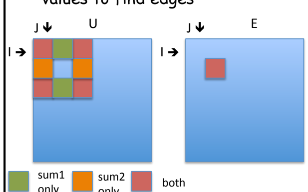
L6: Memory Hierarchy Optimization IV, Bandwidth Optimization

Administrative

- Next assignment available
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Tuesday, Feb. 9, 5PM
  - Use handin program on CADE machines
    - "handin CS6235 lab2 <probfile>"

Assignment 2: Memory Hierarchy Optimization Due Tuesday, February 7 at 5PM

Sobel edge detection:
Find the boundaries of the image where there is significant difference as compared to neighboring "pixels" and replace values to find edges




```
for (i = 1;  i < ImageNRows - 1; i++)
   for (j = 1; j < ImageNCols -1; j++)
    {
     sum1 = u[i-1][j+1] - u[i-1][j-1]
         + 2 * u[i][j+1] - 2 * u[i][j-1]
         + u[i+1][j+1] - u[i+1][j-1];
     sum2 = u[i-1][j-1] + 2 * u[i-1][j] + u[i-1][j+1]
         - u[i+1][j-1] - 2 * u[i+1][j] - u[i+1][j+1];

     magnitude =  sum1*sum1 + sum2*sum2;
     if (magnitude > THRESHOLD)
      e[i][j] = 255;
     else
      e[i][j] = 0;
    }
```


Example

Input

Output

General Approach

0. Provided
   a. Input file
   b. Sample output file
   c. CPU implementation

1. Structure
   a. Compare CPU version and GPU version output [compareInt]
   b. Time performance of two GPU versions (see 2 & 3 below) [EventRecord]

2. GPU version 1 (partial credit if correct)
   implementation using global memory

3. GPU version 2 (highest points to best performing versions)
   use memory hierarchy optimizations from previous, current and Monday's lecture

4. Extra credit: Try two different block / thread decompositions. What happens if you use more threads versus more blocks? What if you do more work per thread? Explain your choices in a README file.

Handin using the following on CADE machines, where probfile includes all files

"handin cs6235 lab2 <probfile>"

Overview

- Bandwidth optimization
  - Global memory coalescing
  - Avoiding shared memory bank conflicts
  - A few words on alignment
- Reading:
  - Chapter 4, Kirk and Hwu
  - http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf
  - Chapter 5, Kirk and Hwu
  - http://courses.ece.illinois.edu/ece498/al/textbook/Chapter5-CudaPerformance.pdf
  - Sections 3.2.4 (texture memory) and 5.1.2 (bandwidth optimizations) of NVIDIA CUDA Programming Guide

Targets of Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain

Optimizing the Memory Hierarchy on GPUs, Overview

- Device memory access times non-uniform so ***data placement*** significantly affects performance.
  - But controlling data placement may require additional copying, so consider overhead.
- Optimizations to increase memory bandwidth. Idea: maximize utility of each memory access.
  - ***Coalesce*** global memory accesses
  - ***Avoid memory bank conflicts*** to increase memory access parallelism
  - ***Align*** data structures to address boundaries
  - More minor effects
    - Partition camping to avoid global memory bank conflicts
    - Use texture accesses to increase parallelism of memory accesses (if other global accesses are occurring simultaneously) [example later in the semester]

Data Location Impacts Latency of Memory Access

- Registers
  - Can load in current instruction cycle
- Constant or Texture Memory
  - In cache? Single address can be loaded for half-warp per cycle
  - O/W, global memory access
- Global memory
- Shared memory
  - Single cycle if accesses can be done in parallel

Introduction to Memory System

- Recall execution model for a multiprocessor
  - **Scheduling unit:** A "warp" of threads is issued at a time (32 threads in current chips)
  - **Execution unit:** Each cycle, 8 "cores" or SPs are executing (32 cores in a Fermi)
  - **Memory unit:** Memory system scans a *"half warp"* or 16 threads for data to be loaded; (full warp for Fermi)

Global Memory Accesses

- Each thread issues memory accesses to data types of varying sizes, perhaps as small as 1 byte entities
- Given an address to load or store, memory returns/updates "segments" of either 32 bytes, 64 bytes or 128 bytes
- Maximizing bandwidth:
  - Operate on an ***entire*** 128 byte segment for each memory transfer

Understanding Global Memory Accesses

Memory protocol for compute capability 1.2 and 1.3* (CUDA Manual 5.1.2.1 and Appendix A.1)

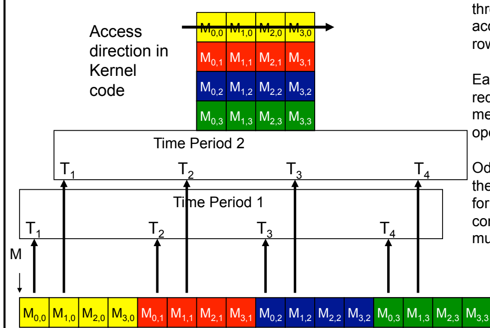
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads ***in half-warp*** are serviced

*Includes Tesla and GTX platforms as well as new Linux machines!

Protocol for most systems (including lab6 machines) even more restrictive

- For compute capability 1.0 and 1.1
  - Threads must access the words in a segment in sequence
  - The kth thread must access the kth word
  - ***Alignment to the beginning of a segment becomes a very important optimization!***

Memory Layout of a Matrix in C

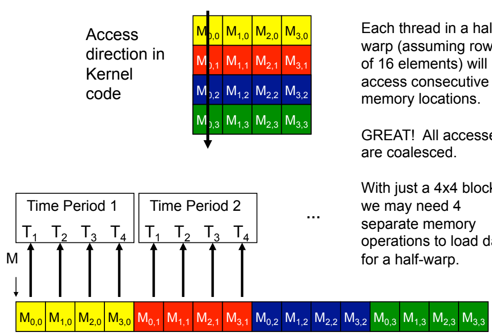


Consecutive threads will access different rows in memory.

Each thread will require a different memory operation.

Odd: But this is the RIGHT layout for a conventional multi-core!

Memory Layout of a Matrix in C



Each thread in a half-warp (assuming rows of 16 elements) will access consecutive memory locations.

GREAT! All accesses are coalesced.

With just a 4x4 block, we may need 4 separate memory operations to load data for a half-warp.

How to find out compute capability

See Appendix A.1 in NVIDIA CUDA Programming Guide to look up your device.

Also, recall “deviceQuery” in SDK to learn about features of installed device.

Linux lab, most CADE machines and Tesla cluster are Compute Capability 1.2 and 1.3.

Fermi machines are 2.x.

Alignment

- Addresses accessed within a half-warp may need to be ***aligned*** to the beginning of a segment to enable coalescing
  - An aligned memory address is a multiple of the memory segment size
  - In compute 1.0 and 1.1 devices, address accessed by lowest numbered thread must be aligned to beginning of segment for coalescing
  - In future systems, sometimes alignment can reduce number of accesses

More on Alignment

- Objects allocated statically or by cudaMalloc begin at aligned addresses
  - But still need to think about index expressions
- May want to align structures

```
struct __align__(8) {
  float a;
  float b;
};
```

```
struct __align__(16) {
  float a;
  float b;
  float c;
};
```


What Can You Do to Improve Bandwidth to Global Memory?

- Think about spatial reuse and access patterns across threads
  - May need a different computation & data partitioning
  - May want to rearrange data in shared memory, even if no temporal reuse (transpose example)
  - Similar issues, but much better in future hardware generations

Bandwidth to Shared Memory: Parallel Memory Accesses

- Consider each thread accessing a different location in shared memory
- Bandwidth maximized if each one is able to proceed ***in parallel***
- Hardware to support this
  - ***Banked memory:*** each bank can support an access on every memory cycle

How addresses map to banks on G80

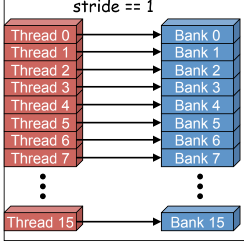
- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

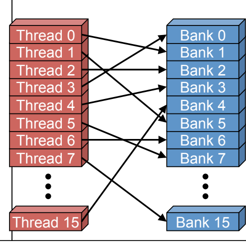
- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank

Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

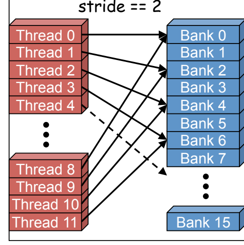


- No Bank Conflicts
  - Random 1:1 Permutation

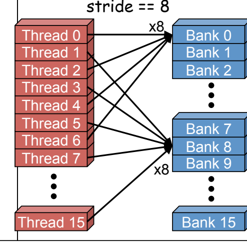


Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2



- 8-way Bank Conflicts
  - Linear addressing stride == 8



Putting It Together: Global Memory Coalescing and Bank Conflicts

- Let's look at matrix transpose
- Simple goal: Replace A[i][j] with A[j][i]
- Any reuse of data?
- Do you think shared memory might be useful?

Matrix Transpose (from SDK)

```
_global__ void transpose(float *odata, float *idata, int width, int height)
{
    // read the element
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    unsigned int index_in = yIndex * width + xIndex;
    temp = idata[index_in];

    // write the transposed element to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    unsigned int index_out = yIndex * height + xIndex;
    odata[index_out] = temp;
}
```

odata and idata in global memory

Coalesced Matrix Transpose

```
_global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_D
    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    unsigned int index_in = yIndex * width + xIndex;
    block[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    unsigned int index_out = yIndex * height + xIndex;
    odata[index_out] = block[threadIdx.x][threadIdx.y];
}
```

odata and idata in global memory

Rearrange in shared memory and write back efficiently to global memory

Optimized Matrix Transpose (from SDK)

```
_global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM][
    // read the matrix tile into shared memory
    unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
    unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
    unsigned int index_in = yIndex * width + xIndex;
    block[threadIdx.y][threadIdx.x] = idata[index_in];

    __syncthreads();

    // write the transposed matrix tile to global memory
    xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
    yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
    unsigned int index_out = yIndex * height + xIndex;
    odata[index_out] = block[threadIdx.x][threadIdx.y];
}
```

odata and idata in global memory

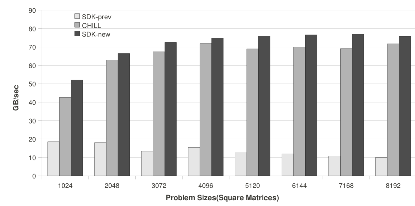
Rearrange in shared memory and write back efficiently to global memory

Further Optimization: Partition Camping

- A further optimization improves bank conflicts in global memory
  - But has not proven that useful in codes with additional computation
- Map blocks to different parts of chips

```
int bid = blockIdx.x + gridDim.x*blockIdx.y;
by = bid%gridDim.y;
bx = ((bid/gridDim.y)+by)%gridDim.x;
```


Performance Results for Matrix Transpose (GTX280)



SDK-prev: all optimizations other than partition camping
CHiLL: generated by our compiler
SDK-new: includes partition camping

Summary of Lecture

- Completion of bandwidth optimizations
  - Global memory coalescing
  - Alignment
  - Shared memory bank conflicts
  - "Partitioning camping"
- Matrix transpose example

Next Time

- A look at correctness
- Synchronization mechanisms