

L3: Memory Hierarchy Optimization I, Locality and Data Placement

CS6963

L3: Memory Hierarchy, 1

1



Administrative

- Next assignment coming on Wednesday
 - Preview (next slide)
 - Goals of assignment:
 - simple memory hierarchy management
 - block-thread decomposition tradeoff
 - Due Friday, Feb. 4, 5PM
 - Use handin program on CADE machines
 - "handin cs6963 lab2 <probfile>"
- 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!
 - cs6963s1-teach@list.eng.utah.edu
 - Please use for questions to Sriram and me

CS6963

L3: Memory Hierarchy, 1



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, \dots, M_nV

```
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

CS6963

L3: Memory Hierarchy, 1



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>

CS6963

L3: Memory Hierarchy, 1



SM Warp Scheduling

The diagram shows a 'SM multithreaded Warp scheduler' at the top. Below it, a vertical timeline labeled 'time' shows the execution of instructions from different warps. The instructions are: warp 8 instruction 17, warp 1 instruction 42, warp 3 instruction 95, warp 8 instruction 12, and warp 3 instruction 96. Arrows indicate that all threads in a warp execute the same instruction when selected.

- 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
From L2: Hardware Overview

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 **scoreboarding** used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp

The block diagram shows the flow of an instruction: IS (Instruction Store) feeds into a Multithreaded Instruction Buffer. The buffer outputs to an Operand Select unit, which then feeds into MAD (Memory Access) and SFU (Store Forwarding Unit) units.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
From L2: Hardware Overview

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

CS6963
From L2: Hardware Overview

Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
 - Status becomes ready after the needed values are deposited
 - prevents hazards
 - cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
 - any thread can continue to issue instructions until scoreboarding prevents issue
 - allows Memory/Processor ops to proceed in shadow of Memory/Processor ops

The diagram shows a timeline of instructions. Each instruction is divided into Thread Blocks (TB) and Warps (W). For example, TB1 W1 has 8 threads. Above the timeline, horizontal bars indicate stalls: TB1, W1 stall; TB2, W1 stall; and TB3, W2 stall. The legend states: TB = Thread Block, W = Warp.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign
From L2: Hardware Overview

Scoreboarding from Example

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

Warp	Current Instruction	Instruction State
Warp 1	42	Computing
Warp 3	95	Computing
Warp 8	11	Operands ready to go
...		

9
From L2: Hardware Overview

Scoreboarding from Example

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

Warp	Current Instruction	Instruction State
Warp 1	42	Ready to write result
Warp 3	95	Computing
Warp 8	11	Computing
...		

10
From L2: Hardware Overview

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)

11
From L2: Hardware Overview

Transparent Scalability

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

12
From L2: Hardware Overview

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

CS6963

L3: Memory Hierarchy, 1



Optimizing the Memory Hierarchy on GPUs, Overview

- Today's Lecture
- 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

CS6963

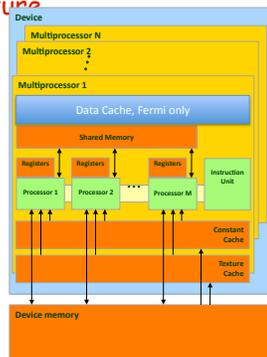
L3: Memory Hierarchy, 1

14



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)**



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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

Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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

CS6963

L3: Memory Hierarchy, 1



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

© David Kirk, NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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

CS6963

L3: Memory Hierarchy, 1



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

CS6963

L3: Memory Hierarchy, 1



Language Extensions: Variable Type Qualifiers

	Memory	Scope	Lifetime
<code>__device__ __local__ int LocalVar;</code>	local	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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;`

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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)

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



Constant Memory Example

- Signal recognition example:
 - Apply input signal (a vector) to a set of precomputed transform matrices
 - Compute M_1V, M_2V, \dots, M_nV

```
__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][threadIdx.x][j] *
            d_signalVector[j];
    R[blockIdx.x][threadIdx.x] = result;
}

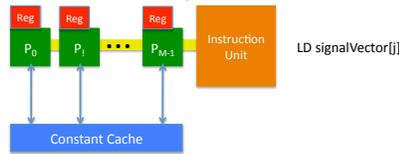
CS6963
```

L3: Memory Hierarchy, 1



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



CS6963

L3: Memory Hierarchy, 1



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.

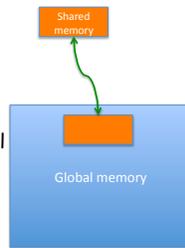
CS6963

L3: Memory Hierarchy, 1



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



CS6963

L3: Memory Hierarchy, 1



Mechanics of Using Shared Memory

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


```

__global__ void compute2() {
    __shared__ float d_s_array[M];

    extern __shared__ float d_s_array[]; // create or copy from global memory
    d_s_array[j] = ...;
    /* a form of dynamic allocation */ //synchronize threads before use
    /* MEMSIZE is size of per-block */ __syncthreads();
    /* shared memory*/ ... = d_s_array[x]; // now can use any element
    __host__ void outerCompute() { // more synchronization needed if updated
        compute<<<gs,bs>>>();
    }
    __global__ void compute() { // may write result back to global memory
        d_s_array[i] = ...;
        d_g_array[j] = d_s_array[j];
    }
}
            
```

CS6963

L3: Memory Hierarchy, 1



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

CS6963

L3: Memory Hierarchy, 1



Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j]+A[j+1]+A[j-1]
```

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

CS6963

L3: Memory Hierarchy, 1



Spatial Reuse (Ignore for now)

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

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j]+A[j+1]+A[j-1];
```

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

CS6963

L3: Memory Hierarchy, 1



Group Reuse

- Same data used by distinct references

```
for (i=1; i<N; i++)
  for (j=1; j<N; j++)
    A[j]= A[j]+A[j+1]+A[j-1];
```

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

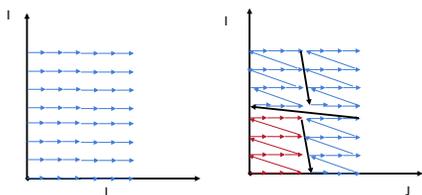
CS6963

L3: Memory Hierarchy, 1



Tiling (Blocking): Another Loop Reordering Transformation

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



CS6963

L3: Memory Hierarchy, 1



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];
```

CS6963

L3: Memory Hierarchy, 1



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

L3: Memory Hierarchy, 1

35



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

CS6963

L3: Memory Hierarchy, 1



Matrix Multiplication A Simple Host Version in C

```

// Matrix multiplication on the (CPU) host in double precision
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;
        }
}
    
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign L3: Memory Hierarchy, 1

Tiled Matrix Multiply Using Thread Blocks

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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign L3: Memory Hierarchy, 1

CUDA Code - Kernel Execution Configuration

```

// Setup the execution configuration
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(N.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.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign L3: Memory Hierarchy, 1

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 ;
}
    
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign L3: Memory Hierarchy, 1

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);

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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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();
```

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

L3: Memory Hierarchy, 1



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.

L3: Memory Hierarchy, 1



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

CS5963

L3: Memory Hierarchy, 1



Summary of Lecture

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

CS6963

L3: Memory Hierarchy, 1



Next Time

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

CS6963

L3: Memory Hierarchy, 1

