L5: Memory Hierarchy Optimization III, Data Placement, cont. and Memory Bandwidth Optimizations

Administrative

- Next assignment available
  - Next four 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?

How to tell if results are correct

- Parallel execution may involve reordering the updates to memory locations
  - Recall "reduction" count6s from L1
- Correct for commutative and associative operations (addition in this case)
  - Is IEEE floating point associative? (not really)
- Also, GPU and CPU arithmetic not always identically implemented
  - Even completely independent operations may yield different results when comparing CPU and GPU implementations

SO, we compare floating point values to a particular level of error tolerance to determine correctness

Example:

```
CUTBoolean res = cutComparefe( d_P, h_P, Width*Width, 0.00001);
```


Overview of Lecture

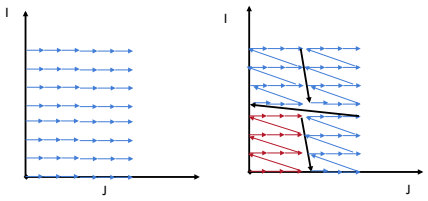
- Review: Tiling for computation partitioning and fixed capacity storage
- Review: More detailed derivation of matrix multiply from text
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

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

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];
```


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

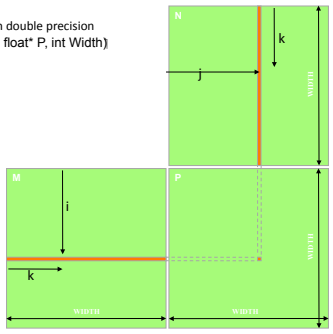
```
…
<<<ComputeI(N/s,s)>>>(d_D, d_B, N);
…

__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

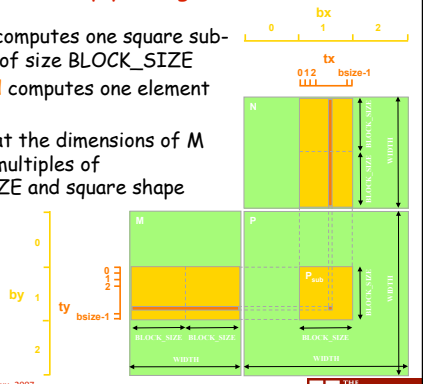
Matrix Multiplication
A Simple Host Version in C

```
// 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;
        }
}
```



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



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

To be used to stage data in shared memory

Derivation of code in text

- TI = TJ = TK = "TILE_WIDTH"
- All matrices square, Width x Width
- Copies of M and N in shared memory
  - TILE_WIDTH x TILE_WIDTH
- "Linearized" 2-d array accesses:
  a[i][j] is equivalent to a[i*Row + j]
- Each SM computes a "tile" of output matrix P from a block of consecutive rows of M and a block of consecutive columns of N
  - dim3 Grid (Width/TILE_WIDTH, Width/TILE_WIDTH);
  - dim3 Block (TILE_WIDTH, TILE_WIDTH)
- Then, location P[i][j] corresponds to
  P [by*TILE_WIDTH+ty] [bx*TILE_WIDTH+tx] or
  P[Row][Col]

Final Code (from text, p. 87)

```
__global__ void MatrixMulKernel (float *Md, float *Nd, float *Pd, int Width) {
1.    __shared__ float Mds [TILE_WIDTH] [TILE_WIDTH];
2.    __shared__ float Nds [TILE_WIDTH] [TILE_WIDTH];
3 & 4.    int bx = blockIdx.x; int by = blockIdx.y;  int tx = threadIdx.x; int ty = threadIdx.y;
//Identify the row and column of the Pd element to work on
5 & 6.    int Row = by * TILE_WIDTH + ty;   int Col = bx * TILE_WIDTH + tx;
7.        float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.        for (int m=0; m < Width / TILE_WIDTH; ++m) {
// Collaborative (parallel) loading of Md and Nd tiles into shared memory
9.          Mds [ty] [tx] = Md [Row*Width + (m*TILE_WIDTH + tx)];
10.         Nds [ty] [tx] = Nd [(m*TILE_WIDTH + ty)*Width + Col];
11.         __syncthreads();                    // make sure all threads have completed copy before calculation
12.         for (int k = 0; k < TILE_WIDTH; ++k)    // Update Pvalue for TKxTK tiles in Mds and Nds
13.           Pvalue += Mds [ty] [k] * Nds [k] [tx];
14.         __syncthreads();                    // make sure calculation complete before copying next tile
          } // m loop
15.       Pd [Row*Width + Col] =  Pvalue;
}
```


Performance

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

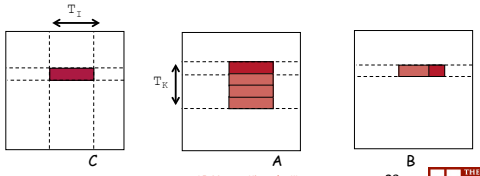
"Tiling" for Registers

- A similar technique can be used to map data to registers
- Unroll-and-jam
  - Unroll outer loops in a nest and fuse together resulting inner loops
  - Equivalent to "strip-mine" followed by permutation and unrolling
- Fusion safe if dependences are not reversed
- Scalar replacement
  - May be followed by replacing array references with scalar variables to help compiler identify register opportunities

Unroll-and-jam for matrix multiply

Tiling inner loops I and K (+permutation)

```
for (K = 0; K<N; K+=T_K)
  for (I = 0; I<N; I+=T_I)
    for (J =0; J<N; J++)
      for (KK = K; KK<min(K+T_K, N); KK++)
        for (II = I; II<min(I+ T_I, N); II++)
          P[J][II] = P[J][II] + M[KK][II] * N[J][KK];
```

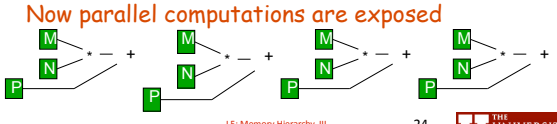


First, Apply Unroll-and-Jam

Unroll II loop, T_I = 4 (equiv. to unroll&jam)

```
for (K = 0; K<N; K+=T_K)
  for (I = 0; I<N; I+=4)
    for (J =0; J<N; J++)
      for (KK = K; KK<min(K+T_K, N); KK++)
        P[J][II] = P[J][II] + M[KK][II] * N[J][KK];
        P[J][II+1] = P[J][II+1] + M[KK][II+1] * N[J][KK];
        P[J][II+2] = P[J][II+2] + M[KK][II+2] * N[J][KK];
        P[J][II+3] = P[J][II+3] + M[KK][II+3] * N[J][KK];
```

Now parallel computations are exposed



Now can expose registers using scalar replacement (or simply unroll kk loop)

Scalar Replacement: Replace accesses to P with scalars

```
for (K = 0; K<N; K+=T_K)
    for (I = 0; I<N; I+=4)
        for (J =0; J<N; J++) {
            P0 = P[J][I]; P1 = P[J][I+1]; P2 = P[J][I+2,J]; P3 = P[J][I+3];
            for (KK = K; KK<min(K+T_K, N); KK++) {
                P0 = P0 + M[KK][II] * N[J][KK];
                P1 = P1 + M[KK][II+1] * N[J][KK];
                P2 = P2 + M[KK][II+2] * N[J][KK];
                P3 = P3 + M[KK][II+3] * N[J][KK];
            }
            P[J][I] = P0; P[J][I+1] = P1; P[J][I+2] = P2; P[J][I+3] = P3;
        }
```

Now P accesses can be mapped to "named registers"

Overview of Texture Memory

- Recall, texture cache of read-only data
- Special protocol for allocating and copying to GPU
  - texture<Type, Dim, ReadMode> texRef;
    - Dim: 1, 2 or 3D objects
- Special protocol for accesses (macros)
  - tex2D(<name>,dim1,dim2);
- In full glory can also apply functions to textures
- Writing possible, but unsafe if followed by read in same kernel

Using Texture Memory (simpleTexture project from SDK)

```
cudaMalloc( (void**) &d_data, size);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(32, 0, 0, 0,
    cudaChannelFormatKindFloat);
cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height );
cudaMemcpyToArray( cu_array, 0, 0, h_data, size, cudaMemcpyHostToDevice);
// set texture parameters
tex.addressMode[0] = tex.addressMode[1] = cudaAddressModeWrap;
tex.filterMode = cudaFilterModeLinear; tex.normalized = true;
cudaBindTextureToArray( tex,cu_array, channelDesc);
// execute the kernel
transformKernel<<< dimGrid, dimBlock, 0 >>>( d_data, width, height, angle);

Kernel function:
// declare texture reference for 2D float texture
texture<float, 2, cudaReadModeElementType> tex;

... = tex2D(tex,i,j);
```


When to use Texture (and Surface) Memory

(From 5.3 of CUDA manual) Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- If memory reads to global or constant memory will not be coalesced, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads (this is less likely for devices of compute capability 2.x given that global memory reads are cached on these devices);
- Addressing calculations are performed outside the kernel by dedicated units;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] (see Section 3.2.4.1.1).

Memory Bandwidth Optimization

- Goal is to maximize utility of data for each data transfer from global memory
- Memory system will "coalesce" accesses for a collection of consecutive threads if they are within an aligned 128 byte portion of memory (from half-warp or warp)
- Implications for programming:
  - Desirable to have consecutive threads in tx dimension accessing consecutive data in memory
  - Significant performance impact, but Fermi data cache makes it slightly less important

Introduction to Global Memory Bandwidth: Understanding Global Memory Accesses

Memory protocol for compute capability 1.2* (CUDA Manual 5.1.2.1)

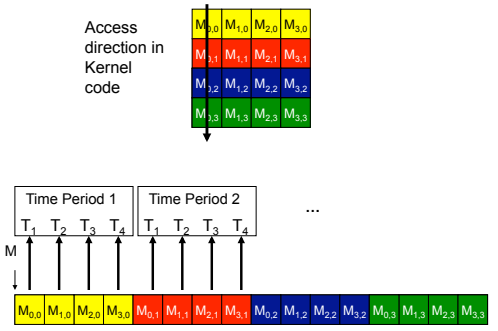
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory and mark threads as "inactive"
- Repeat until all threads ***in half-warp*** are serviced

*Includes Tesla and GTX platforms

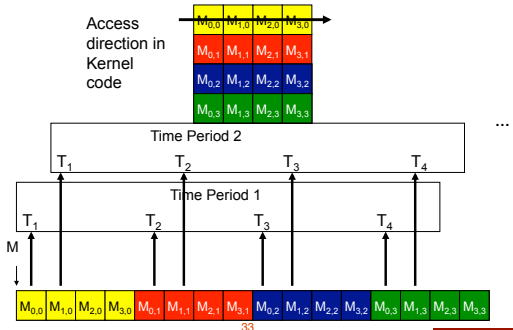
Protocol for most systems (including lab6 machines) even more restrictive

- For compute capability 1.0 and 1.1
  - Threads must access the words in a segment in sequence
  - The kth thread must access the kth word

Memory Layout of a Matrix in C



Memory Layout of a Matrix in C



Summary of Lecture

- Tiling transformation
  - For computation partitioning
  - For limited capacity in shared memory
  - For registers
- Matrix multiply example
- Unroll-and-jam for registers
- Bandwidth optimization
  - Global memory coalescing

Next Time

- Complete bandwidth optimizations
  - Shared memory bank conflicts
  - Bank conflicts in global memory (briefly)