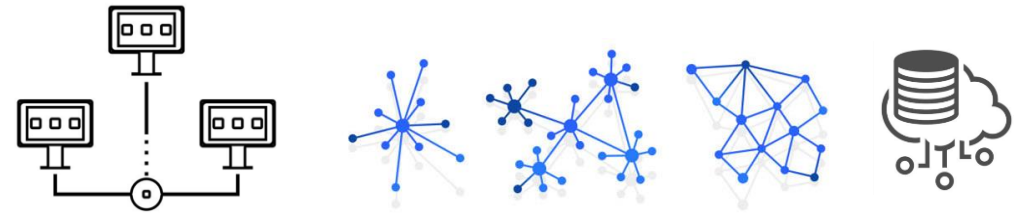


Introduction to Parallel and Distributed Programming

CS 11 and 12
Massive Multi-core Programming

Saikishor Jangiti



Overview of Lecture

Where data can be stored

And how to get it there

Some guidelines for where to store data

Who needs to access it?

Read only vs. Read/Write

Footprint of data

High level description of how to write code to optimize for memory hierarchy

Reference :

Chapter 5, Kirk and Hwu , Programming Massively Parallel Processors A Hands-on Approach, *Third Edition*

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

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

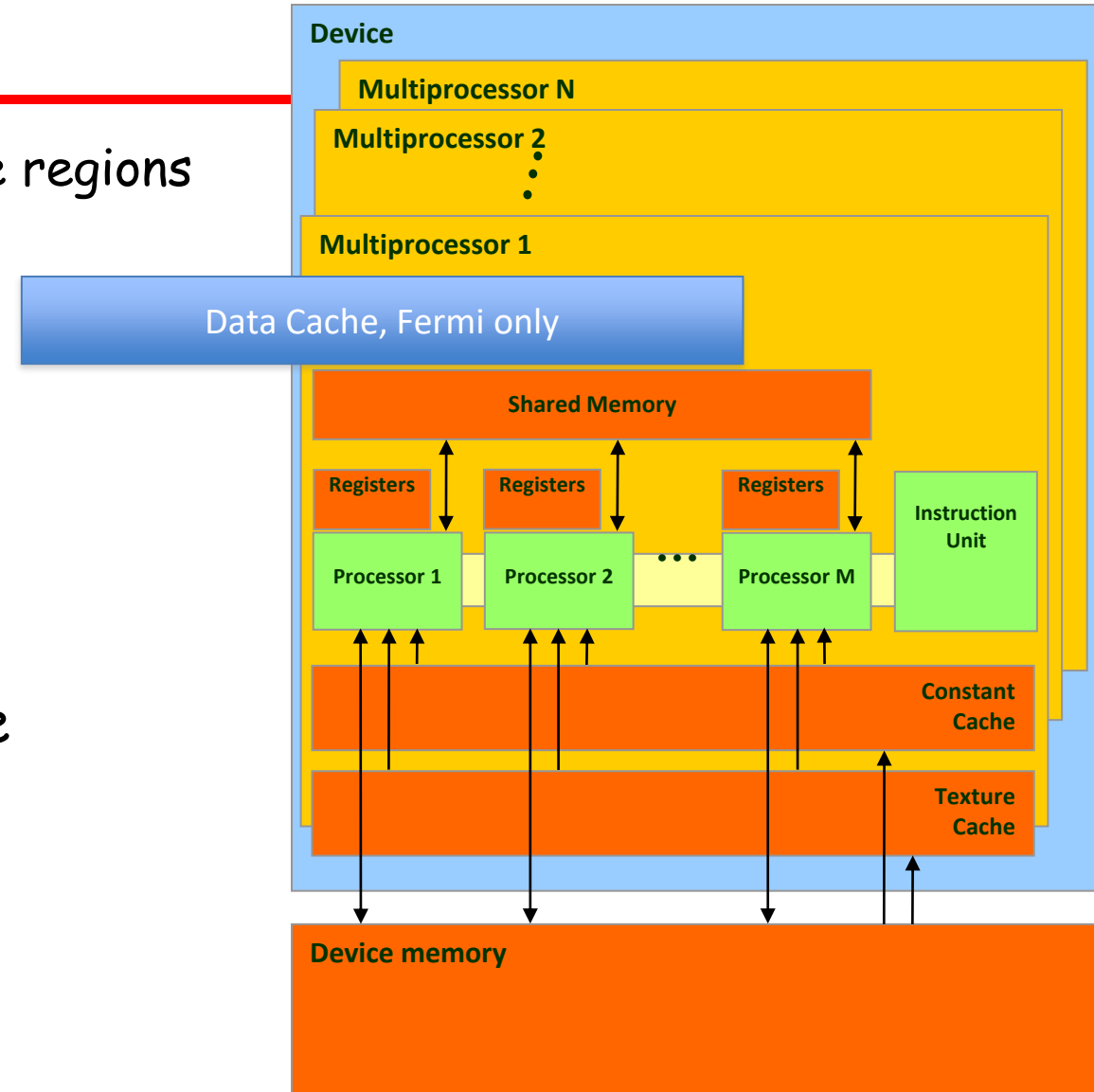
- A read-only **constant cache**


 - 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)**





Memory	Location	Cached	Access	Who
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A - resident	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read	All threads + host

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

Access Times

Register - dedicated HW - single cycle

Constant and Texture caches - possibly single cycle, proportional to addresses accessed by warp

Shared Memory - dedicated HW - single cycle if no "bank conflicts"

Local Memory - DRAM, no cache - *slow*

Global Memory - DRAM, no cache - *slow*

Constant Memory - DRAM, cached, 1...10s...100s of cycles, depending on cache locality

Texture Memory - DRAM, cached, 1...10s...100s of cycles, depending on cache locality

Instruction Memory (invisible) - DRAM, cached

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

Language Extensions: Variable Type Qualifiers

__device__ is optional when used with __local__, __shared__, or __constant__

	Memory	Scope	Lifetime
__device__ __local__ int LocalVar;	local	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

Variable Type Restrictions

Pointers can only point to memory allocated or declared in global memory:

Allocated in the host and passed to the kernel:

```
__global__ void KernelFunc(float* ptr)
```

Obtained as the address of a global variable: `float* ptr = &GlobalVar;`

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

- Data cache (Fermi only)

Now Let's Look at Shared Memory

Common Programming Pattern (5.3 of CUDA 4 manual)

Load data into shared memory

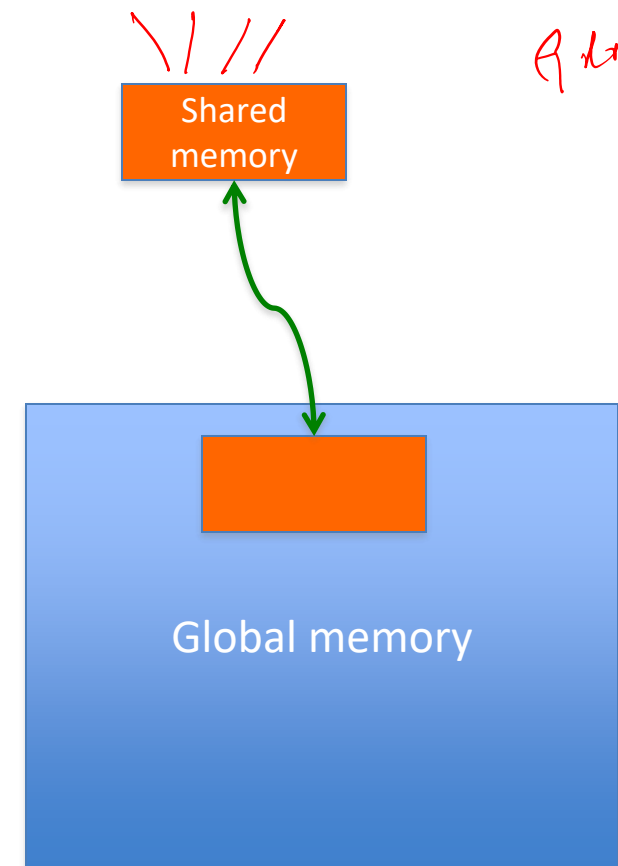
Synchronize (if necessary)

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:

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

    extern __shared__ float d_s_array[]; // create or copy from global memory
    d_s_array[j] = ...;
    /* a form of dynamic allocation */ //synchronize threads before use
    /* MEMSIZE is size of per-block */ __syncthreads();
    /* shared memory*/ ... = d_s_array[x]; // now can use any element

    __host__ void outerCompute() {
        compute<<<gs,bs>>>>(); // more synchronization needed if updated
    }

    __global__ void compute() { // may write result back to global memory
        d_s_array[i] = ...; d_g_array[j] = d_s_array[j];
    }
}
```

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

Temporal Reuse in Sequential Code

Same data used in distinct iterations I and I'

```
for (i=1; i<N; i++)  
    for (j=1; j<N; j++)  
        A[j] = A[j] + A[j+1] + A[j-1]
```

- $A[j]$ has self-temporal reuse in loop i

Spatial Reuse (Ignore for now)

Same data transfer (usually cache line) used in distinct iterations I and I'

```
for (i=1; i<N; i++)  
    for (j=1; j<N; j++)  
        A[j] = A[j] + A[j+1] + A[j-1];
```

- $A[j]$ has self-spatial reuse in loop j
- **Multi-dimensional array note:** C arrays are stored in row-major order



Group Reuse

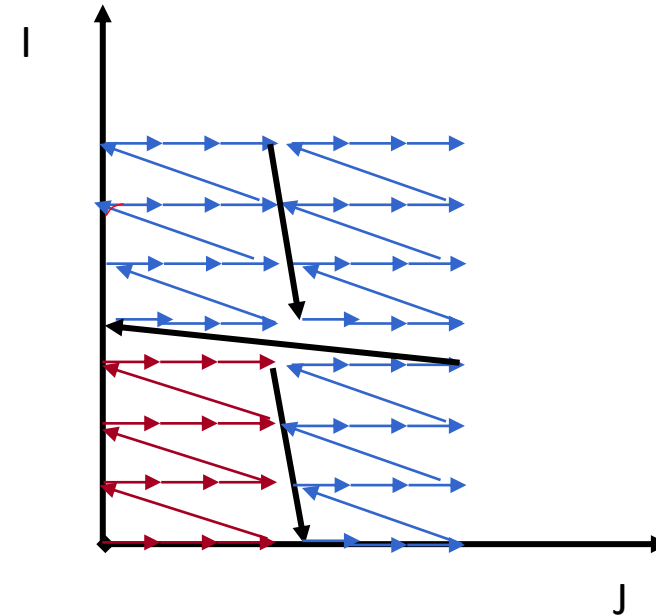
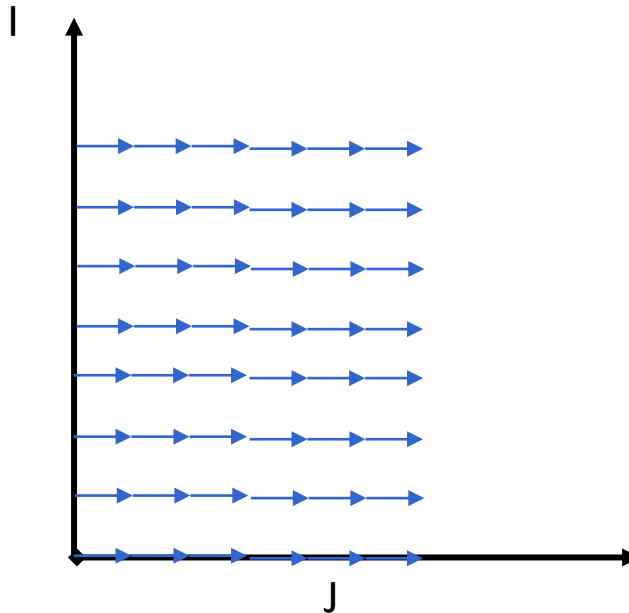
Same data used by distinct references

```
for (i=1; i<N; i++)  
    for (j=1; j<N; j++)  
        A[j] = A[j] + A[j+1] + A[j-1];
```

- $A[j]$, $A[j+1]$ and $A[j-1]$ have group reuse (spatial and temporal) in loop j

Tiling (Blocking): Another Loop Reordering Transformation

Tiling reorders loop iterations to bring iterations that reuse data closer in time



100
10-1
10-2

1 20 | 20 | - - - 1 000

Strip mine

Permute

1

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

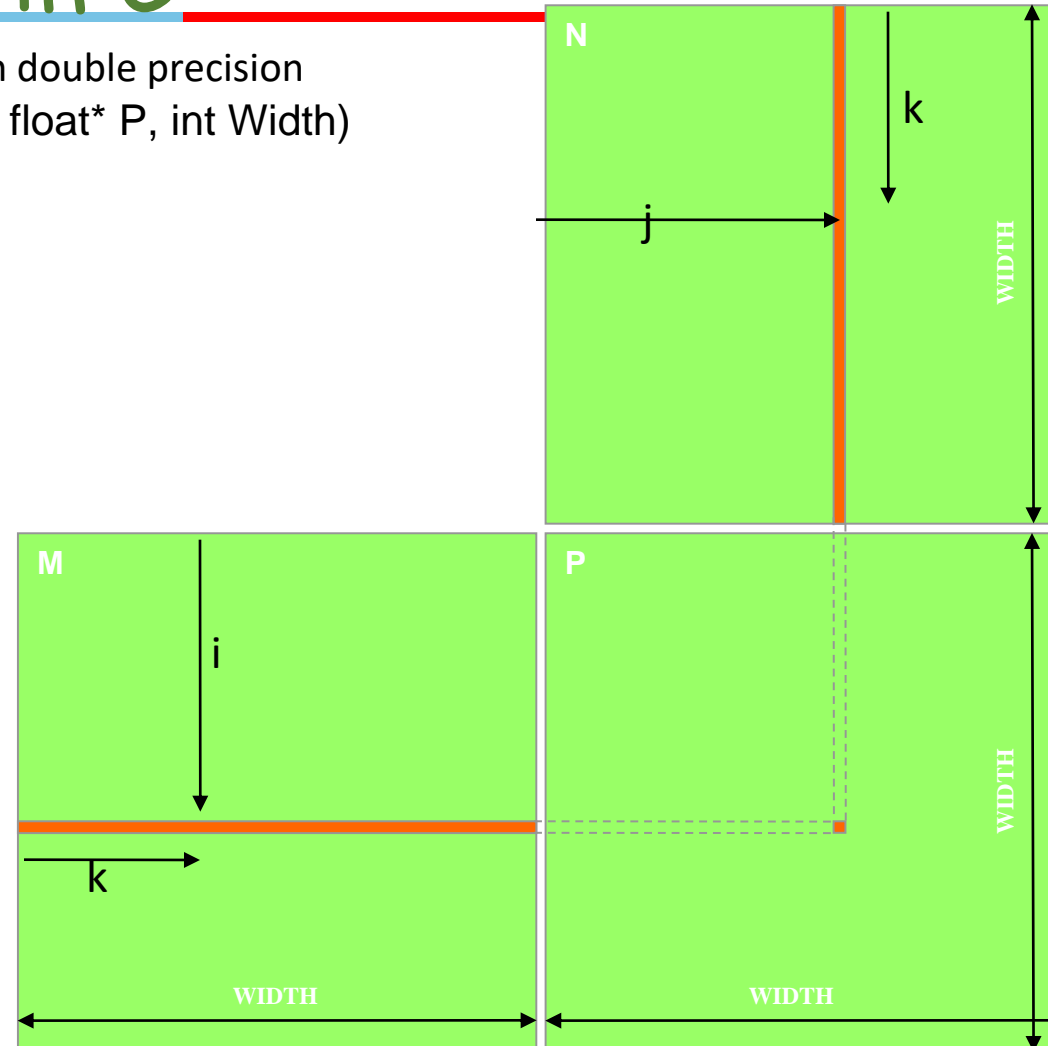
- Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)

- 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;
        }
}
```

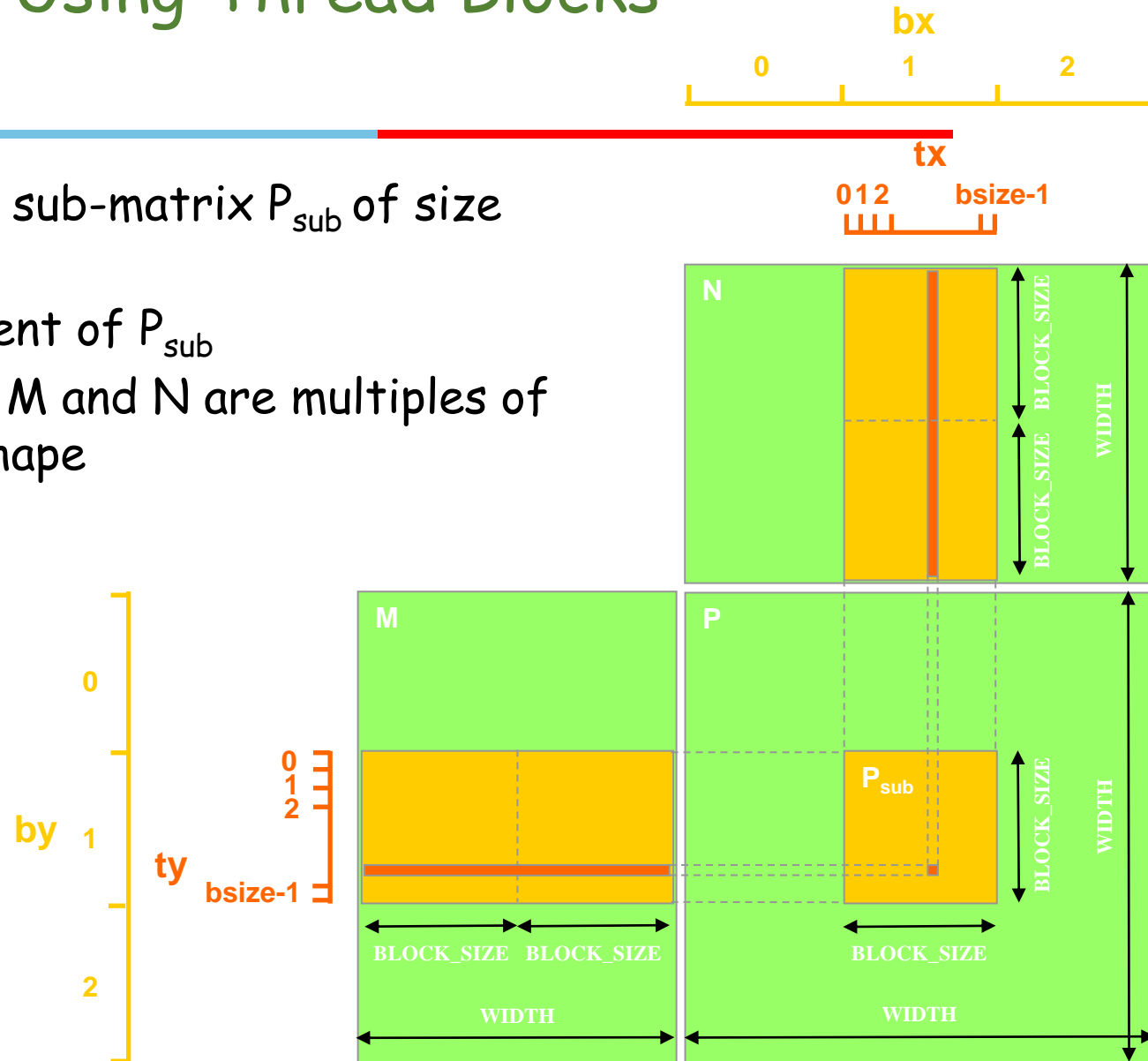


Tiled Matrix Multiply Using Thread Blocks

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



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

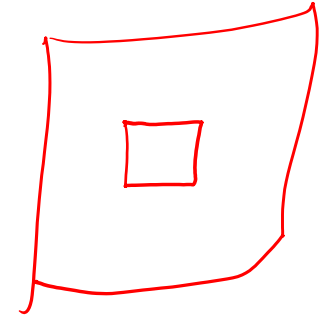
```
// Get a pointer to the current sub-matrix Msub of M
Matrix Msub = GetSubMatrix(M, m, by);

// 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);
```

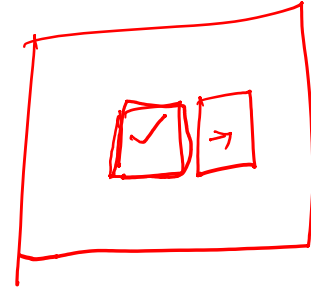


CUDA Code - Compute Result

```
// 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);
```

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

How to place data in constant memory and shared memory

Introduction to Tiling transformation

Matrix multiply example

Recall: Shared Memory

Common Programming Pattern (5.1.2 of CUDA manual)

Load data into shared memory

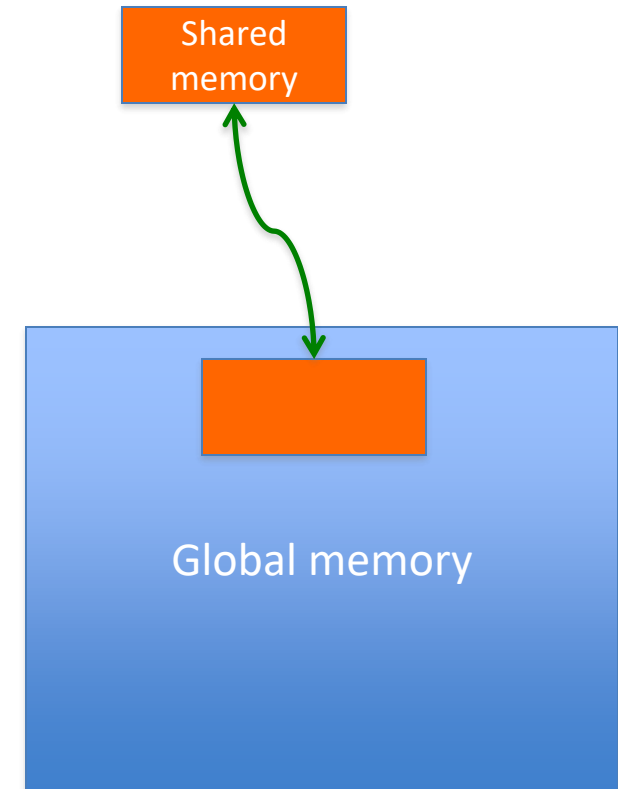
Synchronize (if necessary)

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:

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

    extern __shared__ float d_s_array[]; // create or copy from global memory
    d_s_array[j] = ...;
    /* a form of dynamic allocation */ //synchronize threads before use
    /* MEMSIZE is size of per-block */ __syncthreads();
    /* shared memory*/ ... = d_s_array[x]; // now can use any element

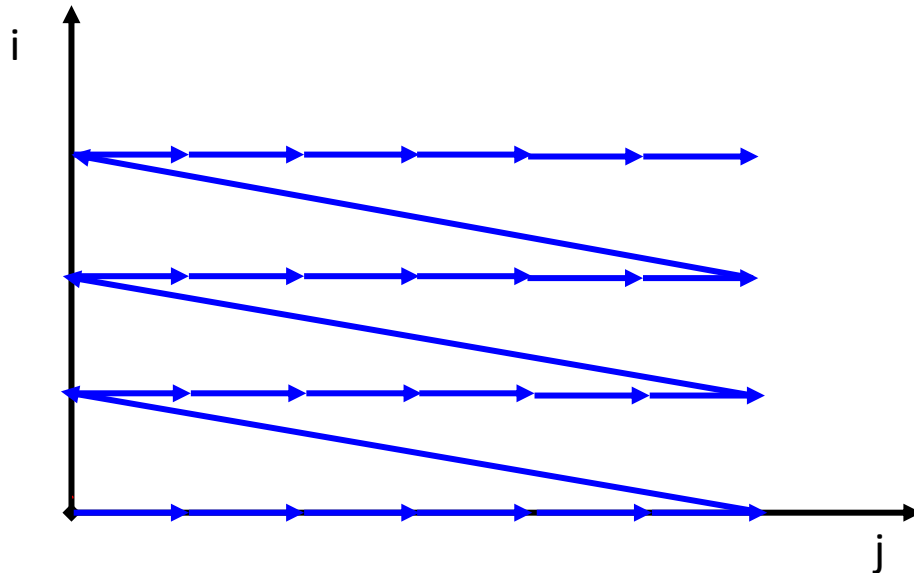
    __host__ void outerCompute() {
        compute<<<gs,bs>>>>(); // more synchronization needed if updated
    }

    __global__ void compute() { // may write result back to global memory
        d_s_array[i] = ...; 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

Ok to permute?

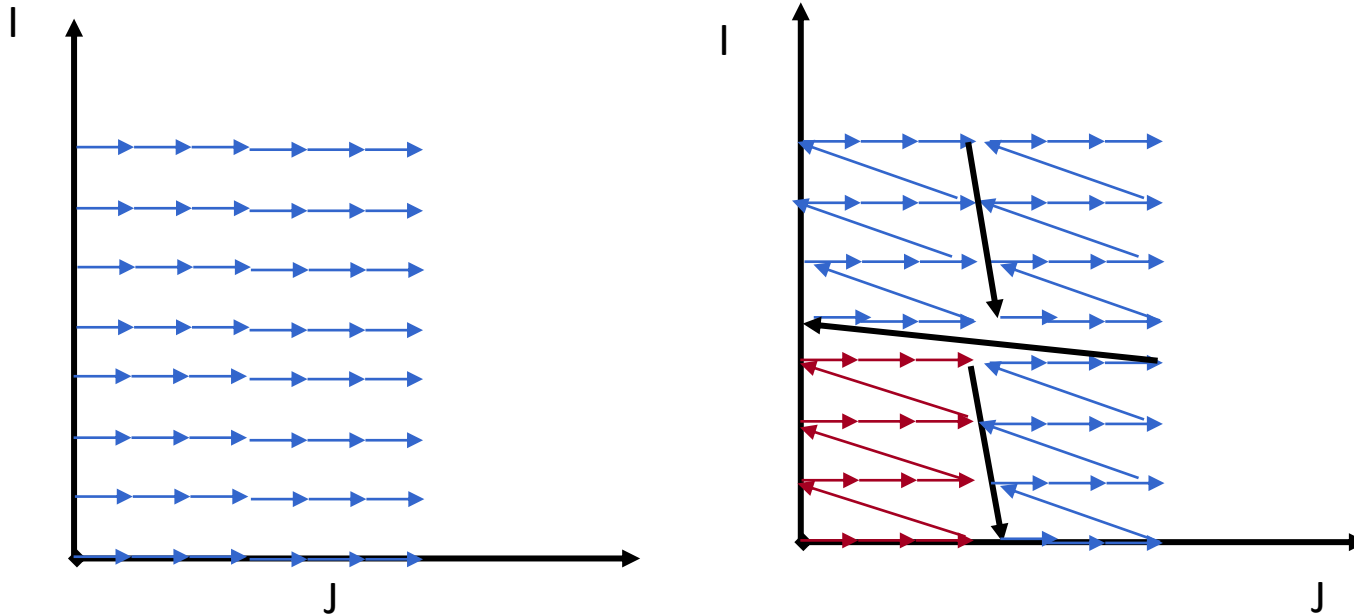
- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so changes the relative order of a read and write or two writes to the same memory location

```
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];
```

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)
  for (i=ii; i<min(ii+s-1,N); i++)
    for (j=1; j<N; j++)
      D[i] = D[i] + B[j][i];
```

← Block dimension
← Thread dimension
← Loop within Thread

...

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

...

```
__global__ Computel (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];
}
```

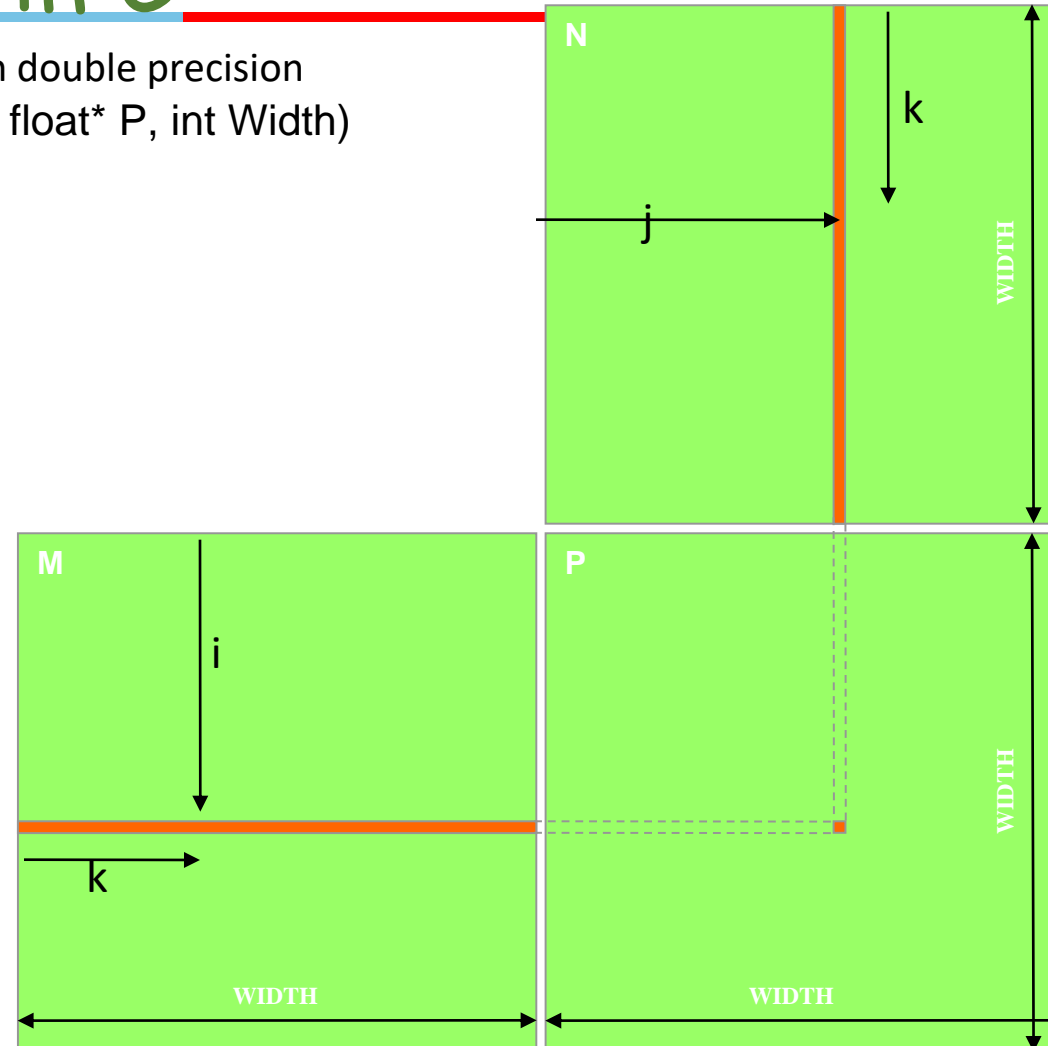
Textbook Shows Tiling for Limited Capacity Shared Memory

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;
        }
}
```

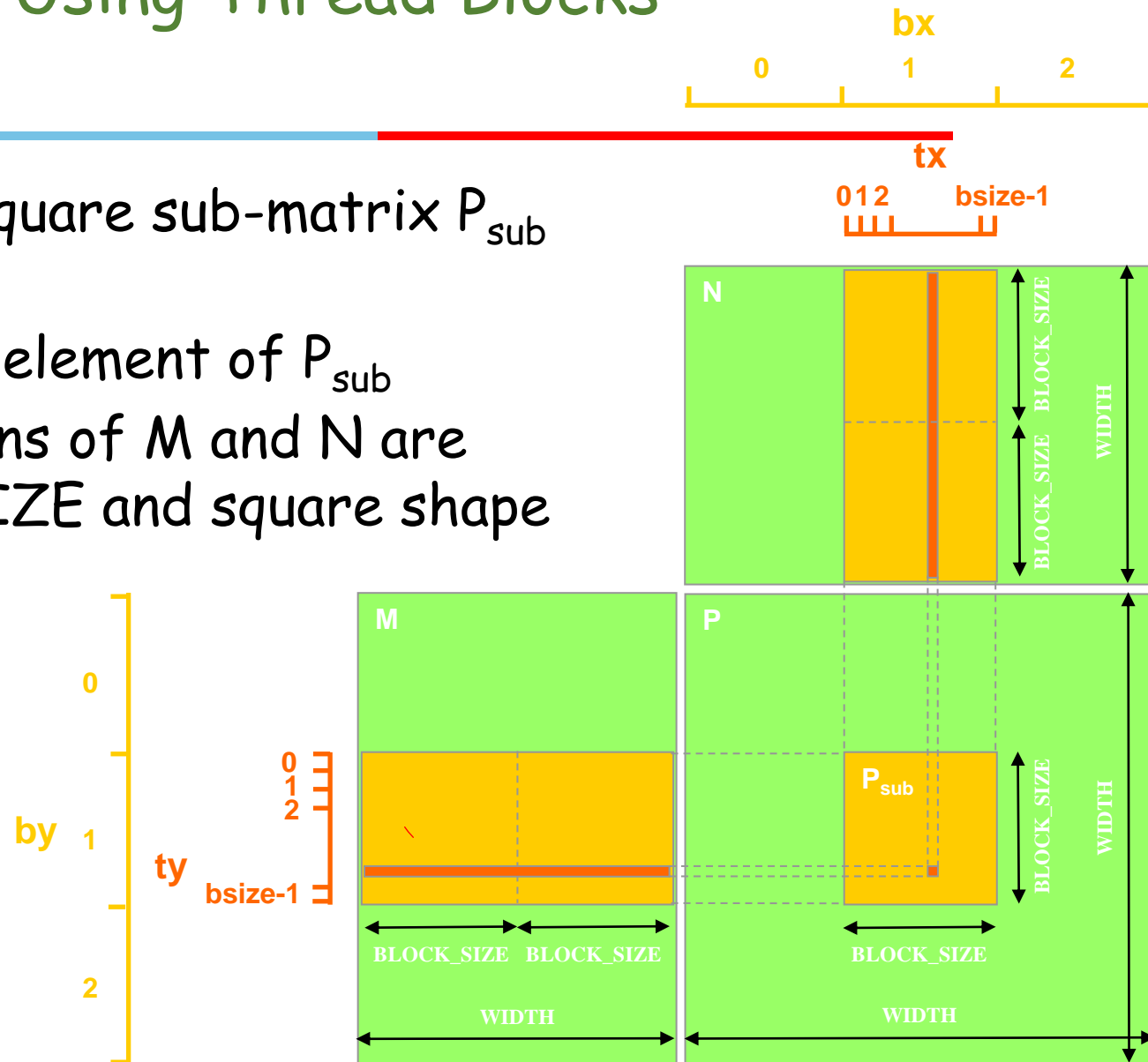


Tiled Matrix Multiply Using Thread Blocks

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)
    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;
    }
```

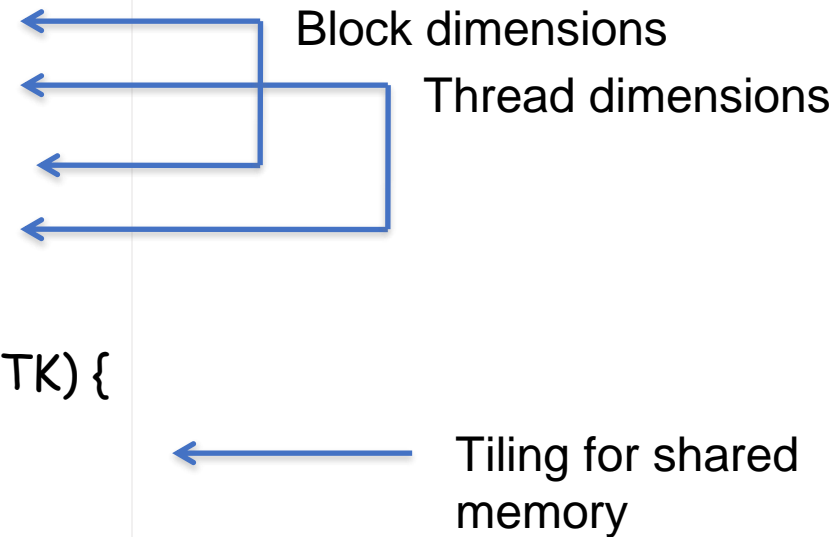
← Tile i

← Tile j

← Tile k (inside thread)

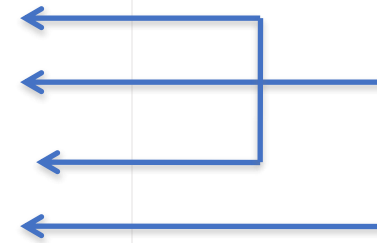
Strip-Mined Code

```
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;
      }
}
```



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

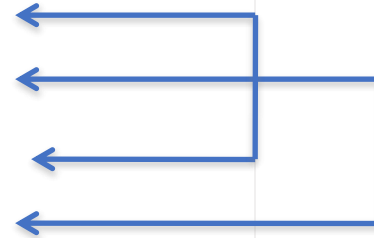


Tiling for shared
memory

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];
        __syncthreads();
    }
    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 $a[i \cdot \text{Row} + j]$

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

$P[\text{by} \cdot \text{TILE_WIDTH} + \text{ty}][\text{bx} \cdot \text{TILE_WIDTH} + \text{tx}]$ or
 $P[\text{Row}][\text{Col}]$

Final Code

```
__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

How to place data in shared memory

Introduction to Tiling transformation

- For computation partitioning

- For limited capacity in shared memory

Matrix multiply example

Matrix Multiplication

```
// function to multiply two matrices
void multiplyMatrices(int first[][10],
                     int second[][10],
                     int result[][10],
                     int r1, int c1, int r2, int c2) {

    // Initializing elements of matrix mult to 0.
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            result[i][j] = 0;
        }
    }

    // Multiplying first and second matrices and storing it in result
    for (int i = 0; i < r1; ++i) {
        for (int j = 0; j < c2; ++j) {
            for (int k = 0; k < c1; ++k) {
                result[i][j] += first[i][k] * second[k][j];
            }
        }
    }
}
```