must be allocated

// Host code
int main()
{
    // Allocate input vectors in host memory
    float* A_h = (float*)malloc(N * sizeof(float));
    float* B_h = (float*)malloc(N * sizeof(float));

    // Initialize input vectors ...

    // Allocate vectors in device memory
    float* A_d, B_d, C_d;
    cudaMalloc(&A_d, N * sizeof(float));
    cudaMalloc(&B_d, N * sizeof(float));
    cudaMalloc(&C_d, N * sizeof(float));

    // Copy vectors from host memory to device memory
    cudaMemcpy(A_d, A_h, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, N * sizeof(float), cudaMemcpyHostToDevice);

    // Invoke kernel
    VecAdd<<<1, N>>>(A_d, B_d, C_d, N);

    // Copy result from device memory to host memory
    cudaMemcpy(C_h, C_d, N * sizeof(float), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaMemcpy(A_d, A_h, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
• Thread Hierarchy
  – Controlled by “dim3” declaration
  – Threads have a limit!

```c
// Kernel definition
__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()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```
Now consider a multi-block multi-thread problem

Break it up!
```c
// Kernel definition
__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];
}

int main()
{
    ...  // Kernel invocation
    dim3 threadsPerBlock(2, 2);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```
GPU Memory
• We will discuss more on threads later
• Introduce memory – Diagram!

- Global Memory
- Texture Memory
- Local Memory
- Shared Memory
- Block 0
- Block 1
- Block 2
GPU Memory

• Global Memory
  – Main GPU memory – but also slow!
  – Try to never run computations here – only in some situations
  – All blocks and all threads

• Texture Memory
  – A little complicated to explain
  – You can read data from here fast!
  – But cannot write data directly
  – All blocks and all threads
GPU Memory

- **Local Memory**
  - Local to each thread in the block
  - Able to communicate – but never do it!
  - Registers are here
  - Very fast

- **Shared Memory**
  - Difficult to use correctly – but very powerful
  - 150x faster than global memory
  - Local to the block
GPU Memory

• Starting an application – We must ...

Global Memory

1. Allocate everything we need to the GPU into global memory
2. You must decide what goes into the texture cache

Texture Memory

3. Now execute a CUDA kernel – everything we need is there
4. Decide what memory you need and where you need it from
5. Run computations – store result into global memory when done
6. Use this memory in other kernels

• There are several deciding factors on where you get your memory and where you store it
GPU Memory

Where to get it?

Global Memory
• Coalesced
• That’s it!

Texture Memory
• Non-Coalesced
• That’s it!
• What is Coalesced?
  – The single most important thing you can do
  – All threads in a HALF Warp access global memory at the same time
    • Again...Warps...
    • How about simple!
  – Neighboring threads access neighboring cells in memory
Where to put it?

Local Memory

• Coalesced access
• One access by thread – then move on!
• Huge performance
• Basically do I need...
  – Coalesced computations?
  – No sharing data?

Shared Memory

• for/do loops
• Required by other blocks
• Required by other threads
• Basically do I need ...
  – Repeated access?
  – Shared access?
Wrap Up

Next time...

- CUDA for you
  - What you need, where to get it, how to install it
- Thread index mapping
  - 2-D or 3-D to 1-D
- Introduce CUDA memory types
  - Texture, local, global, shared
- Program interpolation function (if time permits it)
  - CPU vs. GPU implementation