GPU Computing with CUDA
Lecture 4 - Optimizations

Christopher Cooper
Boston University
August, 2011
UTFSM, Valparaíso, Chile
Outline of lecture

- Recap of Lecture 3
- Control flow
- Coalescing
- Latency hiding
- Occupancy
Recap

- **Shared memory**
  - Small on chip memory
  - Fast (100x compared to global)
  - Allows communication among threads within a block

- **Tiling**
  - Use shared memory as cache
Recap

- Tiling examples for FD
  - Solution 1:
    - Fetching to global in block boundaries
  - Solution 2:
    - Using halo nodes
    - All threads load to shared, only some operate
  - Solution 3:
    - Using halo nodes
    - Load to shared in multiple steps, all threads operate
Recap

- Bank conflicts

- Arrays in shared memory are divided into banks

- Access to different data in the same bank by more than one thread in the same warp creates a bank conflict

- Bank conflicts serializes the access to the minimum number of non-conflicting instructions
Control flow

- If statement
  - Threads are executed in warps
  - Within a warp, the hardware is not capable of executing \textit{if} and \textit{else} statements at the same time!

```c
__global__ void function();
{
    ....

    if (condition)
    {
        ...
    }
    else
    {
        ...
    }
}```
Control flow

- How does the hardware deal with an if statement?
Control flow

- Hardware serializes the different execution paths

Recommendations

- Try to make every thread in the same warp do the same thing
  - If the if statement cuts at a multiple of the warp size, there is no warp divergence and the instruction can be done in one pass
- Remember threads are placed consecutively in a warp (t0-t31, t32-t63, ...)
  - But we cannot rely on any execution order within warps
- If you can’t avoid branching, try to make as many consecutive threads as possible do the same thing
Control flow - An illustration

- Let’s implement a sum reduction

- In a serial code, we would just loop over all elements and add

- Idea:
  - To implement in parallel, let’s take every other value and add in place it to its neighbor
  - To the resulting array do the same thing until we have only one value

```c
__shared__ float partialSum[]
int t = threadIdx.x
for(int stride = 1; stride<blockDim.x; stride*=2)
{
    __syncthreads();
    if (t%(2*stride)==0)
        partialSum[t] += partialSum[t+stride];
}
```
Control flow - An illustration

Thread 0  Thread 2  Thread 4  Thread 6  Thread 8  Thread 10

0 1 2 3 4 5 6 7 8 9 10 11

0+1 2+3 4+5 6+7 8+9 10+11

0...3 4..7 6..7 8..11 8..15

iterations

Array elements

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign
Control flow - An illustration

- Advantages
  - Right result
  - Runs in parallel

- Disadvantages
  - The number of threads decreases per iteration, but we’re using much more warps than needed
    - There is warp divergence in every iteration
  - No more than half the threads per thread are being executed per iteration

- Let’s change the algorithm a little bit...
Control flow - An illustration

- Improved version

  - Instead of adding neighbors, let’s add values with stride half a section away
  
  - Divide the stride by two after each iteration

```c
__shared__ float partialSum[]
int t = threadIdx.x
for(int stride = blockDim.x; stride>1; stride>>=1)
{
    __syncthreads();
    if (t<stride)
        partialSum[t] += partialSum[t+stride];
}
```
Control flow - An illustration

Thread 0

0 1 2 3 ... 13 14 15 16 17 18 19

1 0+16 15+31

3

4
Control flow - An illustration

- 512 elements

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Exec. threads</th>
<th>Warps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Threads > Warp

Threads < Warp

Warp divergence!
Control flow - An illustration

- We get warp divergence only for the last 5 iterations
- Warps will be shut down as the iteration progresses
  - This will happen much faster than for the previous case
  - Resources are better utilized
  - For the last 5 iterations, only 1 warp is still active
Control flow - Loop divergence

- Work per thread data dependent

```c
__global__ void per_thread_sum (int *indices, float *data, float *sums) {
    ...
    for (int j=indices[i]; j<indices[i+1]; j++)
    {
        sum += data[j];
    }
    sums[i] = sum
}
```

David Tarjan - NVIDIA
Control flow - Loop divergence

- Warp won't finish until the last thread finishes
  - Warp will be dragged

- Possible solution:
  - Try flatten peaks by making threads work in multiple data
Memory coalescing

- Global memory is accessed in chunks of aligned 32, 64 or 128 bytes
- Following protocol is used to issue a memory transaction of a half warp (valid for 1.X)
  - Find memory segment that contains address requested by the lowest numbered active thread
  - Find other active threads whose requested address lies in same segment, and reduce transaction size if possible
  - Do transaction. Mark serviced threads as inactive.
  - Repeat until all threads are serviced
- Worse case: fetch 32 bytes, use only 4 bytes: 7/8 wasted bandwidth!
Memory coalescing

- Access pattern visualization
  - Thread 0 is lowest active, accesses address 116
    - Belongs to 128-byte segment 0-127
Memory coalescing

- Simple access pattern

One 64 byte transaction

Will be looking at compute capability 1.X examples
Memory coalescing

- Sequential but misaligned

One 128 byte transaction

One 64 byte transaction and one 32 byte transaction
Memory coalescing

- Strided access

One 128 byte transaction, but half of bandwidth is wasted
Memory coalescing

Example: Copy with stride

```c
__global__ void strideCopy(float *odata, float *idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x+threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

Copy with Stride

![Diagram showing the effective bandwidth for different strides with GTX 280 and GTX 8800]
Memory coalescing

2.X architecture

- Global memory is cached
  - Cached in both L1 and L2: 128 byte transaction
  - Cached only in L2: 32 byte transaction
- Whole warps instead of half warps
Memory coalescing - SoA or AoS?

- Array of structures

```c
struct record
{
    int key;
    int value;
    int flag;
};

record *d_record;
cudaMalloc((void**) &d_records, ...);
```
Memory coalescing - SoA or AoS?

- Structure of array

```c
struct SoA {
    int *key;
    int *value;
    int *flag;
};

SoA *d_AoA_data;
cudaMalloc((void**) &d_SoA_data.keys, ...);
cudaMalloc((void**) &d_SoA_data.value, ...);
cudaMalloc((void**) &d_SoA_data.flag, ...);
```

David Tarjan - NVIDIA
Memory coalescing - SoA or AoS?

- cudaMalloc guarantees aligned memory, then accessing the SoA will be much more efficient

```c
__global__ void bar (record *AoS_data, SoA SoA_data)
{
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    // AoS wastes bandwidth
    int key = AoS_data[i].key;
    // SoA efficient use of bandwidth
    int key_better = SoA_data.keys[i];
}
```

David Tarjan - NVIDIA
Latency hiding

- A warp is not scheduled until all threads have finished the previous instruction.

- These instructions can have high latency (e.g., global memory access).

- Ideally, one wants to have enough warps to keep the GPU busy during the waiting time.

Latency hiding
Latency hiding

- How many instructions do we need to hide latency of L clock cycles?
  - 1.X: L/4 instructions
    - SM issues one instruction per warp over four cycles
  - 2.0: L instructions
    - SM issues one instruction per warp over two clock cycles for two warps at a time
  - 2.1: L/2 instructions
    - SM issues a pair of instructions per warp over two clock cycles for two warps at a time
Latency hiding

Doing the exercise

- Instruction takes 22 clock cycles to complete
  - 1.X: we need 6 available warps to hide latency
  - 2.X: we need 22 available warps to hide latency
- Fetch to global memory: ~600 cycles!
  - Depends on FLOPs per memory access (arithmetic intensity)
    - eg. if ratio is 15:
      - 10 warps for 1.X
      - 40 warps for 2.X
**Occupancy**

- Occupancy is the ratio of resident warps to maximum number of resident warps

- Occupancy is determined by the resource limits in the SM
  - Maximum number of blocks per SM
  - Maximum number of threads
  - Shared memory
  - Registers
**Occupancy**

- --ptxas-options=-v for compiler to tell you register and shared memory usage

- Use CUDA Occupancy Calculator

- The idea is to find the optimal threads per block for maximum occupancy

- Occupancy != performance: but low occupancy codes have a hard time hiding latency

- Demonstration
Measuring performance

- Right measure depends on program
  - Compute bound: operations per second
    - Performance given by ratio of number of operations and timing of kernel
    - Peak performance depends on architecture: Fermi ~1TFLOP/s
  - Memory bound: bandwidth
    - Performance given by ratio of number of global memory accesses and timing of kernel
    - Peak bandwidth for GTX 280 (1.107GHz with 512-bit width)
      \[ 1107 \times 10^6 \times (512/8) \times 2/10^9 = 141.7 \text{ GB/s} \]