CS 193G

Lecture 5: Performance Considerations
Always measure where your time is going!
- Even if you think you know where it is going
- Start coarse, go fine-grained as need be

Keep in mind Amdahl’s Law when optimizing any part of your code
- Don’t continue to optimize once a part is only a small fraction of overall execution time
Performance Considerations

- Memory Coalescing
- Shared Memory Bank Conflicts
- Control-Flow Divergence
- Occupancy
- Kernel Launch Overheads
MEMORY COALESCING
Memory Coalescing

- Off-chip memory is accessed in chunks
  - Even if you read only a single word
  - If you don’t use whole chunk, bandwidth is wasted
- Chunks are aligned to multiples of 32/64/128 bytes
  - Unaligned accesses will cost more
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 64B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 0 is lowest active, accesses address 116
- 128-byte segment: 0-127 (reduce to 32B)
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255
Threads 0-15 access 4-byte words at addresses 116-176

- Thread 3 is lowest active, accesses address 128
- 128-byte segment: 128-255 (reduce to 64B)

![Diagram showing 64B transaction](image)
Consider the stride of your accesses

```c
__global__ void foo(int* input,
                    float3* input2)
{
    int i = blockDim.x * blockIdx.x
            + threadIdx.x;
    // Stride 1
    int a = input[i];
    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];
    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```
Example: Array of Structures (AoS)

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

record *d_records;
cudaMalloc((void**)&d_records, ...);
```
Example: Structure of Arrays (SoA)

```c
struct SoA
{
    int * keys;
    int * values;
    int * flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
```
Example: SoA vs. AoS

```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];
}
```
Memory Coalescing

Structure of array is often better than array of structures

- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
SHARED MEMORY BANK
CONFLICTS
Shared Memory

- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
  - Threads can each access different banks
  - Or can all access the same value

- Consecutive words are in different banks

- If two or more threads access the same bank but different value, get bank conflicts
Bank Addressing Examples
Bank Addressing Examples

2-way Bank Conflicts

8-way Bank Conflicts
Trick to Assess Impact On Performance

- Change all SMEM reads to the same value
  - All broadcasts = no conflicts
  - Will show how much performance could be improved by eliminating bank conflicts

- The same doesn’t work for SMEM writes
  - So, replace SMEM array indices with `threadIdx.x`
  - Can also be done to the reads
Additional “memories”

- **texture** and **__constant__**
- Read-only
- Data resides in global memory
- Different read path:
  - includes specialized caches
Constant Memory

Data stored in global memory, read through a constant-cache path
- `__constant__` qualifier in declarations
- Can only be read by GPU kernels
- Limited to 64KB
- To be used when all threads in a warp read the same address
- Serializes otherwise

Throughput:
- 32 bits per warp per clock per multiprocessor
CONTROL FLOW

DIVERGENCE
Control Flow

- Instructions are issued per 32 threads (warp)

- Divergent branches:
  - Threads within a single warp take different paths
    - if-else, ...
  - Different execution paths within a warp are serialized

- Different warps can execute different code with no impact on performance
Control Flow

- Avoid diverging within a warp

  Example with divergence:
  ```
  if (threadIdx.x > 2) {...}
  else {...}
  ```

  Branch granularity < warp size

- Example without divergence:
  ```
  if (threadIdx.x / WARP_SIZE > 2) {...}
  else {...}
  ```

  Branch granularity is a whole multiple of warp size
Example: Divergent Iteration

```c
__global__ void per_thread_sum(int *indices,
                               float *data,
                               float *sums)
{
    ... 
    // number of loop iterations is data dependent
    for(int j=indices[i];j<indices[i+1]; j++)
    {
        sum += data[j];
    }
    sums[i] = sum;
}
```
Iteration Divergence

- A single thread can drag a whole warp with it for a long time

- Know your data patterns

- If data is unpredictable, try to flatten peaks by letting threads work on multiple data items
OCCUPANCY
Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM *
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

Instruction: 1 2 3 4 5 6 1 2 1 2 3 4 1 2 7 8 1 2 1 2 3 4

TB = Thread Block, W = Warp

TB1, W1 stall
TB2, W1 stall
TB3, W2 stall
Thread Scheduling

What happens if all warps are stalled?
No instruction issued $\rightarrow$ performance lost

Most common reason for stalling?
Waiting on global memory

If your code reads global memory every couple of instructions
You should try to maximize occupancy
What determines occupancy?

- Register usage per thread & shared memory per thread block
Pool of registers and shared memory per SM

- Each thread block grabs registers & shared memory
- If one or the other is fully utilized -> no more thread blocks
Resource Limits (2)

- Can only have 8 thread blocks per SM
  - If they’re too small, can’t fill up the SM
  - Need 128 threads / TB (gt200), 192 thread / TB (gf100)

- Higher occupancy has diminishing returns for hiding latency
Hiding Latency with more threads

Throughput, 32-bit words

GB/s vs. Threads Per Multiprocessor
How do you know what you’re using?

- Use `nvcc -Xptxas -v` to get register and shared memory usage

- Plug those numbers into CUDA Occupancy Calculator
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

### Varying Block Size

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>512%</td>
</tr>
</tbody>
</table>

### Varying Register Count

<table>
<thead>
<tr>
<th>Register Count</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>512%</td>
</tr>
</tbody>
</table>

### Varying Shared Memory Usage

<table>
<thead>
<tr>
<th>Shared Memory</th>
<th>Occupancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>640</td>
<td>512%</td>
</tr>
</tbody>
</table>

### Allocation Per Thread Block

<table>
<thead>
<tr>
<th>Warps</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>3594</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>1024</td>
</tr>
</tbody>
</table>

These data are used in computing the occupancy data in blue.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CUDA GPU Occupancy Calculator</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Just follow steps 1, 2, and 3 below! (or click here for help)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1.) Select Compute Capability (click):</td>
<td>1.3</td>
</tr>
<tr>
<td>2.) Enter your resource usage:</td>
<td></td>
</tr>
<tr>
<td>Threads Per Block</td>
<td>128</td>
</tr>
<tr>
<td>Registers Per Thread</td>
<td>25</td>
</tr>
<tr>
<td>Shared Memory Per Block (bytes)</td>
<td>640</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(Don't edit anything below this line)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3.) GPU Occupancy Data is displayed here and in the graphs:</td>
<td></td>
</tr>
</tbody>
</table>
3.) GPU Occupancy Data is displayed here and in the graphs:

| Active Threads per Multiprocessor | 512 |
| Active Warps per Multiprocessor   | 16  |
| Active Thread Blocks per Multiprocessor | 4   |
| Occupancy of each Multiprocessor  | 50% |

Physical Limits for GPU Compute Capability: 1.3

| Threads per Warp         | 32   |
| Warps per Multiprocessor | 32   |
| Threads per Multiprocessor | 1024 |
| Thread Blocks per Multiprocessor | 8   |
| Total # of 32-bit registers per Multiprocessor | 16384 |
| Register allocation unit size | 512  |
| Register allocation granularity | block |
| Shared Memory per Multiprocessor (bytes) | 16384 |
| Shared Memory Allocation unit size | 512  |
| Warp allocation granularity (for register allocation) | 2   |

Allocation Per Thread Block

| Warps | 4   |
| Registers | 3584 |
| Shared Memory | 1024 |

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor

| Limited by Max Warps / Blocks per Multiprocessor | 8   |
| Limited by Registers per Multiprocessor         | 4   |
| Limited by Shared Memory per Multiprocessor     | 16  |

Thread Block Limit Per Multiprocessor highlighted RED
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

**Varying Block Size**

**Varying Register Count**

**Varying Shared Memory Usage**
How to influence how many registers you use

- Pass option `-maxrregcount=X` to nvcc

- This isn’t magic, won’t get occupancy for free

- Use this very carefully when you are right on the edge
KERNEL LAUNCH OVERHEAD
Kernel Launch Overhead

- Kernel launches aren’t free
  - A null kernel launch will take non-trivial time
  - Actual number changes with HW generations and driver software, so I can’t give you one number

- Independent kernel launches are cheaper than dependent kernel launches
  - Dependent launch: Some readback to the cpu

- If you are launching lots of small grids you will lose substantial performance due to this effect
Kernel Launch Overheads

- If you are reading back data to the CPU for control decisions, consider doing it on the GPU.

- Even though the GPU is slow at serial tasks, can do surprising amounts of work before you used up kernel launch overhead.
Performance Considerations

- Measure, measure, then measure some more!
- Once you identify bottlenecks, apply judicious tuning
  - What is most important depends on your program
  - You’ll often have a series of bottlenecks, where each optimization gives a smaller boost than expected
Questions?
Backup
Shared Memory

**Uses:**
- Inter-thread communication within a block
- Cache data to reduce global memory accesses
- Use it to avoid non-coalesced access

**Organization:**
- 16 banks, 32-bit wide banks (Tesla)
- 32 banks, 32-bit wide banks (Fermi)
- Successive 32-bit words belong to different banks

**Performance:**
- 32 bits per bank per 2 clocks per multiprocessor
- smem accesses are per 16-threads (half-warp)
- **serialization:** if \( n \) threads (out of 16) access the same bank, \( n \) accesses are executed serially
- **broadcast:** \( n \) threads access the same word in one fetch
__global__ void per_thread_sum(...)
{
    while(!done)
    {
        for(int j=indices[i];
            j<min(indices[i+1],indices[i]+MAX_ITER);
            j++)
        {
            ...}
    }
}