L4: Memory Hierarchy Optimization II, Locality and Data Placement, cont.

Administrative

- Next assignment available
  - Next three slides
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Tuesday, Feb. 8, 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

Assigment: 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
  - Revised formulation (for class purposes): compute MV_1, MV_2, …, MV_n

```
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

Assignment 2: What You Will Implement

We provide the sequential code. Your goal is to write two CUDA versions of this code:

(1) one that uses global memory

(2) one that uses a combination of global memory and shared memory

You'll time the code, but will not be graded on the actual performance. Rather, your score will be based on whether you produce two working versions of code, and the analysis of tradeoffs.

For your two versions, you should try three different thread and block decomposition strategies:

(1) a small number of blocks and a large number of threads

(2) a large number of blocks and fewer threads

(3) some intermediate point, or different number of dimensions in the block/thread decomposition

Assignment 2: Analyzing the Results

You'll need to perform a series of experiments and report on results. For each measurement, you should compute the average execution time of five runs.

What insights can you gain from the performance measurements and differences in behavior.

EXTRA CREDIT: Can you come up with a better implementation of this code? You can use other memory structures, or simply vary how much work is performed within a thread. How much faster is it?

Overview of Lecture

- Review: Where data can be stored (summary)
  - And how to get it there
- Review: Some guidelines for where to store data
  - Who needs to access it?
  - Read only vs. Read/Write
  - Footprint of data
- Slightly more detailed description of how to write code to optimize for memory hierarchy
  - More details next week
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

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

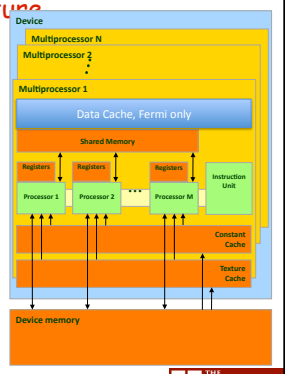
Optimizing the Memory Hierarchy on GPUs, Overview

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

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
    - NEW: surface memory can be written, but unsafe within same kernel
  - Data cache (Fermi only)



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

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

Data Placement: Conceptual

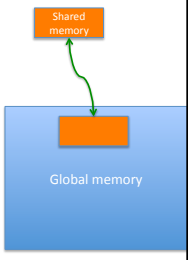
- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use SP shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only "reused" data can be placed in constant & texture 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

Recall: 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[];
/* a form of dynamic allocation */
/* MEMSIZE is size of per-block */
/* shared memory*/
__host__ void outerCompute() {
  compute<<<gs,bs>>>();
}
__global__ void compute() {
  d_s_array[i] = …;
}
```

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

  // create or copy from global memory
  d_s_array[j] = …;
  //synchronize threads before use
  __syncthreads();
  … = d_s_array[x]; // now can use any element

 // more synchronization needed if updated

  // may write result back to global memory
  d_g_array[j] = d_s_array[j];
}
```


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'

```
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`

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

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];
```

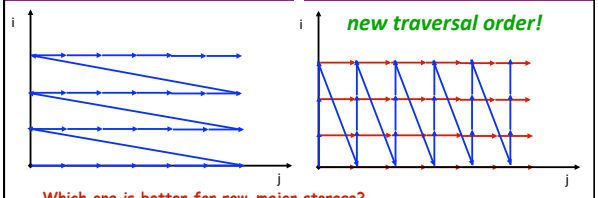
- `A[j]`,`A[j+1]` and `A[j-1]` have group reuse (spatial and temporal) in loop `j`

Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)
 for (i= 0; i<3; i++)
   A[i][j+1]=A[i][j]+B[j];
```



Which one is better for row-major storage?

Safety of Permutation

- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so changes the relative order of a read and write or two writes to the same memory location

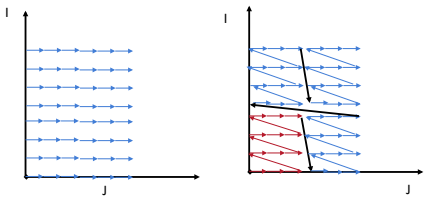
```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

```
for (i= 0; i<3; i++)
  for (j=0; j<6; j++)
    A[i+1][j-1]=A[i][j]
               +B[j];
```

- Ok to permute?

Tiling (Blocking): Another Loop Reordering Transformation

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



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 (Seq. view)

```
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 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

CUDA Version of Example (Tiling for Computation Partitioning)

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

Block dimension
Thread dimension
Loop within Thread

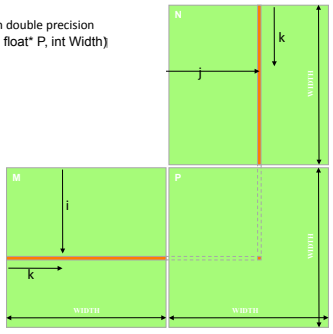
```
…
<<<ComputeI(N/s,s)>>>(d_D, d_B, N);
…

__global__ ComputeI (float *d_D, float *d_B, int N) {
      int ii = blockIdx.x;
      int i = ii*s + threadIdx.x;
      for (j=0; j<N; j++)
         d_D[i] = d_D[i] + d_B[j*N+i];
      }
```


Textbook Shows Tiling for Limited Capacity Shared Memory

- Compute Matrix Multiply using shared memory accesses
- We'll show how to derive it using tiling

Matrix Multiplication A Simple Host Version in C

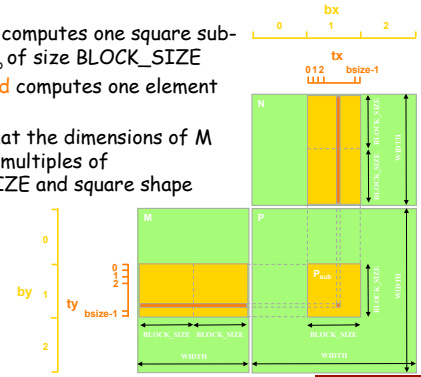
```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i * 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

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

Tiling View (Simplified Code)

```
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;
    }
```

```
for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
        double sum = 0;
        for (int k = 0; k < Width; ++k) {
            sum += M[i][k] * N[k][j];
        }
        P[i][j] = sum;
    }
```


Let's Look at This Code

```
for (int i = 0; i < Width; ++i)
    for (int j = 0; j < Width; ++j) {
        double sum = 0;
        for (int k = 0; k < Width; ++k) {
            sum += M[i][k] * N[k][j];
        }
        P[i][j] = sum;
    }
```

Tile i
Tile j
Tile k (inside thread)

Strip-Mined Code

```
for (int ii = 0; ii < Width; ii+=TI)
  for (int i=ii; i<ii*TI-1; i++)
   for (int jj=0; jj<Width; jj+=TJ)
     for (int j = jj; j < jj*TJ-1; j++) {
        double sum = 0;
       for (int kk = 0; kk < Width; kk+=TK) {
        for (int k = kk; k < kk*TK-1; k++)
            sum += M[i][k] * N[k][j];
        }
        P[i][j] = sum;
     }
```

Block dimensions
Thread dimensions

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

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 - 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);
```


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 shared memory
- Introduction to Tiling transformation
  - For computation partitioning
  - For limited capacity in shared memory
- Matrix multiply example

Next Time

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