Optimization Principles and Application
Performance Evaluation of a Multithreaded
GPU Using CUDA
GPU architecture

![Diagram of GPU architecture]

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Size</th>
<th>Hit Latency</th>
<th>Read-Only</th>
<th>Program Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>off-chip</td>
<td>768MB total</td>
<td>200-300 cycles</td>
<td>no</td>
<td>global</td>
</tr>
<tr>
<td>Local</td>
<td>off-chip</td>
<td>up to global</td>
<td>same as global</td>
<td>no</td>
<td>function</td>
</tr>
<tr>
<td>Shared</td>
<td>on-chip</td>
<td>16KB per SM</td>
<td>(\leq ) register latency</td>
<td>no</td>
<td>function</td>
</tr>
<tr>
<td>Constant</td>
<td>on-chip</td>
<td>64KB total</td>
<td>(\leq ) register latency</td>
<td>yes</td>
<td>global</td>
</tr>
<tr>
<td>Texture</td>
<td>on-chip</td>
<td>up to global</td>
<td>(&gt;100) cycles</td>
<td>yes</td>
<td>global</td>
</tr>
</tbody>
</table>
GeForce 8800 GPU

• 16 Streaming multiprocessors
• 8 Streaming processors pr SM
• 8192 registers pr SM
• 768 threads pr SM
• 8 blocks can be run at a time pr SM
• 32 thread warp
Example

• 4K by 4K matrix multiplication
• 768 threads.
• 10 registers pr thread
• Potential throughput of 43.2 FLOPS
• Performance is 10.58 FLOPS
• Global memory access
Tiling

• Doing computations on smaller «tiles»
• Put tiles in shared memory
  • 4x4 – Only 16 threads, half warps
  • 8x8 – Occupies 2 warps, but need 12 blocks to use all threads. Can only use 8.
  • 12x12 – 144 threads which is not divisible by 32 (warp size).
  • 16x16 – 256/32 = 8 warps. Use three blocks: 256*3 = 768
• Reduced global load by a factor of 16
• 46.49 GFLOPS
• 3 blocks/SM * 256 threads/block
  * 11 registers = 8488 registers > 8192
Unrolling

- Unroll the loop
- Removing
  - Loop address calculation instructions
  - Iterator variable increments (register)
- Potential throughput: 93.72 GFLOPS
- Performance: 91.14 GFLOPS

```c
Ctemp = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        shared float As[16][16];
        shared float Bs[16][16];

        // load input tile elements
        As[ty][tx] = A[indexA];
        Bs[ty][tx] = B[indexB];
        indexA += 16;
        indexB += 16 * widthB;

        __syncthreads();

        // compute results for tile
        Ctemp += As[ty][0] * Bs[0][tx];
        ...;
        Ctemp += As[ty][15] * Bs[15][tx];

        __syncthreads();
    }
}
C[indexC] = Ctemp;
```