# L5: Memory Hierarchy Optimization II, Locality and Data Placement, cont.



## Administrative

- Next assignment on the website
  - Description at end of class
  - Due Wednesday, Feb. 17, 5PM
  - Use handin program on CADE machines
    - "handin cs6963 lab2 <probfile>"
- Mailing lists
  - cs6963s10-discussion@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!
  - cs6963s10-teach@list.eng.utah.edu
    - Please use for questions to Protonu and me



## Overview

- High level description of how to write code to optimize for memory hierarchy
  - Code reordering transformations: permutation, tiling, unroll-and-jam
  - Placing data in registers and texture memory
  - Introduction to bandwidth optimization for global memory
- Reading:
  - Chapter 4, Kirk and Hwu
  - http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf
  - Sections 3.2.4 (texture memory) and 5.1.2 (bandwidth optimizations) of NVIDIA CUDA Programming Guide



## 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

- 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



## Aside: Capacity Questions

- How much shared memory, global memory, registers, constant memory, constant cache, etc.?
  - deviceQuery function (in SDK) instantiates variable of type cudaDeviceProp with this information and prints it out.
- Summary for 9400 M
  - 8192 registers per SM
  - 16KB shared memory per SM
  - 64KB constant memory
    - stored in global memory
    - presumably, 8KB constant cache
  - 256MB global memory



## 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



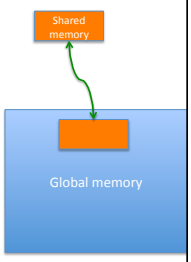
## Rest of Today's Lecture

- Mechanics of how to place data in shared memory and constant memory
- Tiling transformation to reuse data within
  - Shared memory
  - Constant cache



## Now Let's Look at 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






## Can Use Reordering Transformations!

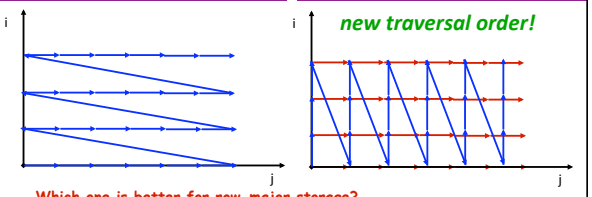
- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - **Safety?** (doesn't reverse dependences)
  - **Profitablity?** (improves locality)

## 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 reverses the direction of any dependence.

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

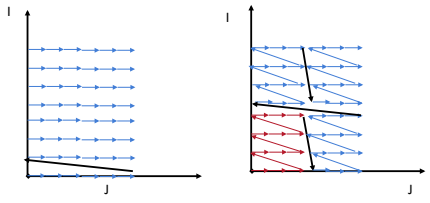
```
for (i= 0; i<3; i++)
  for (j=1; j<6; j++)
    A[i+1][j-1]=A[i][j]
                +B[j];
```

- Ok to permute?



## Tiling (Blocking): Another Loop Reordering Transformation

- Blocking 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

```
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 = strip-mine and permutation
  - Strip-mine does not reorder iterations
  - Permutation must be legal

  OR
  - strip size less than dependence distance



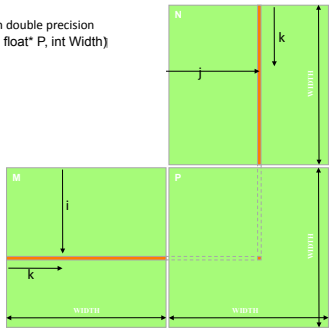
## A Few Words On Tiling

- 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



## 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;
        }
}
```

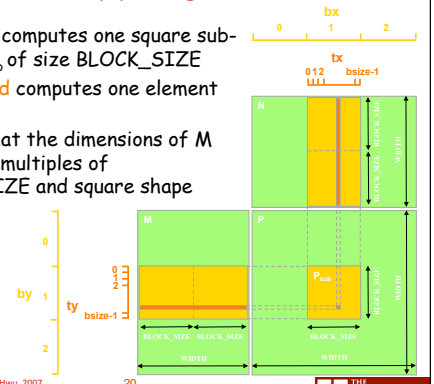






## Tiled Matrix Multiply Using Thread Blocks

- 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



## Shared Memory Usage

- Assume each SMP has 16KB shared memory
  - Each Thread Block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 8 Thread Blocks actively executing
  - For BLOCK_SIZE = 16, this allows up to 8*512 = 4,096 pending loads
    - In practice, there will probably be up to half of this due to scheduling to make use of SPs.
  - The next BLOCK_SIZE 32 would lead to 2*32*32*4B= 8KB shared memory usage per Thread Block, allowing only up to two Thread Blocks active at the same time



## First-order Size Considerations

- Each Thread Block should have many threads
  - BLOCK_SIZE of 16 gives 16*16 = 256 threads
- And many Thread Blocks
  - A 1024*1024 P Matrix gives 64*64 = 4096 Thread Blocks
- Each thread block performs 2*256 = 512 float loads from global memory for 256 * (2*16) = 8,192 mul/add operations.
  - Memory bandwidth no longer a limiting factor



## 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 2.0) yields just under 400 Gflops.




## 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
- 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
  - Used to be important because earlier compilers would not place array variables in registers, but not the case with nvcc compiler



## 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



## 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);
```



## 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

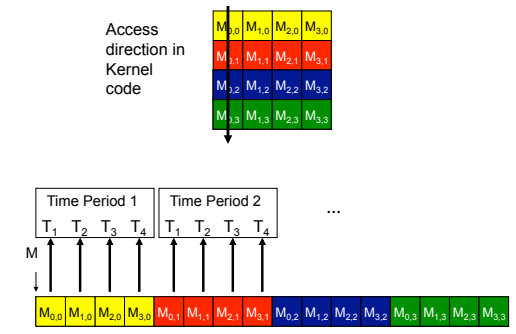
*Includes Tesla and GTX platforms

## 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



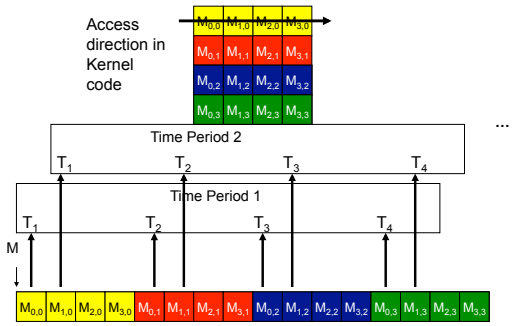
## Memory Layout of a Matrix in C







## Memory Layout of a Matrix in C







## Summary of Lecture

- Reordering transformations to improve locality
  - Tiling, permutation and unroll-and-jam
- Matrix multiply example for shared memory
- Guiding data to be placed in registers
- Placing data in texture memory
- Introduction to global memory bandwidth (if time)

## Next Time

- Bandwidth optimizations
  - Global memory access coalescing
  - Avoiding bank conflicts in shared memory
- If time permits, cudaProfiler
  - How to tell if your optimizations are working