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

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

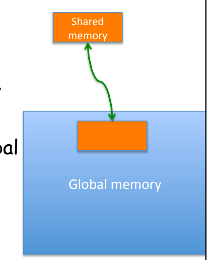
- Device memory access times non-uniform so ***data placement*** significantly affects performance.
  - But controlling data placement may require additional copying, so consider overhead.
- Optimizations to increase memory bandwidth. Idea: maximize utility of each memory access.
  - ***Coalesce*** global memory accesses
  - ***Avoid memory bank conflicts*** to increase memory access parallelism
  - ***Align*** data structures to address boundaries

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

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] = …;
}
```

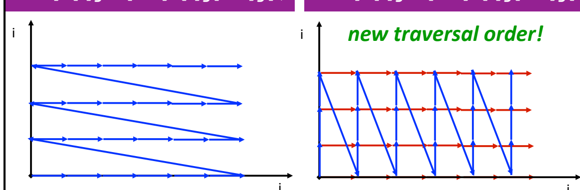
```
__global__ void compute2() {
   __shared__ float d_s_array[M];

   // 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];
}
```


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

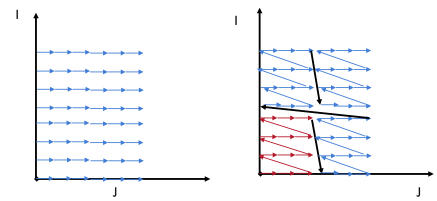
```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
  A[i][j+1]=A[i][j]+B[j];
```

```
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
  - To partition computation across blocks and threads
  - Across thread blocks if shared data exceeds shared memory capacity
  - 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)              ← Block dimension
for (i=ii; i<min(ii+s-1,N); i++)     ← Thread dimension
   for (j=1; j<N; j++)               ← Loop within Thread
      D[i] = D[i] + B[j][i];
```

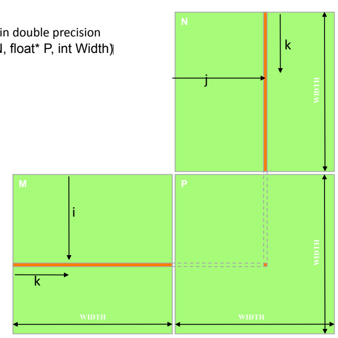
```
…
<<<ComputeI(N/s,s)>>>(d_D, d_B, N);
…

__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

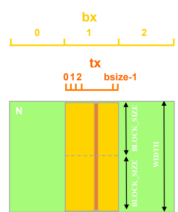
Matrix Multiplication
A Simple Host Version in C

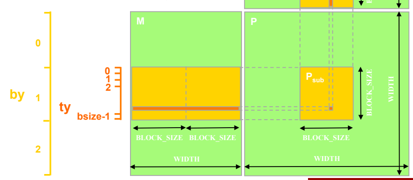
```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i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


Strip-Mined Code

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

But this code doesn't match CUDA Constraints

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

Block dimensions – must be unit stride

Thread dimensions – must be unit stride

Tiling for shared memory

Can we fix this?

Unit Stride Tiling – Reflect Stride in Subscript Expressions

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

Block dimensions – must be unit stride

Thread dimensions – must be unit stride

Tiling for shared memory, no need to change

What Does this Look Like in CUDA

```
#define TI 32
#define TJ 32
dim3  dimGrid(Width/TI, Width/TJ);
dim3  dimBlock(TI,TJ);
matMult<<<dimGrid,dimBlock>>>(M,N,P);

__global__ matMult(float *M, float *N, float *P) {
    ii = blockIdx.x; jj = blockIdx.y;
    i = threadIdx.x; j = threadIdx.y;
    double sum = 0;
    for (int kk = 0; kk < Width; kk+=TK) {
      for (int k = kk; k < kk+TK; k++)
        sum += M[(ii*TI+i)*Width+k] *
               N[k*Width+jj*TJ+j];
    }
    P[(ii*TI+i)*Width+jj*TJ+j] = sum;
}
```

Block and thread loops disappear

Tiling for shared memory, next slides

Array accesses to global memory are "linearized"

What Does this Look Like in CUDA

```
#define TI 32
#define TJ 32
#define TK 32
...
__global__ matMult(float *M, float *N, float *P) {
     ii = blockIdx.x; jj = blockIdx.y;
     i = threadIdx.x; j = threadIdx.y;
     __shared__ Ms[TI][TK], Ns[TK][TJ];
     double sum = 0;
     for (int kk = 0; kk < Width; kk+=TK) {
         Ms[j][i] = M[(ii*TI+i)*Width+TJ*jj+j+kk];
         Ns[j][i] = N[(ii*TI+i+kk)*Width+TJ*jj+j];
         __syncthreads();
         for (int k = kk; k < kk+TK; k++)
           sum += Ms[k%TK][i] * Ns[j][k%TK];
         __synchthreads();
     }
     P[(ii*TI+i)*Width+jj*TJ+j] = sum;
}
```

Tiling for shared memory

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.

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.

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;
}
```


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