L4: Memory Hierarchy Optimization I, Locality and Data Placement

Overview
- Where data can be stored
  - And how to get it there
- Some guidelines for where to store data
  - Who needs to access it?
  - Read only vs. Read/Write
  - Footprint of data
- High level description of how to write code to optimize for memory hierarchy
  - More details Wednesday and (probably) next week
- Reading:
  - Chapter 4, Kirk and Hwu
  - http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

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

CS6963

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

CS6963
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.
- **Coalesce** global memory accesses
- **Avoid memory bank conflicts** to increase memory access parallelism
- **Align** data structures to address boundaries

Hardware Implementation: Memory Architecture

- The local, global, constant, and texture spaces are regions of device memory (DRAM)
- Each multiprocessor has:
  - A set of 32-bit registers per processor
  - On-chip shared memory
    - Where the shared memory space resides
    - A read-only constant cache
      - To speed up access to the constant memory space
    - A read-only texture cache
      - To speed up access to the texture memory space

Programmer’s View: Memory Spaces

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can read/write global, constant, and texture memory

Terminology Review

- **device** = GPU = set of multiprocessors
- **Multiprocessor** = set of processors & shared memory
- **Kernel** = GPU program
- **Grid** = array of thread blocks that execute a kernel
- **Thread block** = group of SIMD threads that execute a kernel and can communicate via shared memory
Reuse and Locality

- Consider how data is accessed
  - **Data reuse:**
    - Same data used multiple times
    - Intrinsic in computation
  - **Data locality:**
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Access Times

- Register - dedicated HW - single cycle
- Constant and Texture caches - possibly single cycle, proportional to addresses accessed by warp
- Shared Memory - dedicated HW - single cycle if no "bank conflicts"
- Local Memory - DRAM, no cache - *slow*
- Global Memory - DRAM, no cache - *slow*
- Constant Memory - DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- Texture Memory - DRAM, cached, 1...10s...100s of cycles, depending on cache locality
- Instruction Memory (invisible) - DRAM, cached

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

Data Placement: Syntax

- Through type qualifiers
  - __constant__, __shared__, __local__, __device__
- Through cudaMemcpy calls
  - Flavor of call and symbolic constant designate where to copy
- Implicit default behavior
  - Device memory without qualifier is global memory
  - Host by default copies to global memory
  - Thread-local variables go into registers unless capacity exceeded, then local memory
### Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Memory Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>local</td>
</tr>
<tr>
<td><strong>local</strong></td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong></td>
<td>block</td>
</tr>
<tr>
<td><strong>block</strong></td>
<td>block</td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td>grid</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>application</td>
</tr>
</tbody>
</table>

- __local__ is optional when used with __shared__, or __constant__.
- Automatic variables without any qualifier reside in a register.
  - Except arrays that reside in local memory.

### Variable Type Restrictions

- **Pointers** can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    ```
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable:
    ```
    float* ptr = &GlobalVar;
    ```

### Rest of Today's Lecture

- Mechanics of how to place data in shared memory and constant memory.
- Tiling transformation to reuse data within.
  - Shared memory
  - Constant cache

### Constant Memory Example

- **Signal recognition**:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute \( M_1 V, M_2 V, \ldots, M_n V \)

```c
__constant__ float d_signalVector[M];
__device__ float R[N][M];

__host__ void outerApplySignal() {  
  float h_inputSignal;
  dim3 dimGrid(N);
  dim3 dimBlock(M);
  cudaMemcpyToSymbol(d_signalVector, h_inputSignal, M*sizeof(float));  
  // input matrix is in d_mat
  ApplySignal<<<dimGrid, dimBlock>>>(d_mat, M);
}
__global__ void ApplySignal(float* d_mat, int M) {  
  float result = 0.0; /* register */
  for (j = 0; j < M; j++)  
    result += d_mat[blockIdx.x][j] * d_signalVector[j];  
  R[blockIdx.x][threadIdx.x] = result;
}
```
More on Constant Cache

- Example from previous slide
  - All threads in a block accessing same element of signal vector
  - Brought into cache for first access, then latency equivalent to a register access

![Diagram of register file]

Additional Detail

- Suppose each thread accesses different data from constant memory on same instruction
  - Reuse across threads?
    - Consider capacity of constant cache and locality
    - Code transformation needed? (later in lecture)
  - Cache latency proportional to number of accesses in a warp
  - No reuse?
    - Should not be in constant memory.

Now Let's Look at Shared Memory

- Common Programming Pattern (5.1.2 of CUDA manual)
  - Load data into shared memory
  - Synchronize (if necessary)
  - Operate on data in shared memory
  - Synchronize (if necessary)
  - Write intermediate results to global memory
  - Repeat until done

Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```c
extern __shared__ float d_s_array[];
__global__ void compute()
{
  d_s_array[i] = ...;
}
```

```c
__global__ void compute2()
{
  __shared__ float d_s_array[M];
  d_s_array[j] = ...;
  __global__ void compute()
  {
    d_s_array[] = ...;
  }
```
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];
}
```

Reuse and Locality

- Consider how data is accessed
  - Data reuse:
    - Same data used multiple times
    - Intrinsic in computation
  - Data locality:
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```c
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

- A[j] has self-temporal reuse in loop i

Spatial Reuse (Ignore for now)

- Same data transfer (usually cache line) used in distinct iterations I and I'

```c
for (i=1; i<N; i++)
    for (j=1; j<N; j++)
```

- A[j] has self-spatial reuse in loop j
- Multi-dimensional array note: C arrays are stored in row-major order
Group Reuse

- Same data used by distinct references

```c
for (i=1; i<N; i++)
for (j=1; j<N; j++)
```

- \(A[j], A[j+1]\) and \(A[j-1]\) have group reuse (spatial and temporal) in loop \(j\)

Can Use Reordering Transformations!

- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - Safety? (doesn’t reverse dependences)
  - Profitability? (improves locality)

Safety of Permutation

- Intuition: Cannot permute two loops \(i\) and \(j\) in a loop nest if doing so reverses the direction of any dependence.

```c
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```

Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

```c
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```

Which one is better for row-major storage?

```c
for (i=0; i<3; i++)
for (j=0; j<6; j++)
```
Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time

Example:

```c
for (j=1; j<M; j++)
    for (i=i; i<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
Matrix Multiplication
A Simple Host Version in C

```c
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j)
            for (int k = 0; k < Width; ++k)
            {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                double sum = a * b;
                P[i * Width + j] = sum;
            }
}
```

Tiled Matrix Multiply Using Thread Blocks

- One block computes one square sub-matrix \( P_{bb} \) of size \( BLOCK_SIZE \)
- One thread computes one element of \( P_{bb} \)
- Assume that the dimensions of \( M \) and \( N \) are multiples of \( BLOCK_SIZE \) and square shape

Shared Memory Usage

- Assume each SMP has 16KB shared memory
  - Each Thread Block uses \( 2^2 \times 256^2 \times 4B = 2KB \) of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
  - For \( BLOCK_SIZE = 16 \), this allows up to \( 8 \times 512 = 4,096 \) pending loads
    - In practice, there will probably be up to half of this due to scheduling to make use of SPSes
  - The next \( BLOCK_SIZE = 32 \) would lead to \( 2^2 \times 32^2 \times 4B = 8KB \) shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time.

First-order Size Considerations

- Each Thread Block should have a minimal of 192 threads
  - \( BLOCK_SIZE \) of 16 gives \( 16 \times 16 = 256 \) threads
- A minimal of 32 Thread Blocks
  - A \( 1024 \times 1024 \) \( P \) Matrix gives \( 64 \times 64 = 4096 \) Thread Blocks
- Each thread block performs \( 2^2 \times 256 = 512 \) float loads from global memory for \( 256 \times (2^2 \times 16) = 8,192 \) mul/add operations
  - Memory bandwidth no longer a limiting factor
CUDA Code – Kernel Execution Configuration

// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(M.width / dimBlock.x,
               M.height / dimBlock.y);

For very large N and M dimensions, one will need to add another level of blocking and execute the second-level blocks sequentially.

CUDA Code – Kernel Overview

// Block index
int bx = blockIdx.x;
int by = blockIdx.y;
// Thread index
int tx = threadIdx.x;
int ty = threadIdx.y;

// Pvalue stores the element of the block sub-matrix
// that is computed by the thread
float Pvalue = 0;

// Loop over all the sub-matrices of M and N
// required to compute the block sub-matrix
for (int m = 0; m < M.width/BLOCK_SIZE; ++m) {
    // code from the next few slides
};
CUDA Code - Save Result

// Get a pointer to the block sub-matrix of P
Matrix Psub = GetSubMatrix(P, bx, by);

// Write the block sub-matrix to device memory;
// each thread writes one element
SetMatrixElement(Psub, tx, ty, Pvalue);

This code should run at about 150 Gflops on a GTX or Tesla.
State-of-the-art mapping (in CUBLAS 2.0) yields just under 400 Gflops.

Assignment 2:
Memory Hierarchy Optimization

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 j = 1; j < ImageNRows - 1; j++
for i = 1; i < ImageNCols - 1; i++
{
    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;
}
General Approach

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

1. Structure
   a. Compare GPU version and GPU version output (compareInt from L3, slide 30)
   b. Time performance of two GPU versions (see 2 & 3 below) (see timing construct from L2, p. 14)

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 this and next 2 lectures

Handin using the following on CADE machines, where probfile includes all files
   "handin cs6963 lab2 <probfile>"

Summary of Lecture

- How to place data in constant memory and shared memory
- Reordering transformations to improve locality
- Tiling transformation
- Matrix multiply example

Next Time

- Complete this example
  - Also, registers and texture memory
- Reasoning about reuse and locality
- Introduction to bandwidth optimization