L7: Memory Hierarchy Optimization, cont.

Administrative

- Homework #2, posted on website
  - Due 5PM, Thursday, February 19
  - Use handin program to submit
- Project proposals
  - Due 5PM, Friday, March 13 (hard deadline)
  - Discuss today

Outline

- Homework discussion
- Project discussion
- Complete tiling discussion and matrix multiply example
- Calculating data reuse and data footprint

Project Proposal

- Project Logistics:
  - 2-3 person teams
  - Significant implementation, worth 55% of grade
  - Parts: proposal, design review (3/30 and 4/1), final presentation and report (end of semester)
  - Each person turns in the proposal (should be same as other team members)
- Proposal:
  - 3-4 page document (11pt, single-spaced)
  - Submit with handin program:
    "handin cs6963 prop <pdf-file>"

Content of Proposal

I. Team members: Name and a sentence on expertise for each member
II. Problem description
  - What is the computation and why is it important?
  - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
III. Suitability for GPU acceleration
  - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation. Use measurements from CPU execution of computation if possible.
  - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
  - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
IV. Intellectual Challenges
  - Generally, what makes this computation worthy of a project?
  - Point to any difficulties you anticipate at present in achieving high speedup

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 (last homework problem)
  - 8192 registers per SM
  - 16KB shared memory per SM
  - 64KB constant memory
    - stored in global memory
    - presumably, 8KB constant cache
  - 256MB global memory

Main points from Previous Lecture

- Considered latencies of different levels of memory hierarchy
  - Global memory latency roughly hundreds of cycles
  - Registers, shared memory and constant cache roughly single cycle latency
  - Constant memory (stored in global memory) can be used for read-only data, but only a win if it is cached
- Examples showing how to place data in constant or shared memory
- Tiling transformation for managing limited capacity storage (shared memory, constant cache, global memory, even registers)

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

- 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.
  - ***Align*** data structures to address boundaries
  - ***Coalesce*** global memory accesses
  - ***Avoid memory bank conflicts*** to increase memory access parallelism

Reuse and Locality

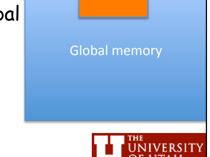
- Consider how data is accessed
  - ***Data reuse:***
    - Same data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

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

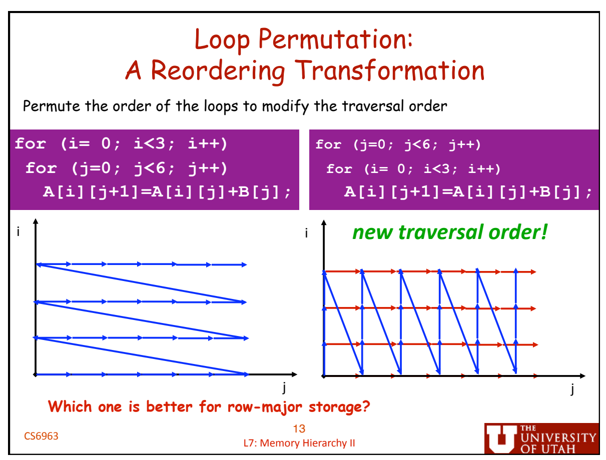
Shared memory

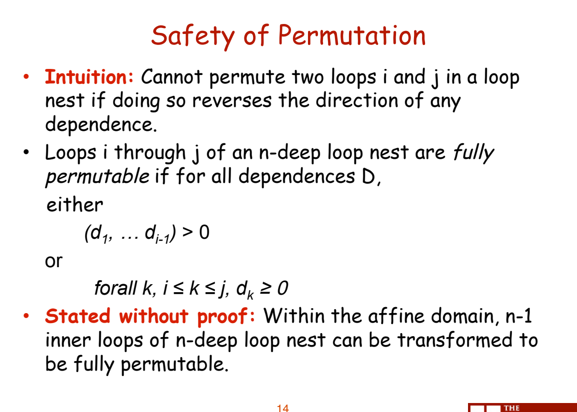
Global memory



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)





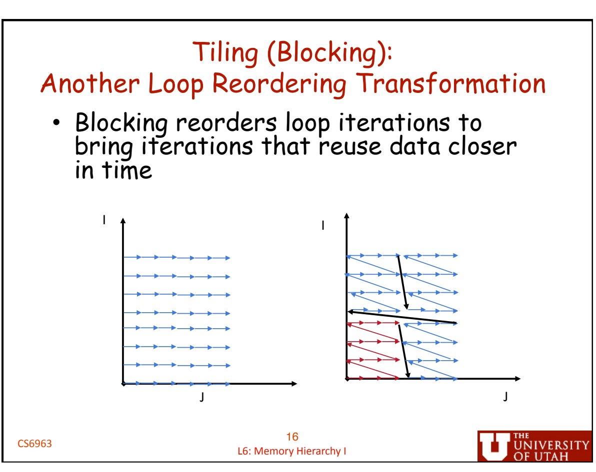
Simple Examples: 2-d Loop Nests

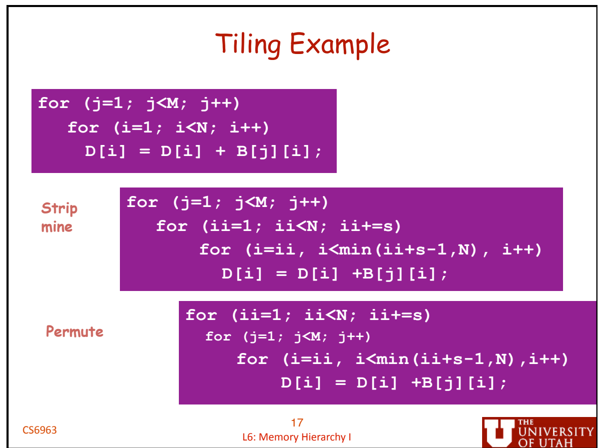
```
for (i= 0; i<3; i++)
 for (j=0; j<6; j++)
   A[i][j+1]=A[i][j]+B[j];
```

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

- Distance vectors
- Ok to permute?

CS6963 15 L6: Memory Hierarchy I





Legality of Tiling

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

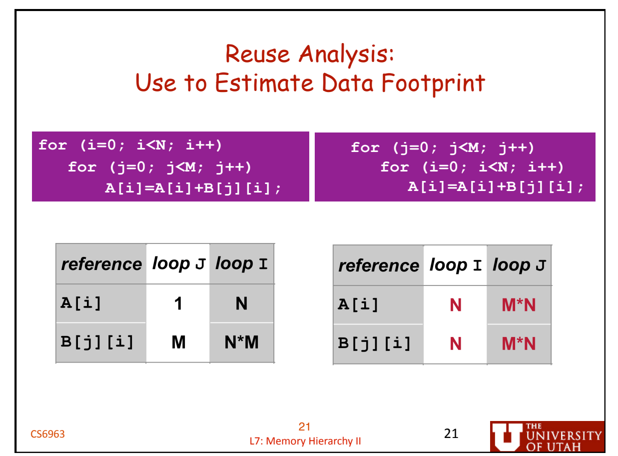
  OR
  - strip size less than dependence distance

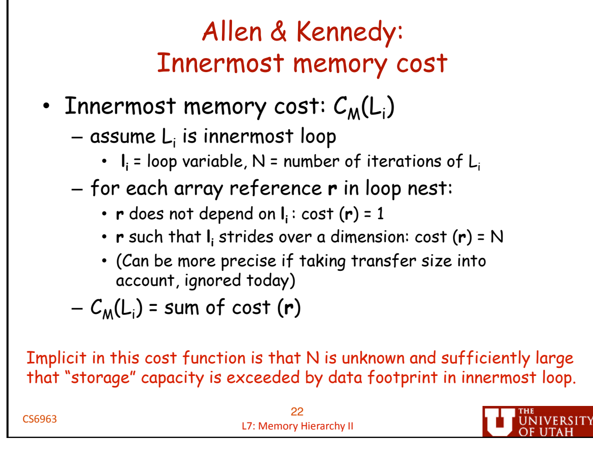
A Few Words On Tiling

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity
  - Within threads if data in constant cache exceeds cache capacity
  - Special form (unroll-and-jam) used for registers

Locality Optimization

- Reuse analysis can be formulated in a manner similar to dependence analysis
  - Particularly true for temporal reuse
  - Spatial reuse requires special handling of most quickly varying dimension (still ignoring)
- Simplification for today's lecture
  - Estimate data footprint for innermost loop for different scenarios
  - Select scenario that minimizes footprint

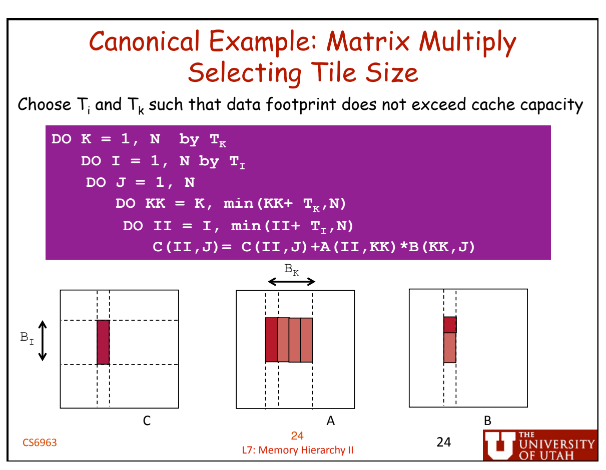




Canonical Example: matrix multiply
Selecting Loop Order for Cache-based Architecture

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
        C(I,J)= C(I,J) + A(I,K) * B(K,J)
```

- C_M(I) = $2N^3$/cls + N^2
- C_M(J) = $2N^3$ + N^2
- C_M(K) = N^3 + N^3/cls + N^2
- **Ordering by innermost loop cost: (J, K, I)**



"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
- Scalar replacement
  - May be followed by replacing array references with scalar variables to help compiler identify register opportunities

Unroll II, TI = 4 (Equiv. to unroll-and-jam)

```
DO K = 1, N  by TK
   DO I = 1, N by 4
    DO J = 1, N
       DO KK = K, min(KK+ TK,N)
           C(I,J)= C(I,J)+A(I,KK)*B(KK,J)
           C(I+1,J)= C(I+1,J)+A(I+1,KK)*B(KK,J)
           C(I+2,J)= C(I+2,J)+A(I+2,KK)*B(KK,J)
           C(I+3,J)= C(I+3,J)+A(I+3,KK)*B(KK,J)
```

In other architectures with deep instruction pipelines, this optimization can also be used to expose instruction-level parallelism.

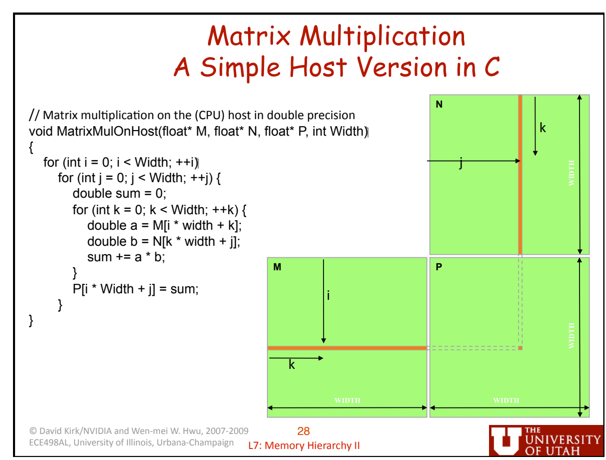
Scalar Replacement

```
DO K = 1, N  by TK
   DO I = 1, N by 4
    DO J = 1, N
       C1 = C(I,J) C2 =C(I+1,J) C3=C(I+2,J) C4=C(I+3,J)
       DO KK = K, min(KK+ TK,N)
           C1 = C1+A(I,KK)*B(KK,J)
           C2 = C2+A(I+1,KK)*B(KK,J)
           C3 = C3+A(I+2,KK)*B(KK,J)
           C4 = C4+A(I+3,KK)*B(KK,J)
```

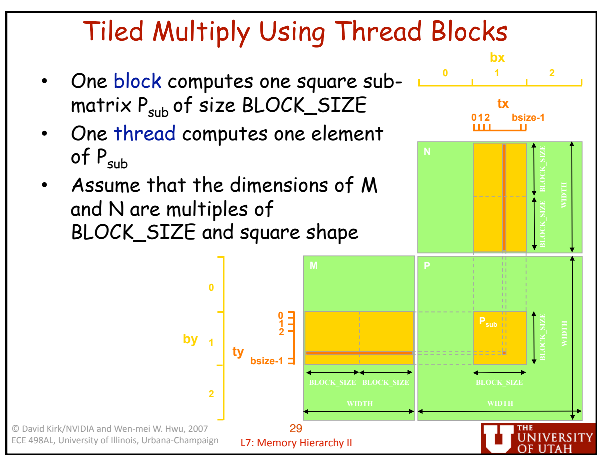
Now C accesses are to named registers.
Compiler guaranteed to map to registers.

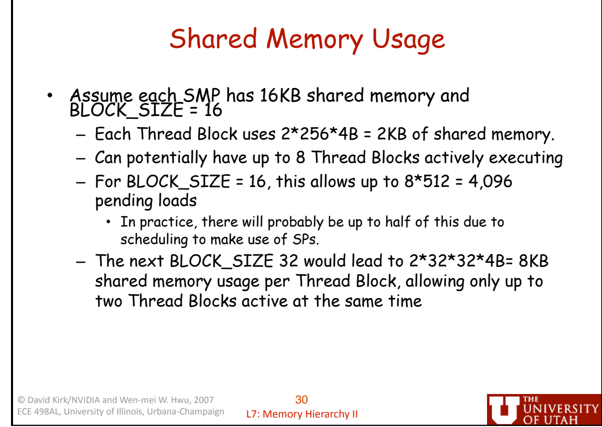
Matrix Multiplication
A Simple Host Version in C

```
// 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;
      }
}
```



© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
ECE498AL, University of Illinois, Urbana-Champaign





First-order Size Considerations

- Each Thread Block should have a minimal of 192 threads
  - BLOCK_SIZE of 16 gives 16*16 = 256 threads
- A minimal of 32 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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

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.

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007
ECE 498AL, University of Illinois, Urbana-Champaign

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 45 GFLOPS