# L5: Memory Bandwidth Optimization

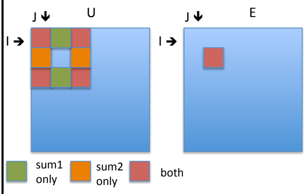
## Administrative

- Next assignment available
  - Next three slides
  - Goals of assignment:
    - simple memory hierarchy management
    - block-thread decomposition tradeoff
  - Due Friday, Feb. 8, 5PM
  - Use handin program on CADE machines
    - "handin CS6235 lab2 <probfile>"

## Assignment 2: Memory Hierarchy Optimization Due Fri day, February 8 at 5PM

Sobel edge detection:

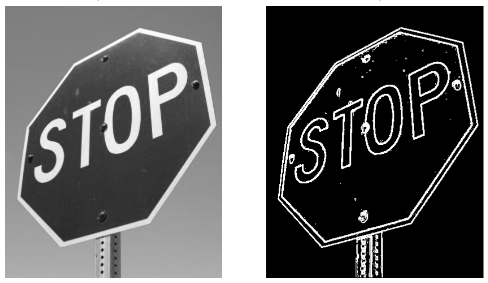
Find the boundaries of the image where there is significant difference as compared to neighboring "pixels" and replace values to find edges



```
for (i = 1;  i < ImageNRows - 1; i++)
   for (j = 1; j < ImageNCols -1; j++)
    {
     sum1 = u[i-1][j+1] - u[i-1][j-1]
         + 2 * u[i][j+1] - 2 * u[i][j-1]
         + u[i+1][j+1] - u[i+1][j-1];
     sum2 = u[i-1][j-1] + 2 * u[i-1][j] + u[i-1][j+1]
         - u[i+1][j-1] - 2 * u[i+1][j] - u[i+1][j+1];

     magnitude =  sum1*sum1 + sum2*sum2;
     if (magnitude > THRESHOLD)
      e[i][j] = 255;
     else
      e[i][j] = 0;
    }
```

## Example

Input Output



## General Approach

0. Provided
   - a. Input file
   - b. Sample output file
   - c. CPU implementation

1. Structure
   - a. Compare CPU version and GPU version output [compareInt]
   - b. Time performance of two GPU versions (see 2 & 3 below) [EventRecord]

2. GPU version 1 (partial credit if correct)
   implementation using global memory

3. GPU version 2 (highest points to best performing versions)
   use memory hierarchy optimizations from previous, current lecture

4. Extra credit: Try two different block / thread decompositions. What happens if you use more threads versus more blocks? What if you do more work per thread? Explain your choices in a README file.

Handin using the following on CADE machines, where probfile includes all files

"handin cs6235 lab2 <probfile>"

## Overview of Lecture

- Tiling for constant memory and registers
- Global Memory Coalescing
- Reading:
  - Chapter 5, Kirk and Hwu book

## 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

## Discussion (Simplified Code)

| | |
|---|---|
| <pre>for (int i = 0; i < Width; ++i)<br>    for (int j = 0; j < Width; ++j) {<br>        double sum = 0;<br>        for (int k = 0; k < Width; ++k) {<br>            double a = M[i * width + k];<br>            double b = N[k * width + j];<br>            sum += a * b;<br>        }<br>        P[i * Width + j] = sum;<br>    }</pre> | <pre>for (int i = 0; i < Width; ++i)<br>    for (int j = 0; j < Width; ++j) {<br>        double sum = 0;<br>        for (int k = 0; k < Width; ++k) {<br>            sum += M[i][k] * N[k][j];<br>        }<br>        P[i][j] = sum;<br>    }</pre> |

## 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.y; jj = blockIdx.x;
      i = threadIdx.y; j = threadIdx.x;
      __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[(kk*TK+i)*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

Now eliminate mods

## 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"

## 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;
}
```

## "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 relative order of memory reads and writes is preserved
- Scalar replacement
  - May be followed by replacing array references with scalar variables to help compiler identify register opportunities

## Unroll and Jam: Matrix Multiply Code Example

```
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;
    }
```

```
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;
    }
```

## Unroll J Loop and Fuse K Loop Copies

```
for (int i = 0; i < Width; i++)
    for (int j = 0; j < Width; j+=2) {
      double sum1 = sum2 = 0;
      for (int k = 0; k < Width; k++) {
         sum1 += M[i][k] * N[k][j];
         sum2 += M[i][k] * N[k][j+1];
      }
      P[i][j] = sum1;
      P[i][j+1] = sum2;
    }
```

Why is this helpful?
- Reuse M[i][k], possibly in register
- Two independent memory streams increases instruction-level parallelism

## Unroll I and J Loops and Fuse J, K Loop Copies

```
for (int i = 0; i < Width; i+=2)
    for (int j = 0; j < Width; j+=2) {
      double sum1 = sum2 = sum3 = sum4 = 0;
      for (int k = 0; k < Width; k++) {
         sum1 += M[i][k] * N[k][j];
         sum2 += M[i][k] * N[k][j+1];
         sum3 += M[i+1][k] * N[k][j];
         sum4 += M[i+1][k] * N[k][j+1];
      }
      P[i][j] = sum1;
      P[i][j+1] = sum2;
      P[i+1][j] = sum3;
      P[i+1][j+1] = sum4;
    }
```

Why is this helpful?
- Added reuse of N
- More independent memory streams

This code is almost always faste than the original on ANY architecture

More unrolling is better up to a point where registers are exceeded

## Scalar Replacement (beyond sum)

```
for (int i = 0; i < Width; i+=2)
    for (int j = 0; j < Width; j+=2) {
      double sum1 = sum2 = sum3 = sum4 = 0;
      for (int k = 0; k < Width; k++) {
         tmpm1 = M[i][k]; tmpm2 = M[i+1][k];
         tmpn1 = N[k][j]; tmpn2 = N[k][j+1];
         sum1 += tmpm1 * tmpn1;
         sum2 += tmpm1 * tmpn2;
         sum3 += tmpm2 * tmpn1;
         sum4 += tmpm2 * tmpn2;
      }
      P[i][j] = sum1;
      P[i][j+1] = sum2;
      P[i+1][j] = sum3;
      P[i+1][j+1] = sum4;
    }
```

**Scalar Replacement**
- Replace array variables with scalar temporaries
- Sometimes this helps compilers put array variables in registers, but not always necessary
- This code is almost always faster than the original on ANY architecture
- More unrolling is better up to a point where registers are exceeded

## Constant Memory Example

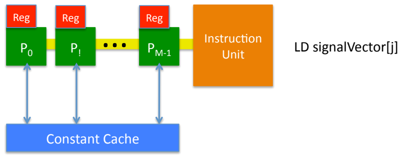
- Signal recognition example:
  - Apply input signal (a vector) to a set of precomputed transform matrices
  - Compute $M_1V$, $M_2V$, …, $M_nV$

```
__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;
}
```

## 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



## 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? -- tile for constant memory, constant cache
    - Cache latency proportional to number of accesses in a warp
  - No reuse?
    - Should not be in constant memory

## 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)

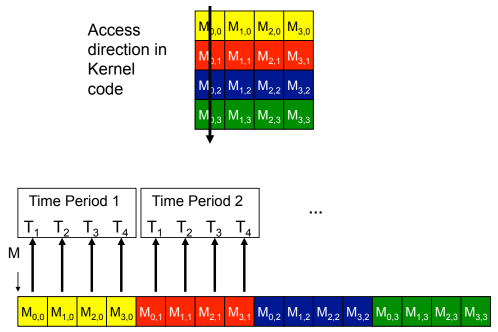
- Start with memory request by smallest numbered thread. Find the memory segment that contains the address (32, 64 or 128 byte segment, depending on data type)
- Find other active threads requesting addresses within that segment and ***coalesce***
- Reduce transaction size if possible
- Access memory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

*Includes Tesla and GTX platforms

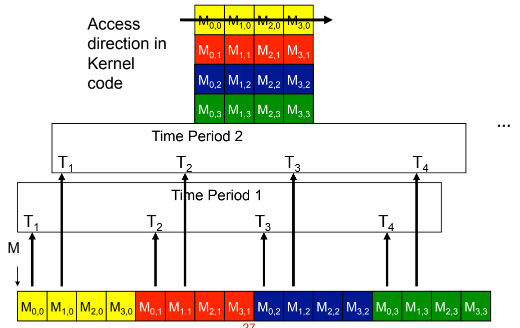
## Protocol for most systems (including lab6 machines) even more restrictive

- 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

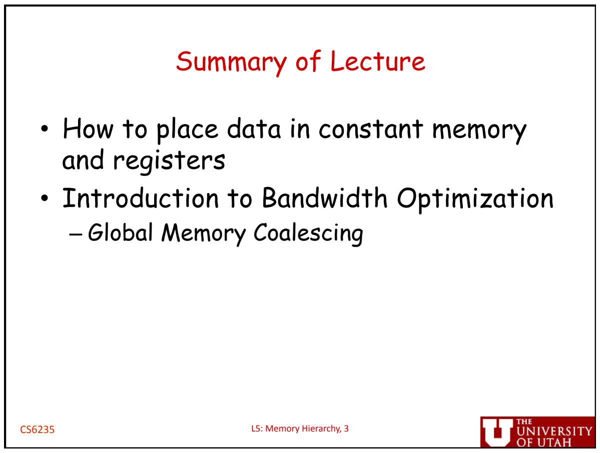

Access direction in Kernel code
Time Period 1
Time Period 2
...
M
$T_1$ $T_2$ $T_3$ $T_4$
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

## Memory Layout of a Matrix in C


Access direction in Kernel code
Time Period 2
Time Period 1
...
M
$T_1$ $T_2$ $T_3$ $T_4$
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

## Coalescing in Matrix Multiply

```
__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)];    COALESCED – See tx
                                                                            This one required a transpose
10.             Nds [ty] [tx] = Nd [(m*TILE_WIDTH + ty)*Width + Col];    COALESCED – See Col
11.             __syncthreads();
12.             for (int k = 0; k < TILE_WIDTH; ++k)
13.                 Pvalue += Mds [ty] [k] * Nds [k] [tx];
14.             __syncthreads();
            }  // m loop
15.         Pd [Row*Width + Col]  =  Pvalue;      <-----        COALESCED – See Col
}
```



## Summary of Lecture

- How to place data in constant memory and registers
- Introduction to Bandwidth Optimization
  - Global Memory Coalescing

CS6235 L5: Memory Hierarchy, 3 THE UNIVERSITY OF UTAH