L6: Memory Hierarchy Optimization IV, Bandwidth Optimization

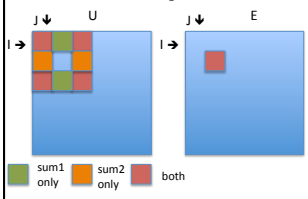
Administrative

- Next assignment available
  - Next three slides
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Friday, Feb. 8, 5PM
  - Use handin program on CADE machines
    - "handin CS6235 lab2 <probfile>"

Assignment 2: Memory Hierarchy Optimization Due Fri day, February 8 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



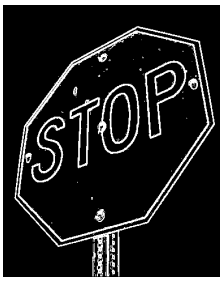

```
for (i = 1;  i < ImageNRows - 1; i++)
  for (j = 1; j < ImageNCols -1; j++)
   {
    sum1 = u[i-1][j+1] - u[i-1][j-1]
        + 2 * u[i][j+1] - 2 * u[i][j-1]
        + u[i+1][j+1] - u[i+1][j-1];
    sum2 = u[i-1][j-1] + 2 * u[i-1][j] + u[i-1][j+1]
        - u[i+1][j-1] - 2 * u[i+1][j] - u[i+1][j+1];

    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 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
  - Chapter 5, Kirk and Hwu
  - Sections 3.2.4 (texture memory) and 5.1.2 (bandwidth optimizations) of NVIDIA CUDA Programming Guide

Overview of Texture Memory

- Recall, texture cache of read-only data
- Special protocol for allocating and copying to GPU
  - texture<Type, Dim, ReadMode> texRef;
    - Dim: 1, 2 or 3D objects
- Special protocol for accesses (macros)
  - tex2D(<name>,dim1,dim2);
- In full glory can also apply functions to textures
- Writing possible, but unsafe if followed by read in same kernel

Using Texture Memory (simpleTexture project from SDK)

```
cudaMalloc( (void**) &d_data, size);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
    cudaChannelFormatKindFloat);
cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height );
cudaMemcpyToArray( cu_array, 0, 0, h_data, size, cudaMemcpyHostToDevice);
// set texture parameters
tex.addressMode[0] = tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear; tex.normalized = true;
cudaBindTextureToArray( tex,cu_array, channelDesc);
// execute the kernel
transformKernel<<< dimGrid, dimBlock, 0 >>>( d_data, width, height, angle);
```

Kernel function:

```
// declare texture reference for 2D float texture
texture<float, 2, cudaReadModeElementType> tex;

... = tex2D(tex,i,j);
```

When to use Texture (and Surface) Memory

(From 5.3 of CUDA manual) Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- If memory reads to global or constant memory will not be coalesced, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads (this is less likely for devices of compute capability 2.x given that global memory reads are cached on these devices);
- Addressing calculations are performed outside the kernel by dedicated units;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] (see Section 3.2.4.1.1).

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)

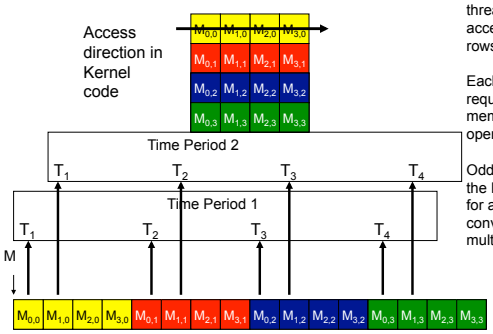
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory 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

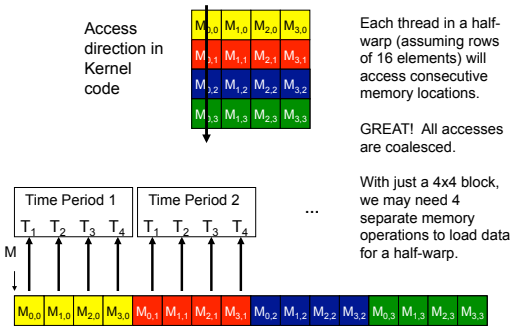


Consecutive threads will access different rows in memory.

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

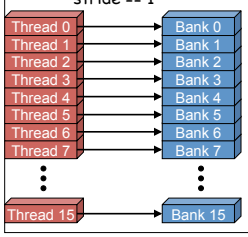
- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- G80 has 16 banks
  - So bank = address % 16
  - Same as the size of a half-warp
    - No bank conflicts between different half-warps, only within a single half-warp

Shared memory bank conflicts

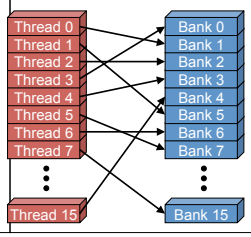
- Shared memory is as fast as registers if there are no bank conflicts
- The fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access the identical address, there is no bank conflict (broadcast)
- The slow case:
  - Bank Conflict: 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

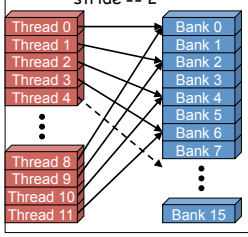


- No Bank Conflicts
  - Random 1:1 Permutation

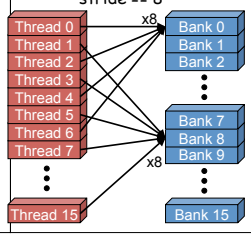


Bank Addressing Examples

- 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)
{
      int xIndex = blockIdx.x * TILE_DIM + thr
      int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
      int index_in  = xIndex + width * yIndex;
      int index_out = yIndex + height * xIndex;

   for (int r=0; r < nreps; r++)
   {
      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS)
      {
         // read the element
         // write the transposed element to global memory
         odata[index_out+i] = idata[index_in+i*width];
      }
   }
```

odata and idata in global memory

Coalesced Matrix Transpose

```
_global__ void transpose(float *odata, float *idata, int width, int height)
{
   __shared__ float tile[TILE_DIM][TILE_DIM

   // read the matrix tile into shared memory
   int xIndex = blockIdx.x * TILE_DIM + threadI
   int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
   int index_in = xIndex + (yIndex)*width;

   xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
   yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
   int index_out = xIndex + (yIndex)*height;

   for (int r=0; r < nreps; r++)  {
      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
         tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
      }
      __syncthreads();
      // write the transposed matrix tile to global memory
      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
         odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
      }
   }
}
```

odata and idata in global memory

Rearrange in shared memory and write back efficiently to global memory

Coalesced Matrix Transpose – No Shared Memory Bank Conflicts

```
_global__ void transpose(float *odata, float *idata, int width, int height)
{
   __shared__ float tile[TILE_DIM][TILE_DIM

   // read the matrix tile into shared memory
   int xIndex = blockIdx.x * TILE_DIM + threadI
   int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
   int index_in = xIndex + (yIndex)*width;

   xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
   yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
   int index_out = xIndex + (yIndex)*height;

   for (int r=0; r < nreps; r++)  {
      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
         tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
      }
      __syncthreads();
      // write the transposed matrix tile to global memory
      for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
         odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
      }
   }
}
```

odata and idata in global memory

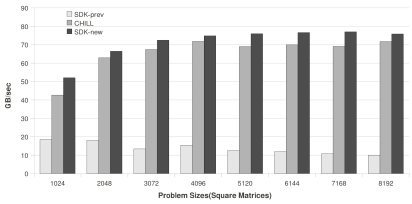
Rearrange in shared memory and write back efficiently to global memory

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