Device Memory Throughput and Latency

ECE 6397
Electrical and Computer Engineering
University of Houston
Dr. David Mayerich
Thread Scheduling

• Threads within a block are divided into warps
  • this is a feature of modern devices – it is not part of the CUDA specification
  • currently 32 threads per warp (Pascal architecture)
  • each warp executes a single instruction
  • this instruction is provided by the warp scheduler

• Warps are sequential based on threadIdx
  • threads 0 to 31 are assigned to a single warp (as are 32 to 63, etc.)
  • warps belonging to the same block execute on the same SM

• more warps are assigned to an SM than the SM can process simultaneously
  • this is how the GPU deals with (hides) latency
Warp Scheduling – Latency Tolerance

- *latency tolerance* – filling latency by executing other threads, thereby making the processor more robust to stalls

- A large number of executions are required to hide a global memory fetch (≈300 clock cycles)

- If there are a large number of global fetches in a kernel, the chance of stalling increases substantially
  - remember that each warp is still running the same kernel

- Your first step in optimization is minimizing global memory fetches
Device Memory Transactions

- Global memory is part of device memory
  - large chunk of off-chip memory
  - this is where cudaMemcpy(⋯) copies your data
- Device memory accesses occur via transactions
  - There are 3 transaction sizes: 32-, 64-, and 128-byte
  - Any access to device memory must:
    - (a) use a transaction of one of these sizes
    - (b) the transaction must be “aligned” – meaning the address must be divisible by the transaction size
- Global memory fetches within a warp are automatically combined into transactions
  - the fetches are coalesced into one or more transactions
  - the GPU will attempt to minimize the number of transactions created
Assume that the specified threads require the following data from global memory:

How many transactions will be issued from the warp?
1x 64-byte transaction, 1x 32-byte transaction

How much additional (unnecessary) data will be copied?
16 bytes from the first transaction + 8 byte from the second = 24 bytes
Device Memory Transactions – Examples

Assume that the user is processing the red channel of an RGBA image represented using float32 values:

<table>
<thead>
<tr>
<th>R0</th>
<th>G0</th>
<th>B0</th>
<th>A0</th>
<th>R1</th>
<th>G1</th>
<th>B1</th>
<th>A1</th>
<th>R2</th>
<th>G2</th>
<th>B2</th>
<th>A2</th>
<th>R3</th>
<th>G3</th>
<th>B3</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
<td>t0</td>
</tr>
</tbody>
</table>

How many transactions are required?
1x 128-byte transaction

How much unused data is copied?
24 bytes per thread – 4x what is actually needed

Consider case of completely coalesced warp transactions that are unaligned:

<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
</tr>
</thead>
</table>

How many transactions are required?
1x 128-byte transaction

How much unused data is copied?
32 bytes
Global Memory Instructions

• Transactions are composed of *global memory instructions*
  • global memory instructions can request word sizes of 1, 2, 4, 8, or 16 bytes

• Global memory instructions must be *naturally aligned*
  • an $n$-byte read/write instruction must start at an address $A$ divisible by $n$
  • if the data is not naturally aligned, it requires multiple memory instructions
  • a warp can only execute one memory instruction from each thread at a time

• Multiple memory instructions from a single thread cannot be coalesced into a transaction!
Global Memory Instructions

- All data requested could fit into a single 128-byte transaction
- Data for $t_0$ fits within a 4-byte instruction and is aligned
  - the fetch can be fulfilled with one instruction
- Despite being 4-bytes, data for $t_1$ and $t_2$ are not aligned
  - $t_1$ and $t_2$ each require two memory instructions
  - $t_0$ will be stalled waiting for the second memory instructions
Aligning Memory Instructions

• The GPU handles alignment of standard data types: char, short, int, long, size_t, float, double

• Pointers returned from cudaMemcpy(⋯) are aligned to 256-bytes
  cudaMemcpy(&gpuPtr, N * sizeof(float));
  • this is sufficient for all standard data types

• As a programmer, you will only have to worry about structures that you create
  __device__ struct badAlign{
    float a;  //the first float in an array will be aligned
    char b;   //but this will throw off the rest
  }

  each thread requires two instructions to retrieve data from global memory

  | [000] | [001] | [002] | [003] | [004] | [005] | [006] | [007] | [008] | [009] | [00A] | [00B] | [00C] | [00D] | [00E] | [00F] |
  | a0   | a0   | a0   | a0   | b0   | a1   | a1   | a1   | a1   | b1   | a2   | a2   | a2   | a2   | b2   | a3   |

Enforcing Alignment

```c
__device__ struct badAlign{
    float a; //the first float in an array will be aligned
    char b;  //but this will throw off the rest
}
```

- You can force alignment with the `__align__(n)`, where n is the number of bytes to align to:

```c
__device__ struct __align__(8) badAlign{
    float a; //the first float in an array will be aligned
    char b;  //but this will throw off the rest
    ... //3 bytes of padding will be added (ex. when sizeof() is used)
}
```

- All global memory instructions coalesce into a single device memory transaction issued by the warp.
Peak Global Memory Performance

• Examine our earlier matrix multiplication kernel:

```c
__global__ void kernelMatrixMult(float* C, float* A, float* B, size_t M, size_t N){
    size_t i = blockIdx.y * blockDim.y + threadIdx.y;  //calculate the i (row) index
    size_t j = blockIdx.x * blockDim.x + threadIdx.x;  //calculate the j (column) index
    if(i >= M || j >= M) return;  //return if (i,j) is outside the matrix
    float c = 0;  //initialize a register to store the result
    for(size_t n = 0; n < N; n++)  //for each element in the dot product
        c += A[n*M+i] * B[j*N+n];  //perform a multiply-add
    C[i*M + j] = c;  //send the register value to global memory
}
```

• The primary inner loop instruction:  
  ```c
  c += A[n*M+i] * B[j*N+n];
  ```
  • performs 2 floating point operations (* and +)
  • requires two 32-bit (4 byte) global memory accesses: 8 B / 2 FLOPS = 4B/FLOP

• The maximum 32-bit floating point performance for a Tesla P100 is 5304 GFLOPS

• The maximum global memory bandwidth for a Tesla P100 is 720 GB/s

\[
720 \times 10^9 \text{B} / (4\text{B}/\text{FLOP})
= 180 \text{ GFLOPS} \approx 3.4\% \text{ of maximum performance}
\]
Other Ways to Access Device Memory

• Constant Memory
  ```
  __constant__ keyword in CUDA compiled files
  ```
  • accessible to all ```__device__``` and ```__global__``` functions
  • accessible in ```__host__``` functions through the API
    ```
    cudaMemcpyToSymbol(⋯)
    cudaMemcpyFromSymbol(⋯)
    ```

• Texture Memory
  • global memory accessed through texture units
  • weird caching method – for people who don’t work with computer graphics
  • values are cached based on 2D/3D spatial locality
    • when one thread accesses global memory, values surrounding that location are available to other threads in a warp
  • used less and less because of global memory caching in recent GPUs
Memory Declarations

```c
int cpu_global;
• defaults to __host__ if declared in global scope
  • accessible for lifetime of the application in __host__ functions

int local;
• __device__ functions
  • allocates a register
  • scope is the thread (or based on C/C++ scope rule)
  • lifetime is the duration of the thread
• __host__ functions
  • allocates a local variable with scope and lifetime as per C/C++ rules

__device__ int gpu_global;
• allocates a global variable on the device
• accessible to all device code for the duration of the application

__constant__ int gpu_const;
• allocates a constant variable in constant (read only) memory
• accessible to all device code for the duration of the application
```