L6: Memory Hierarchy Optimization III, Bandwidth Optimization

Administrative

- Next assignment on the website
  - Description at end of class
  - Due Wednesday, Feb. 17, 5PM
  - Use handin program on CADE machines
    - "handin cs6963 lab2 <probfile>"
- Mailing lists
  - cs6963s10-discussion@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
  - cs6963s10-teach@list.eng.utah.edu
    - Please use for questions to Protonu and me

Administrative, cont.

- New Linux Grad Lab coming on-line!
  - 6 machines up and running
  - All machines have the GTX260 graphics cards, Intel Core i7 CPU 920 (quad-core 2.67GHz) and 6Gb of 1600MHz (DDR) RAM.
- Need CUDA installed

Overview

- Complete discussion of data placement in registers and texture memory
- Introduction to memory system
- 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

Data Placement: Conceptual

- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use SP shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only "reused" data can be placed in constant & texture memory by host
- Also, how to use registers
  - Most locally-allocated data is placed directly in registers
  - Even array variables can use registers if compiler understands access patterns
  - Can allocate "superwords" to registers, e.g., float4
  - Excessive use of registers will "spill" data to local memory
- Local memory
  - Deals with capacity limitations of registers and shared memory
  - Eliminates worries about race conditions
  - … but SLOW

Tiling Example

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
     D[i] = D[i] + B[j][i];
```

Strip mine

```
for (j=1; j<M; j++)
   for (ii=1; ii<N; ii+=s)
       for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j][i];
```

Permute

```
for (ii=1; ii<N; ii+=s)
  for (j=1; j<M; j++)
     for (i=ii; i<min(ii+s-1,N); i++)
         D[i] = D[i] +B[j][i];
```

Legality of Tiling

- Tiling = strip-mine and permutation
  - Strip-mine does not reorder iterations
  - Permutation must be legal

  OR
  - strip size less than dependence distance

A Few Words On Tiling

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity

"Tiling" for Registers

- A similar technique can be used to map data to registers
- Unroll-and-jam
  - Unroll outer loops in a nest and fuse together resulting inner loops
- Jamming safe if dependences are not reversed
- Scalar replacement
  - May be followed by replacing array references with scalar variables to help compiler identify register opportunities

Unroll-and-Jam Example

```
for (j=1; j<M; j++)
   for (i=1; i<N; i++)
     D[i] = D[i] + B[j][i];
```

Unroll Outer Loop

```
for (j=1; j<M; j+=2)
   for (i=1; i<N; i++)
     D[i] = D[i] + B[j][i];
   for (i=1; i<N; i++)
     D[i] = D[i] + B[j+1][i];
```

"Jam" copies of inner loop

```
for (j=1; j<M; j+=2)
   for (i=1; i<N; i++)
     D[i] = D[i] + B[j][i];
     D[i] = D[i] + B[j+1][i];
```

Scalar Replacement Example

Result of Unroll-and-jam

```
for (j=1; j<M; j+=2)
   for (i=1; i<N; i++) {
     D[i] = D[i] + B[j][i];
     D[i] = D[i] + B[j+1][i];
   }
```

Scalar replacement for D[i]

```
for (j=1; j<M; j+=2)
   for (i=1; i<N; i++){
     t0 = D[i];
     t0 = t0 + B[j][i];
     t0 = t0 + B[j+1][i];
     D[i] = t0;
   }
```


Legality of Unroll-and-Jam

- Unrolling is always safe
  - If you take care not to go past end of iterations
- Jamming inner loops back together
  - This optimization is safe as long as no dependences are reversed

More Details

- Similar to tiling, but reuse must be explicit in code
- Interaction with the register allocator
  - Historically, array variables were not placed in registers due to concerns about data dependences
  - Nvcc capable of placing small array variables in registers if subscripts are constant and for some simple subscripts
  - Can tell from compiler output whether data is in a register (more later)

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

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);
```


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
  - ***Memory unit**: Memory system scans a "half warp" or 16 threads for data to be loaded*

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 (next)
- Shared memory
  - Single cycle if accesses can be done in parallel

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)

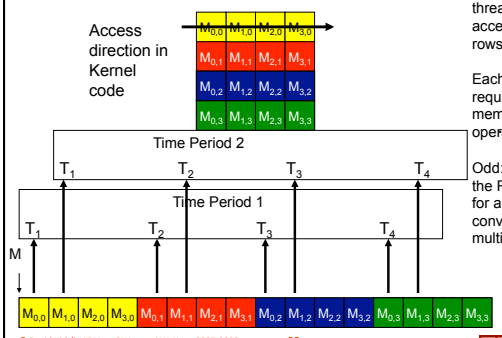
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory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

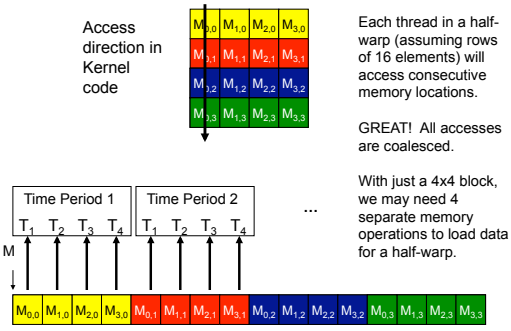


Consecutive threads will access different rows in memory.

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.

CADE machines are all Compute Capability 1.0 or 1.1!

New Linux lab, and Tesla cluster are Compute Capability 1.2 and 1.3.

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 from L4)
  - Similar issues, but much better in future hardware generations

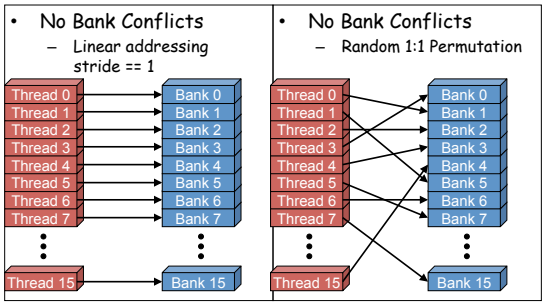
Bandwidth to Shared Memory: Parallel Memory Accesses

- 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

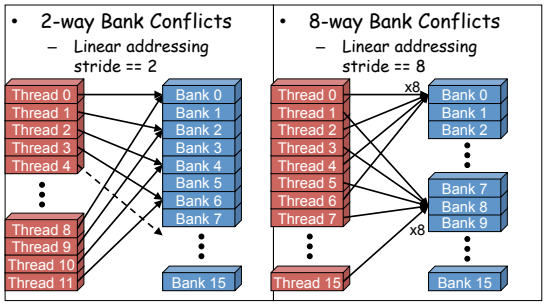
How addresses map to banks on G80

- 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

Bank Addressing Examples



Bank Addressing Examples



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

Summary of Lecture

- Reordering transformations to improve locality
  - Tiling, permutation and unroll-and-jam
- Guiding data to be placed in registers
- Placing data in texture memory
- Introduction to global memory bandwidth

Next Time

- Real examples with measurements
- cudaProfiler and output from compiler
  - How to tell if your optimizations are working