L7: Memory Hierarchy Optimization IV, Bandwidth Optimization and Case Studies

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
  – cs6963s10-teach@list.eng.utah.edu
    • Please use for questions to Proton and me

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-
    cuDadeMemoryModel.pdf
  – Chapter 5, Kirk and Hwu
  – http://courses.ece.illinois.edu/ece498/al/textbook/Chapter5-
    cuDadePerformance.pdf
  – Sections 3.2.4 (texture memory) and 5.1.2 (bandwidth optimizations) of NVIDIA CUDA Programming Guide

Administrative, cont.

• New Linux Grad Lab 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.
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.
  - Coalesce global memory accesses
  - Avoid memory bank conflicts to increase memory access parallelism
  - Align data structures to address boundaries

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
  - Random 1:1 Permutation

Linear Addressing

- Given:
  ```
  __shared__ float shared[256];
  float foo = shared[baseIndex + s * threadIdx.x];
  ```

- This is only bank-conflict-free if \( s \) shares no common factors with the number of banks
  - 16 on G80, so \( s \) must be odd
**Data types and bank conflicts**

- This has no conflicts if type of `shared` is 32-bits:
  ```
  foo = shared[baseIndex + threadIdx.x]
  ```

- But not if the data type is smaller
  - 4-way bank conflicts:
    ```
    _shared__ char shared[];
    foo = shared[baseIndex + threadIdx.x]
    ```
  - 2-way bank conflicts:
    ```
    _shared__ short shared[];
    foo = shared[baseIndex + threadIdx.x]
    ```

**Structs and Bank Conflicts**

- Struct assignments compile into as many memory accesses as there are struct members:
  ```
  struct vector { float x, y, z; };
  struct myType {
    float f;
    int c;
  };
  
  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for vector; struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)
  ```
  struct vector v = vectors[baseIndex + threadIdx.x];
  ```

- This has 2-way bank conflicts for myType; (2 accesses per thread)
  ```
  struct myType m = myTypes[baseIndex + threadIdx.x];
  ```

**Common Bank Conflict Patterns, 1D Array**

- Each thread loads 2 elements into shared mem:
  - 2-way-interleaved loads result in 2-way bank conflicts:
    ```
    int tid = threadIdx.x;
    shared[2*tid] = global[2*tid];
    shared[2*tid+1] = global[2*tid+1];
    ```
  - This makes sense for traditional CPU threads, exploits spatial locality in cache line and reduces sharing traffic

**A Better Array Access Pattern**

- Each thread loads one element in every consecutive group of blockDim elements.
  ```
  shared[tid] = global[tid];
  shared[tid + blockDim.x] = global[tid + blockDim.x];
  ```
What Can You Do to Improve Bandwidth to Shared Memory?

- Think about memory access patterns across threads
  - May need a different computation & data partitioning
  - Sometimes "padding" can be used on a dimension to align accesses

A Running Example: 2-D Jacobi Relaxation

- A "stencil" computation
  - Output for a point depends on neighboring points from input
  - A common pattern in scientific computing and image/signal processing (Sobel)

for (int i; i<n; i++)
for (int j; j<n; j++)
b[i][j] = 0.5*(a[i-1][j] + a[i+1][j] + a[i][j+1] + a[i][j-1]);

How to Map Jacobi to GPU (Tiling)

for (i=1; i<n; i++)
for (j=1; j<n; j++)
b[i][j] = 0.5*(a[i-1][j] + a[i+1][j] + a[i][j+1] + a[i][j-1]);

TILED SEQUENTIAL CODE

// For clarity, assume n is evenly divisible by TX and TY
for (i=0; i<n/TX; i++) // MAP TO blockIdx.x
for (j=0; j<n/TY; j++) // MAP TO threadIdx.y
for (x=0; x<TX; x++) // MAP TO threadIdx.x
for (y=0; y<NY; y++) // MAP TO threadIdx.y
b[(TX*x+1)/TY*y] = 0.5*(a[(TX*x+2)/TY*y] + a[(TX*x+1)/TY*y+1] + a[(TX*x+1)/TY*y+2] + a[(TX*x+1)/TY*y+2]);

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
Automatically Generated Code

// GPU Kernel Code
extern __global__ void Jacobi_GPU(float *b, float *a) {
    int t2;
    int t4;
    int t6;
    int t10;
    t2 = blockIdx.x;
    t4 = blockIdx.y;
    t6 = threadIdx.x;
    t8 = threadIdx.y;
    // make sure we don't go off end
    b[(TX*t2+t6)*TY+t8] = 0.5*(
        a[(TX*t2+t6+2)*TY+t8+1] +
        a[(TX*t2+t6)*TY+t8+1] +
        a[(TX*t2+t6+1)*TY*t8+2] +
        a[(TX*t2+t6+1)*TY*t8]);
}

Slightly Different Code – Using Texture Memory

texture<float, 1, cudaReadModeElementType> texRef;

// GPU Kernel Code
__global__ void jacobi_GPU(float *a[], float* b) {
    int thidx = SBX * blockIdx.y;
    int thidy = threadIdx.x + SBY * blockIdx.x;
    if(thidy > 0 && thidy < (N-1) )
        for(int j = 0 ; j < SBX ; j++)
            if (thidx > 0 && thidx < (N-1) )
                b[(thidx-1)*(N-2) + (thidy-1)] = 0.5* ( tex1Dfetch(texRef,(thidx+1)*N + thidy) +
                    tex1Dfetch(texRef,(thidx-1)*N + thidy) +
                    tex1Dfetch(texRef,thidx*N + (thidy+1)) +
                    tex1Dfetch(texRef,(thidx)*N + (thidy-1)));
            thidx++;
    //end for
}
Matrix Transpose (from SDK)

```c
__global__ void transpose(float *odata, float *idata, int width, int height)
{
    __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];
    // 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];
}
```

How to Get Compiler Feedback

```
nvcc --ptxas-options=-v -I/Developer/CUDA/common/inc -L/Developer/CUDA/lib mmul.cu -lcutil
```

Returns:

- `ptxas info`: Compiling entry function
  `__globfunc__Z12mmul_computePfS_S_i` used 9 registers, 2080+1056 bytes smem, 8 bytes cmem

From 2-D Jacobi Example

- Use of tiling just for computation partitioning to GPU
- Factor of 2 difference due to coalescing, even for identical layout and just differences in partitioning
- Texture memory improves performance
CUDA Profiler

- What it does:
  - Provide access to hardware performance monitors
  - Pinpoint performance issues and compare across implementations
- Two interfaces:
  - Text-based:
    - Built-in and included with compiler
  - GUI:

Example

- Reverse array from Dr. Dobb’s journal
  - http://www.ddj.com/architect/207306659 (Part 6)
- Reverse_global
  - Copy from global to shared, then back to global in reverse order
- Reverse_shared
  - Copy from global to reverse shared and rewrite in order to global
- Output

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