L4: Memory Hierarchy Optimization II, Locality and Data Placement

Administrative

- Assignment due Friday, Jan. 18, 5 PM
  - Use handin program on CADE machines
    - "handin CS6235 lab1 <probfile>"
- Mailing list
  - CS6235@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!

Overview of Lecture

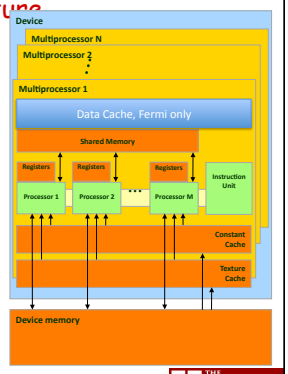
- More on tiling for shared memory and constant memory
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

Review: 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

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)



Review: 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

Memory Hierarchy Example: Matrix vector multiply

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    a[i] += c[j][i] * b[j];
  }
}
```

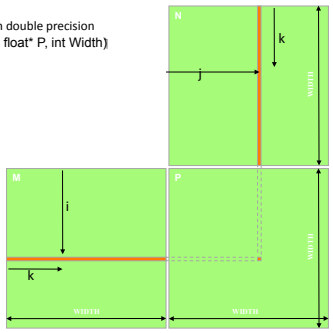
Remember to:
- Consider correctness of parallelization strategy (next week)
- Exploit locality in shared memory and registers

Resulting CUDA code (Automatically Generated by our Research Compiler)

```
__global__ mv_GPU(float* a, float* b, float** c) {
 int bx = blockIdx.x; int tx = threadIdx.x;
 __shared__ float bcpy[32];
 double acpy = a[tx + 32 * bx];
 for (k = 0; k < 32; k++) {
   bcpy[tx] = b[32 * k + tx];
   __syncthreads();
  //this loop is actually fully unrolled
   for (j = 32 * k; j <= 32 * k + 32; j++) {
     acpy = acpy + c[j][32 * bx + tx] * bcpy[j];
   }
   __synchthreads();
 }
 a[tx + 32 * bx] = acpy;
}
```

Ch. 4: 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;
      }
}
```



Discussion (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 for shared memory (inside thread)

Strip-Mined Code

```
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
← Thread dimensions
← Tiling for shared memory

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

Block and thread loops disappear

Tiling for shared memory

Array accesses to global memory are "linearized"

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;
}
```


Summary of Lecture

- Matrix-matrix multiply example