Assignment: Signal Recognition

- **Definition:**
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Examine result to determine which of a collection of transform matrices is closest to signal
  - Compute $M_1V$, $M_2V$, ..., $M_nV$

```c
ApplySignal(float* mat, float* signal, int M) {
    float result = 0.0; /* register */
    for (i=0; i<M; i++) {
        for (j=0; j<M; j++) {
            result[i] += mat[i][j] * signal[j];
        }
    }
}
```

Requirements:
- Use global memory, registers and shared memory only (no constant memory)
- Explore different ways of laying out data
- Explore different numbers of blocks and threads
- Be careful that formulation is correct

Overview of Lecture

- Complete scheduling example from last time
- 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 next week
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, [http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf](http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf)

Administrative

- Next assignment coming on Wednesday
  - Preview (next slide)
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoffs
  - Due Friday, Feb. 4, 5PM
  - Use handin program on CADE machines
    - `handin cs6963 lab2 profilename`
- Mailing lists
  - `cs6963s11-discussion@list.eng.utah.edu`
  - Please use for all questions suitable for the whole class
  - Feel free to answer your classmates questions!
  - `cs6963s11-teach@list.eng.utah.edu`
  - Please use for questions to Sriram and me
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
  - 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
    - If one global memory access is needed for every 4 instructions
    - A minimum of 13 Warps are needed to fully tolerate 200-cycle memory latency

Scoreboarding

- How to determine if a thread is ready to execute?
- A scoreboard is a table in hardware that tracks
  - instructions being fetched, issued, executed
  - resources (functional units and operands) they need
  - which instructions modify which registers
- Old concept from CDC 6600 (1960s) to separate memory and computation

SM Instruction Buffer - Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction cache
  - into any instruction buffer slot
- Issue one "ready-to-go" warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboard used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3, and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>warp1</td>
<td>42</td>
<td>Computing</td>
</tr>
<tr>
<td>warp3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>warp8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
</tbody>
</table>

Details of Mapping

- If #blocks in a grid exceeds number of SMs,
  - multiple blocks mapped to an SM
  - treated independently
  - provides more warps to scheduler so good as long as resources not exceeded
  - Possibly stalls when scheduling across blocks (registers and shared memory cannot support multiple blocks)

Transparent Scalability

- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors

Each block can execute in any order relative to other blocks.
Switching Gears: 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

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
  - Data cache (Fermi only)

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

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>No</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>
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><strong>device</strong></th>
<th><strong>local</strong></th>
<th><strong>shared</strong></th>
<th><strong>constant</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>thread</td>
<td>block</td>
<td>grid</td>
</tr>
<tr>
<td>thread</td>
<td>block</td>
<td></td>
<td>application</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__.

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
  - Data cache (Fermi only)

Constant Memory Example

- Signal recognition example:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute $M_1V, M_2V, \ldots, M_nV$
    ```
    __constant__ float d_signalVector[M];
    __device__ float R[N][M];
    __host__ void outerApplySignal() {
        __global__ void applySignal(float* d_mat, int M) {
            float result = 0.0; /* register */
            for (j=0; j<M; j++)
                result += d_mat[blockIdx.x][threadIdx.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

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:
  ```
  extern __shared__ float d_s_array[M];
  // create or copy from global memory
  d_s_array[i] = ...;
  // synchronize threads before use
  __syncthreads();
  // now can use any element
  // more synchronization needed if updated
  // may write result back to global memory
  d_g_array[j] = d_s_array[i];
  ```
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'

\[
\begin{align*}
\text{for } (i=1; i<N; i++) & \\
\text{for } (j=1; j<N; j++) & \\
\end{align*}
\]

- 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'

\[
\begin{align*}
\text{for } (i=1; i<N; i++) & \\
\text{for } (j=1; j<N; j++) & \\
\end{align*}
\]

- 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

\[
\begin{align*}
\text{for } (i=1; i<N; i++) & \\
\text{for } (j=1; j<N; j++) & \\
\end{align*}
\]

- A[j], A[j+1] and A[j-1] have group reuse (spatial and temporal) in loop j
Tiling (Blocking): Another Loop Reordering Transformation

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

### Tiling Example

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

#### Strip mine

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

#### Permute

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

### Legality of Tiling

- Tiling is safe only if it does not change the order in which memory locations are read/written
  - We’ll talk about correctness after memory hierarchies
- Tiling can conceptually be used to perform the decomposition into threads and blocks
  - We’ll show this later, too

### 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

---

1/24/11
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) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

CUDA Code – Kernel Configuration

```c
// 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

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

float Pvalue = 0;

for (int m = 0; m < M.width/BLOCK_SIZE; ++m) {
    code from the next few slides;
}
```
CUDA Code - Load Data to Shared Memory

// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// Get a pointer to the current sub-matrix Nsub of N
Matrix Nsub = GetSubMatrix(N, bx, m);

__shared__ float Ms[BLOCK_SIZE][BLOCK_SIZE];
__shared__ float Ns[BLOCK_SIZE][BLOCK_SIZE];

// each thread loads one element of the sub-matrix
Ms[ty][tx] = GetMatrixElement(Msub, tx, ty);

Ns[ty][tx] = GetMatrixElement(Nsub, tx, ty);

CUDA Code - Compute Result

// Synchronize to make sure the sub-matrices are loaded
// before starting the computation
__syncthreads();

// each thread computes one element of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
Pvalue += Ms[ty][k] * Ns[k][tx];

// Synchronize to make sure that the preceding
// computation is done before loading two new
// sub-matrices of M and N in the next iteration
__syncthreads();

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 3.2 on C2050) yields just above 600 Gflops. Higher on GTX480.

Matrix Multiply in CUDA

- Imagine you want to compute extremely large matrices.
  - That don’t fit in global memory
- This is where an additional level of tiling could be used, between grids
Summary of Lecture

• How to place data in constant memory and shared memory
• Introduction to Tiling transformation
• Matrix multiply example

Next Time

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