上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

Lecture 7 CUDA

Zhuoran Song

2023/10/13

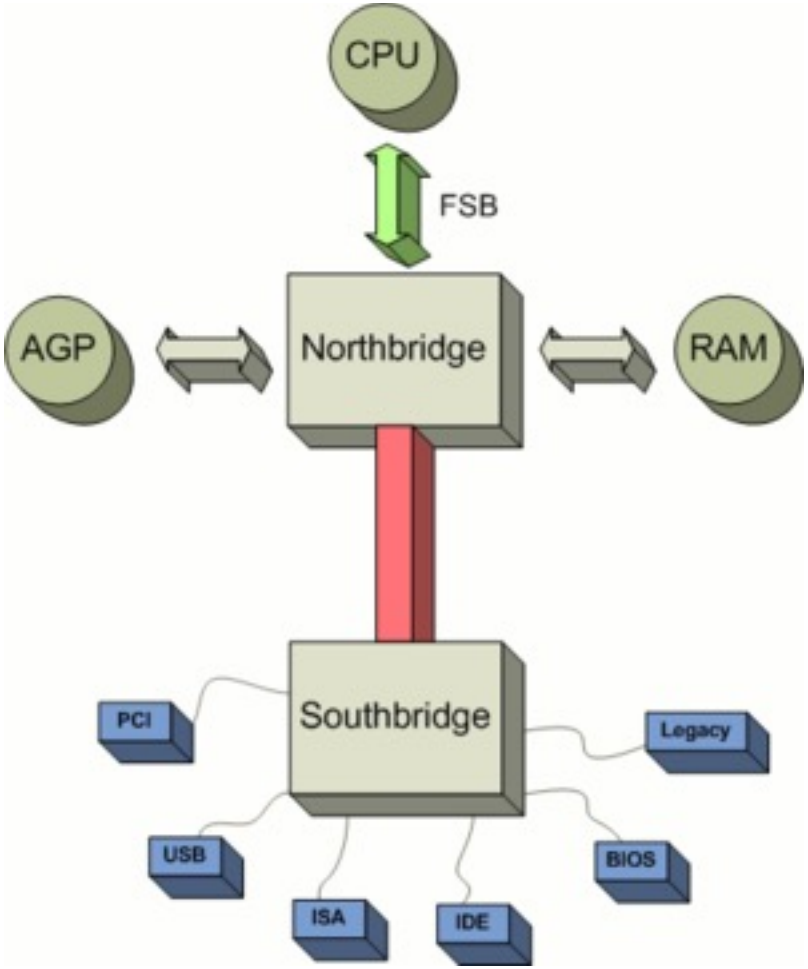
CUDA

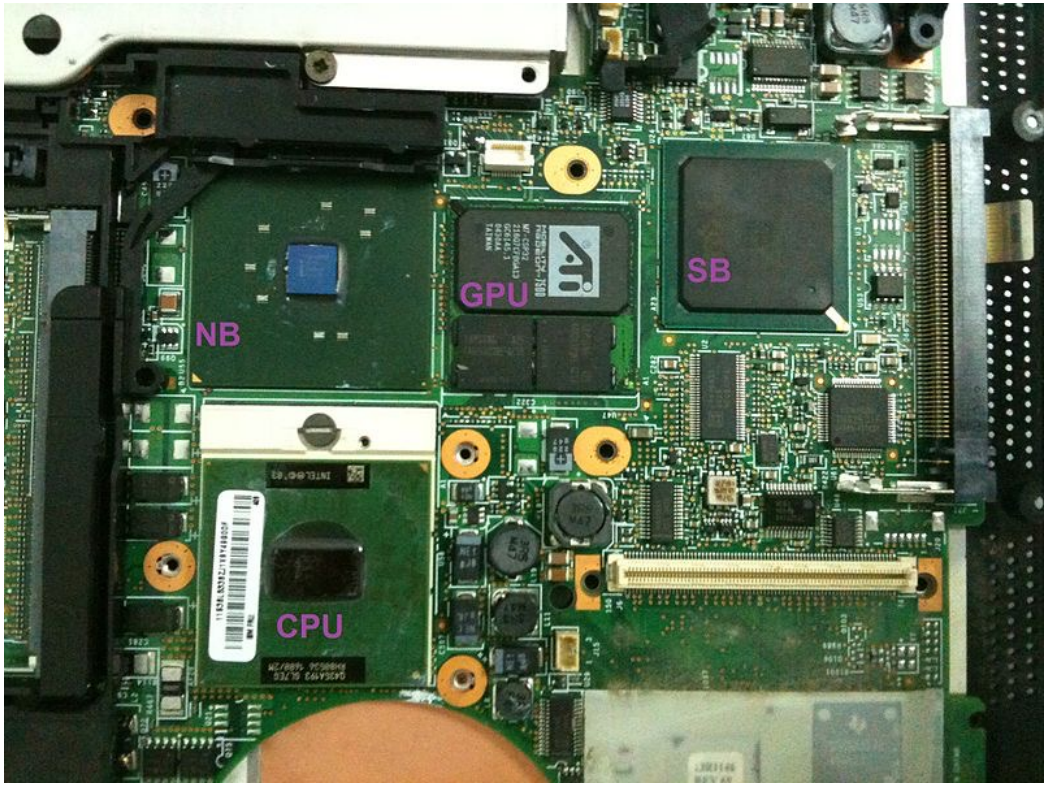
- **"Compute Unified Device Architecture"**
  - *General purpose programming model*
  - *User kicks off batches of threads on the GPU*

- **Targeted software stack**
  - *Compute oriented drivers, language, and tools*

- **Driver for loading computation programs into GPU**
  - *Standalone Driver - Optimized for computation*
  - *Interface designed for compute – graphics-free API*
  - *Data sharing with OpenGL buffer objects*
  - *Guaranteed maximum download & readback speeds*
  - *Explicit GPU memory management*

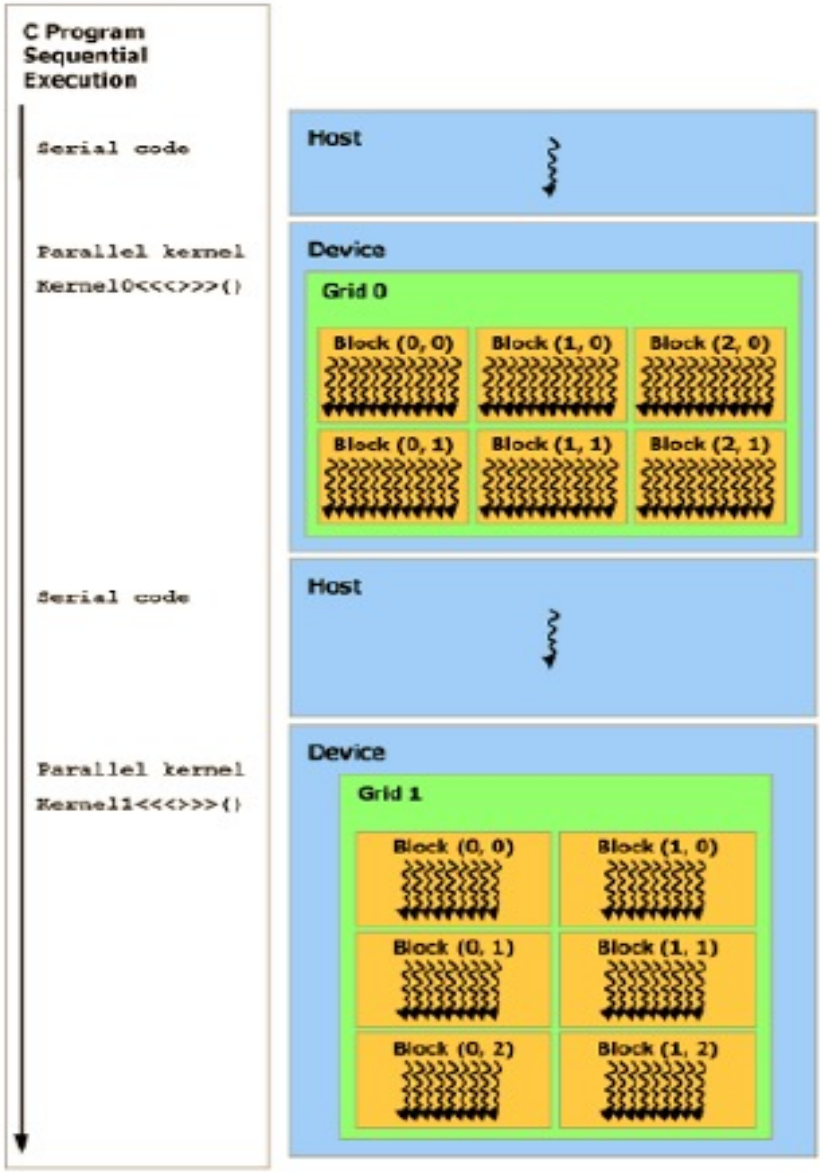
GPU Location





GPU Vs. CPU

CUDA Execution Model



CUDA Device and Threads

- **A compute device**
  - *Is a coprocessor to the CPU or host*
  - *Has its own DRAM (device memory)*
  - *Runs many threads in parallel*

- **Data-parallel portions of an application are expressed as device kernels which run on many threads**

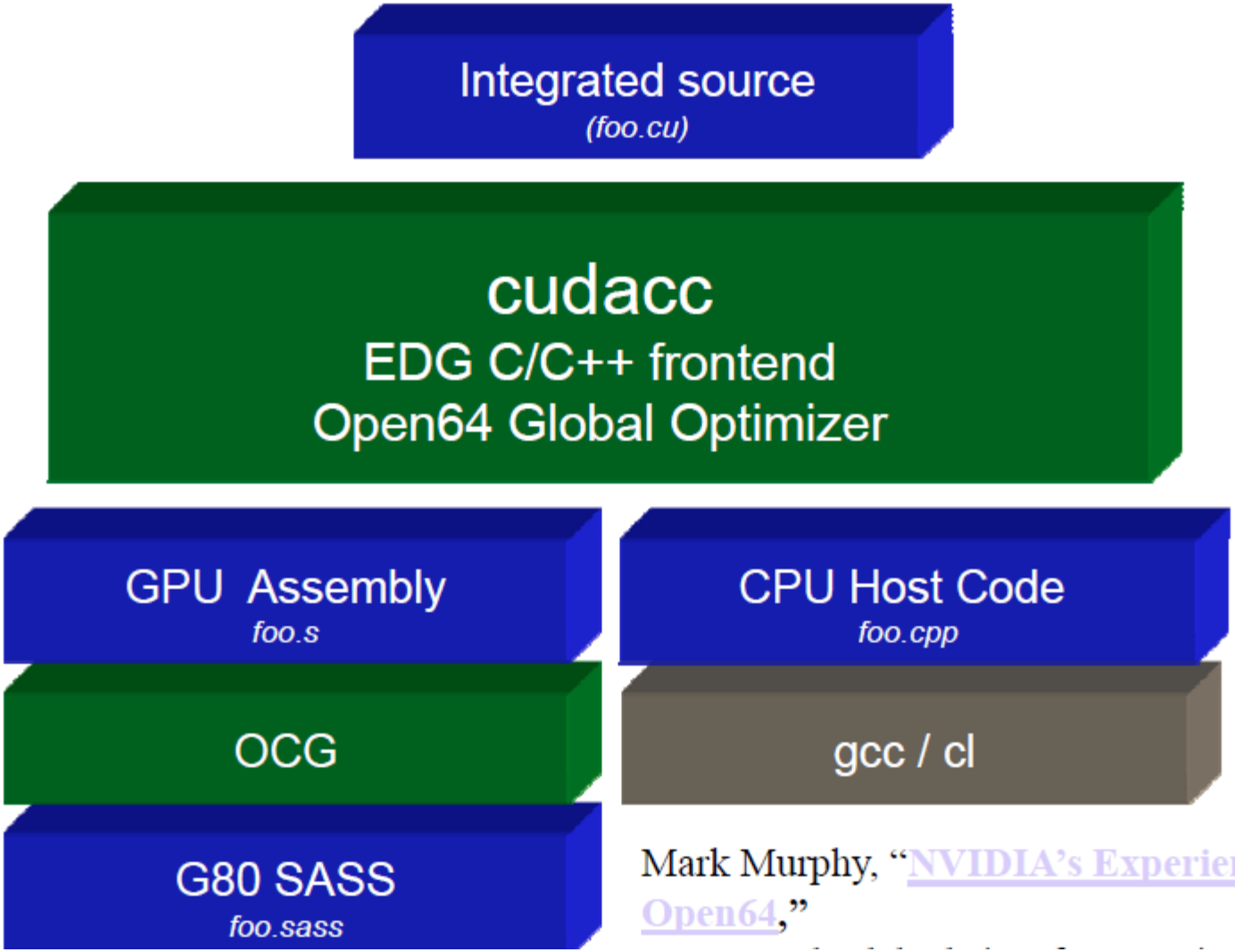
- **Differences between GPU and CPU threads**
  - *GPU threads are extremely light weight*
  - *Very little creation overhead*
  - *GPU needs 1000s of threads for full efficiency*
  - *Multi-core CPU needs only a few*

C Extension

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

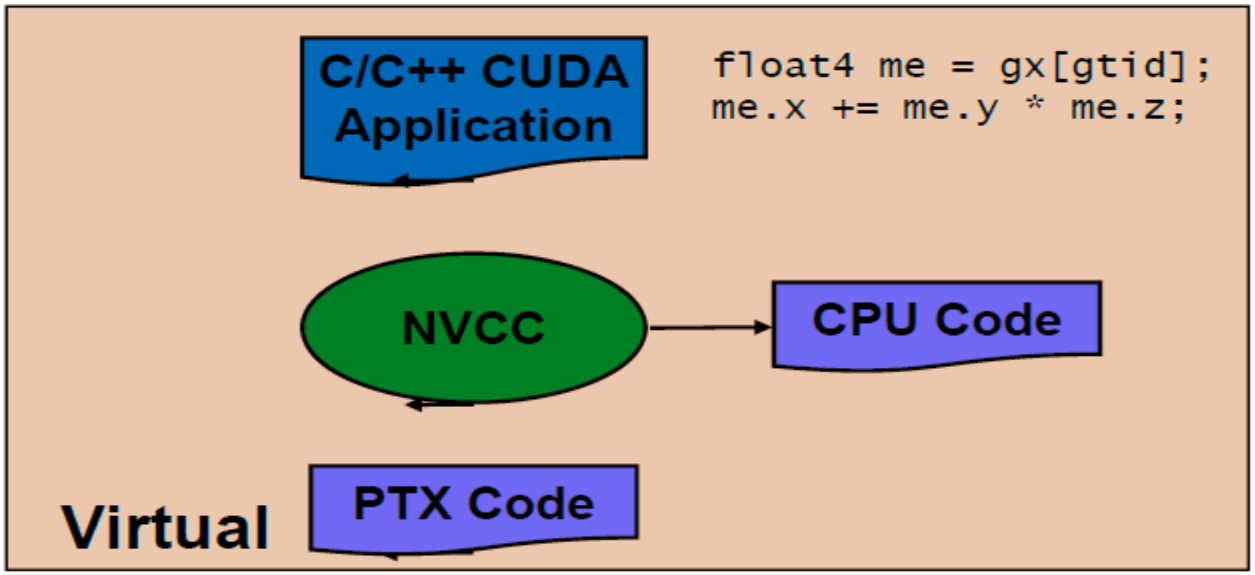
```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

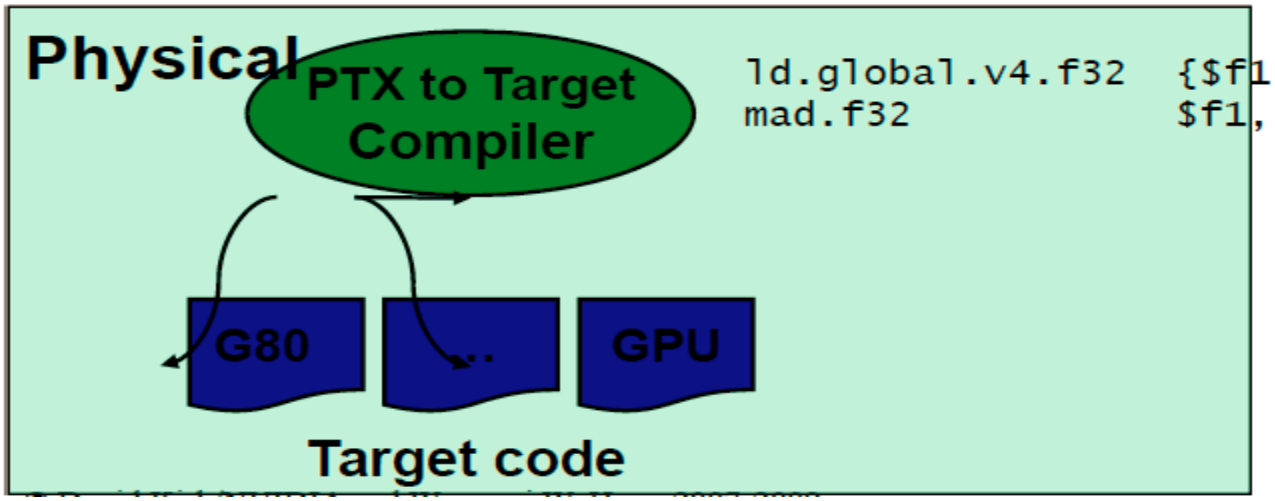
Compilation Flow



Mark Murphy, "NVIDIA's Experience with Open64,"

Compilation Flow





PTX

- PTX：指示+指令的集合，指明该指令进行的操作和需要的操作数

```
tmp0:
    mov.u32     %r11, %ctaid.x
```

- PTX指示

.address_size	.entry	.local	.pragma	.target
.align	.extern	.maxnctapersm	.reg	.tex
.branchtargets	.file	.maxnreg	.reqntid	.version
.callprototype	.func	.maxntid	.section	.visible
.calltargets	.global	.minnctapersm	.shared	.weak
.const	.loc	.param	.sreg	

PTX

- PTX存储指示

名称	说明
.reg	寄存器，访问快速
.sreg	特殊寄存器，只读，需要提前定义，而且不同平台存在差异
.const	常量存储器
.global	全局存储器
.local	局部存储器
.param	用于存储内核函数的参数，需要提前定义
.shared	共享存储器
.tex	纹理存储器

PTX

- PTX采用的数据类型

基础类型	说明
有符号整数	.s8, .s16, .s32, .s64
无符号整数	.u8, .u16, .u32, .u64
浮点数	.f16, .f16x2, .f32, .f64
未定型类型	.b8, .b16, .b32, .b64
掩码	.pred 谓词寄存器

PTX

- PTX的部分指令

abs	cvta	neg	shfl	vabsdiff
add	div	not	shl	vabsdiff2, vabsdiff4
addc	ex2	or	shr	vadd
and	exit	pmevent	sin	vadd2, add4
atom	fma	popc	slct	vavrg2, vavrg4
bar	isspacep	prefetch	sqrt	vmad
bfe	ld	prefetchu	st	vmax
bfi	ldu	prmt	sub	vmax2, vmax4
bfind	lg2	rcp	subc	vmin
bra	mad	red	suld	vmin2, vmin4
brev	mad24	rem	suq	vote
brkpt	madc	ret	sured	vset
call	max	rsqrt	sust	vset2, vset4
clz	membar	sad	testp	vshl
cnot	min	selp	tex	vshr
copysign	mov	set	tld4	vsub
cos	mul	setp	trap	vsub2, vsub4
cvt	mul 24	shf	txq	xor

Loop Transformation

- We will study a few loop transformations that reorder memory accesses to improve locality.

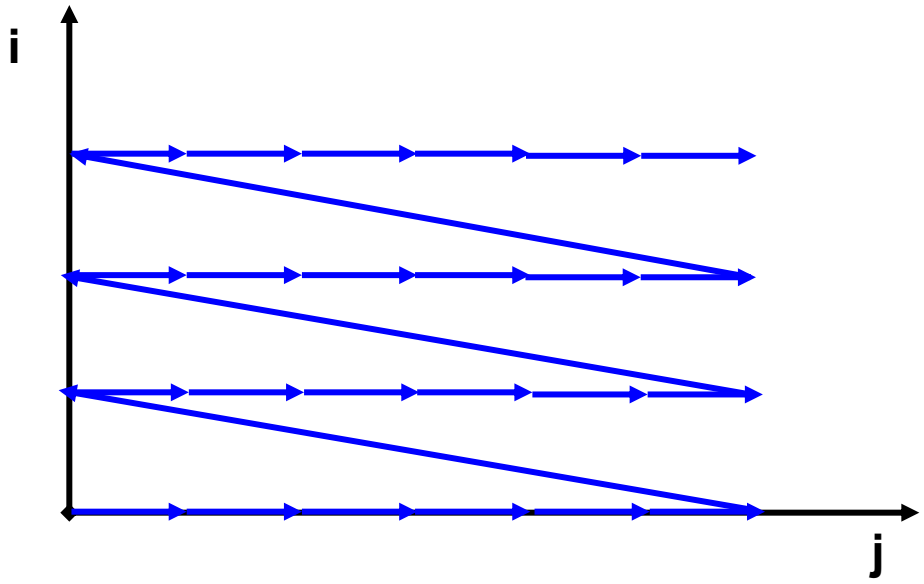
- Two key questions:
  - *Safety: Does the transformation preserve dependences?*
  - *Profitability: Is the transformation likely to be profitable? Will the gain be greater than the overheads (if any) associated with the transformation?*

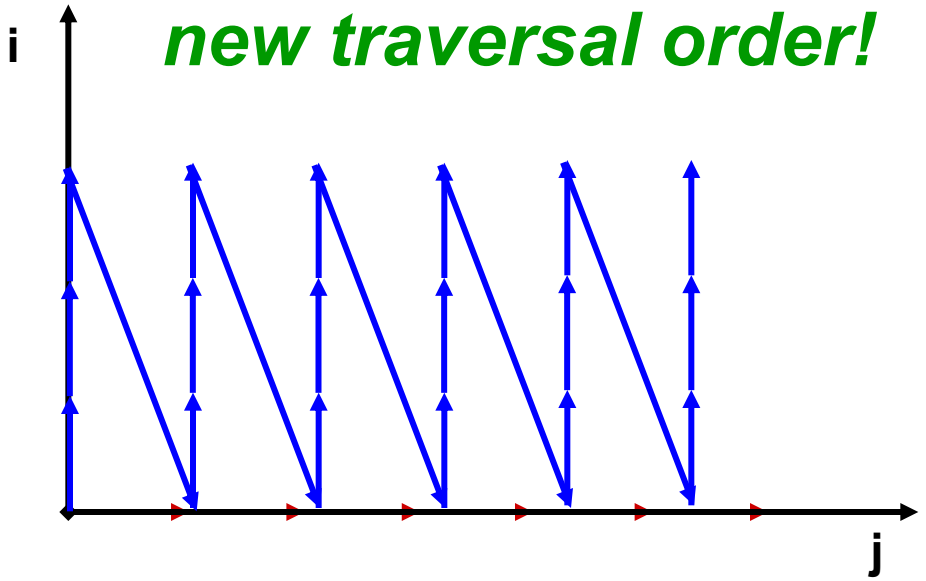
Permutation

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]=A[i][j]+B[j];
```

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

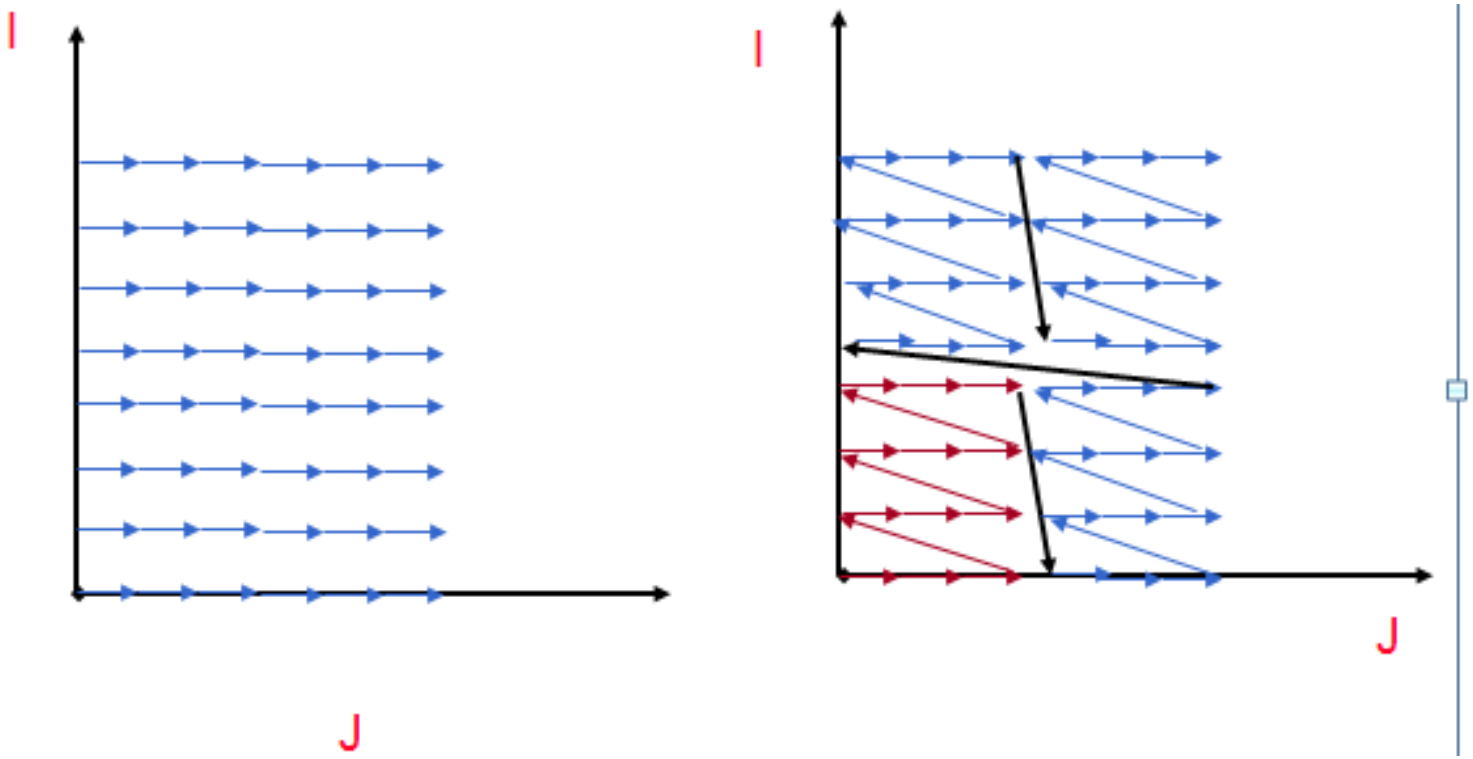




NOTE: C multi-dimensional arrays are stored in row-major order, Fortran in column major

Tiling

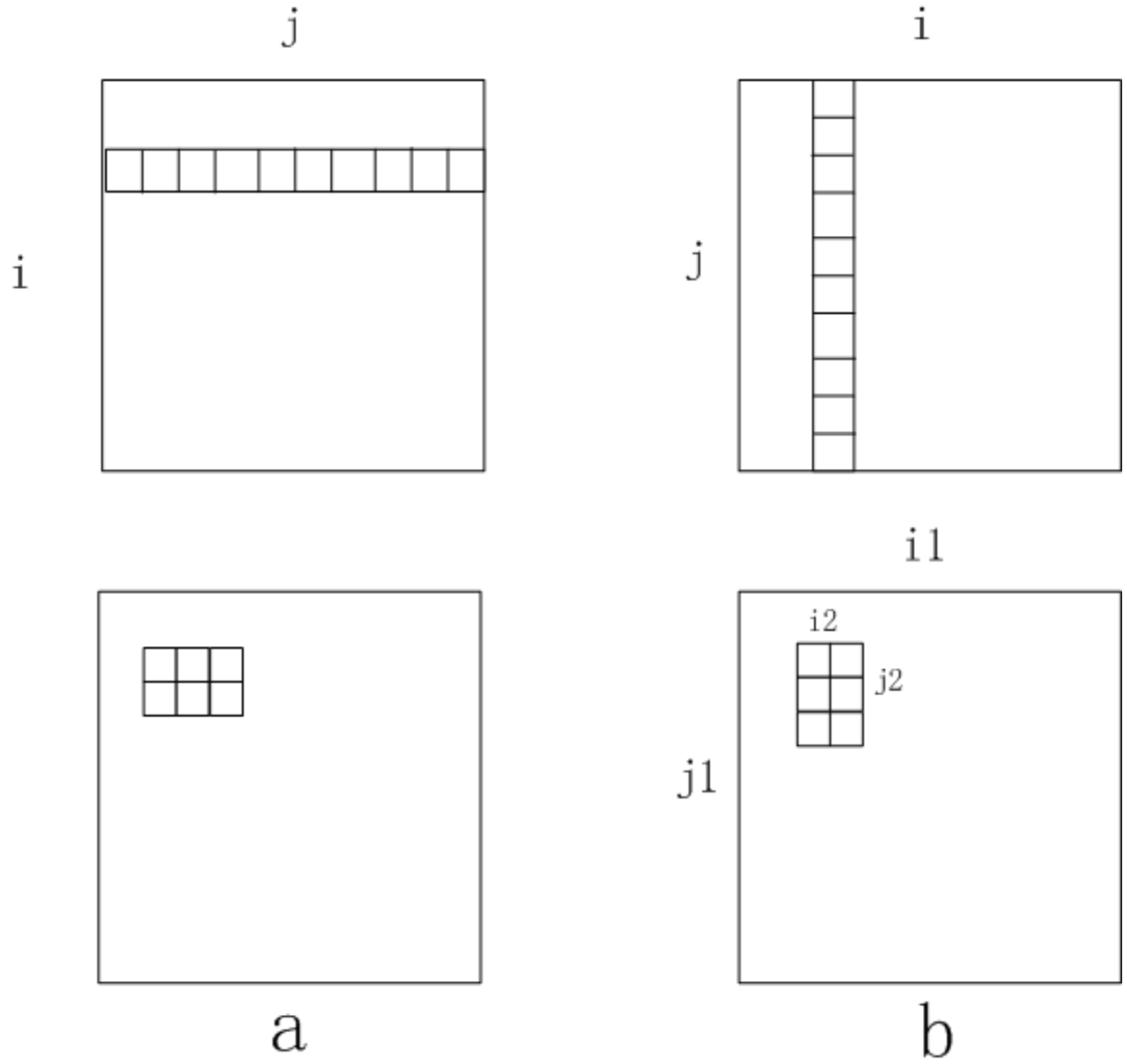
- Tiling reorders loop iterations to bring iterations that reuse data closer in time
- Goal is to retain in cache/register/scratchpad (or other constrained memory structure) between reuse



Tiling

- Tiling is very commonly used to manage limited storage
  - *Registers*
  - *Caches*
  - *Software-managed buffers*
  - *Small main memory*
- Can be applied hierarchically
- Also used in context of managing granularity of parallelism

Tiling



Tiling

```
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]
```

Blocking

```
for (i=0; j<n; j++)
   for (j=0; i<m; j++)
     b[i][j] = a[j,i]
```

```
for (j1=0; j1<n; j1+=nbj)
   for (i1=0; i1<n; i1+=nbi)
      for (j2=0; j2<min(n-j1,nbj); j2++)
         for (i2=0; i2<min(n-i1,nbi); i2++)
            b[i1+i2][j1+j2] = a[j1+j2][i1+i2]
```

Increased cache hit rate and TLB hit rate

Unroll and Jam

- Unroll simply replicates the statements in a loop, with the number of copies called the unroll factor
- As long as the copies don't go past the iterations in the original loop, it is always safe
- Unroll-and-jam involves unrolling an outer loop and fusing together the copies of the inner loop (not always safe)
- One of the most effective optimizations there is, but there is a danger in unrolling too much

```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i];
```

```
Unroll j
for (i=0; i<4; i++)
 for (j=0; j<8; j+=2)
  A[i][j] = B[j+1][i];
  A[i][j+1] = B[j+2][i];
```

```
Unroll-and-jam i
for (i= 0; i<4; i+=2)
 for (j=0; j<8; j++)
   A[i][j] = B[j+1][i];
   A[i+1][j] = B[j+1][i+1];
```

Unroll and Jam

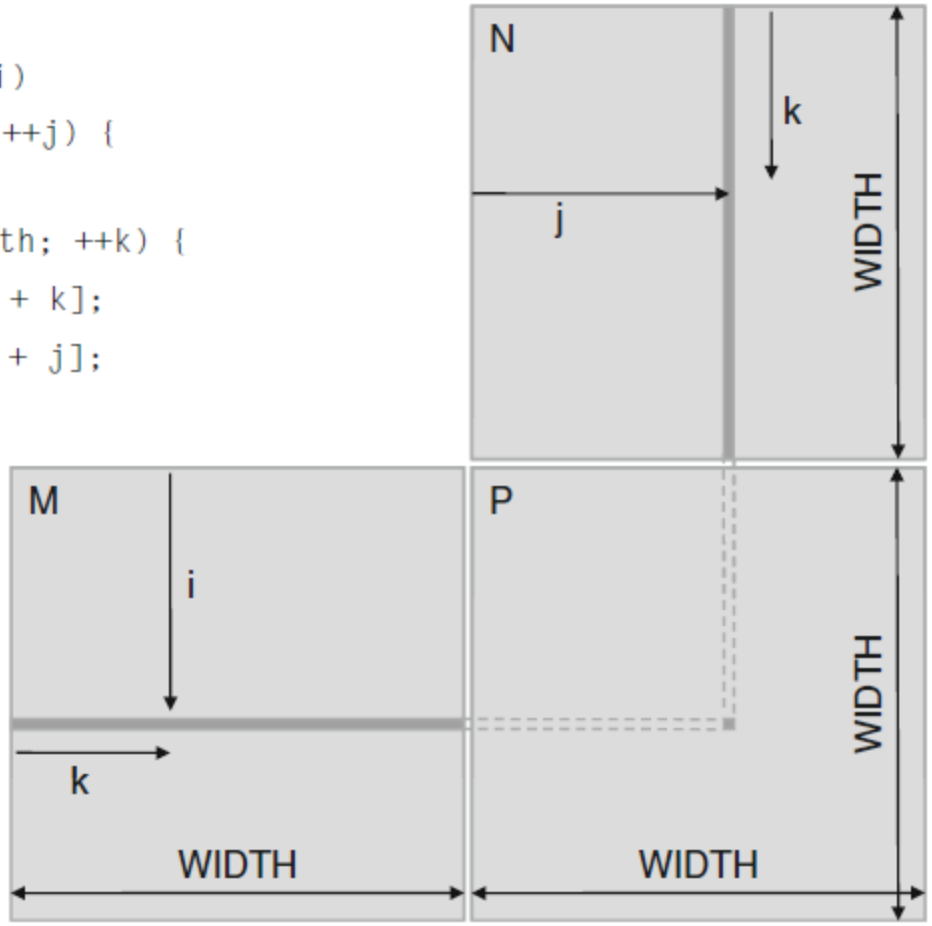
```
Original:
for (i=0; i<4; i++)
 for (j=0; j<8; j++)
  A[i][j] = B[j+1][i] + B[j+1][i+1];
```

```
Unroll-and-jam i and j loops
for (i=0; i<4; i+=2)
 for (j=0; j<8; j+=2) {
  A[i][j]   = B[j+1][i] + B[j+1][i+1];
  A[i+1][j] = B[j+1][i+1] + B[j+1][i+2];
  A[i][j+1] = B[j+2][i] + B[j+2][i+1];
  A[i+1][j+1] B[j+2][i+1] + B[j+2][i+2];
}
```

- Temporal reuse of B in registers
- Less loop control

Matrix Multiplication

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         float sum = 0;
         for (int k  = 0; k < Width; ++k) {
            float a = M[i * width + k];
            float b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```



1000X1000=1,000,000 independent dot product
1000 multiply+1000 accumulate per dot

Matrix Main Program

```
int main(void) {
1.  // Allocate and initialize the matrices M, N, P
    // I/O to read the input matrices M and N
....

2.  // M * N on the device
    MatrixMultiplication(M, N, P, Width);


3.  // I/O to write the output matrix P
    // Free matrices M, N, P
...
return 0;
}
```

Kernel Program

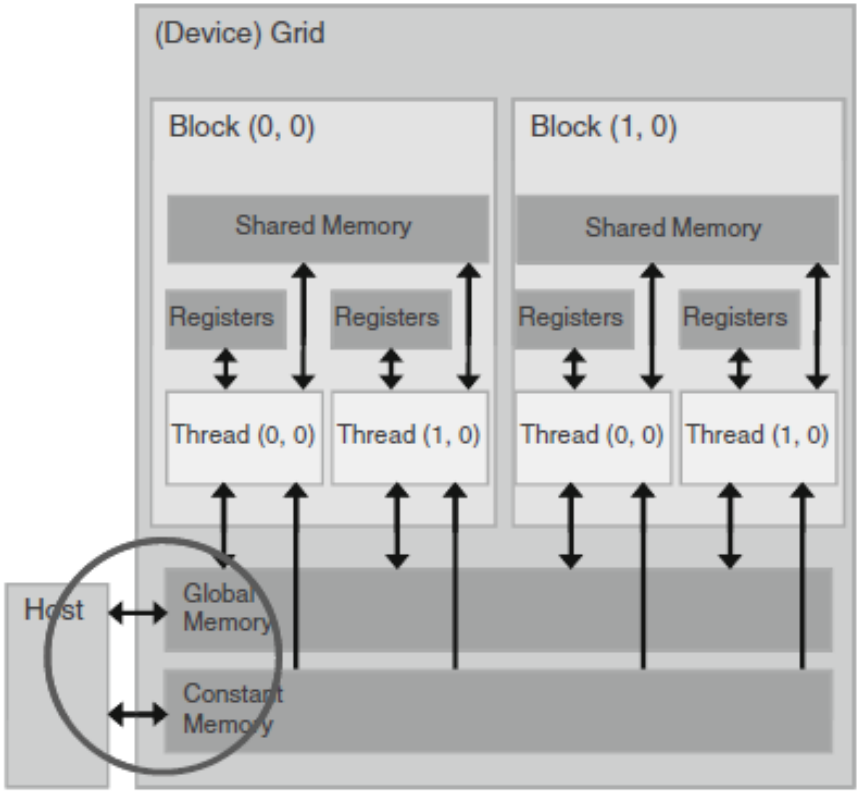
```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
  int size = Width * Width * sizeof(float);
  float* Md, Nd, Pd;
  ...
1. // Allocate device memory for M, N, and P
   // copy M and N to allocated device memory locations

2. // Kernel invocation code - to have the device to perform
   // the actual matrix multiplication

3. // copy P from the device memory
   // Free device matrices
}
```

Creating CUDA Memory Space

```
TILE_WIDTH = 64;
Float* Md
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```

Memory Copy

- cudaMemcpy()
  - **Memory** data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Transfer is asynchronous

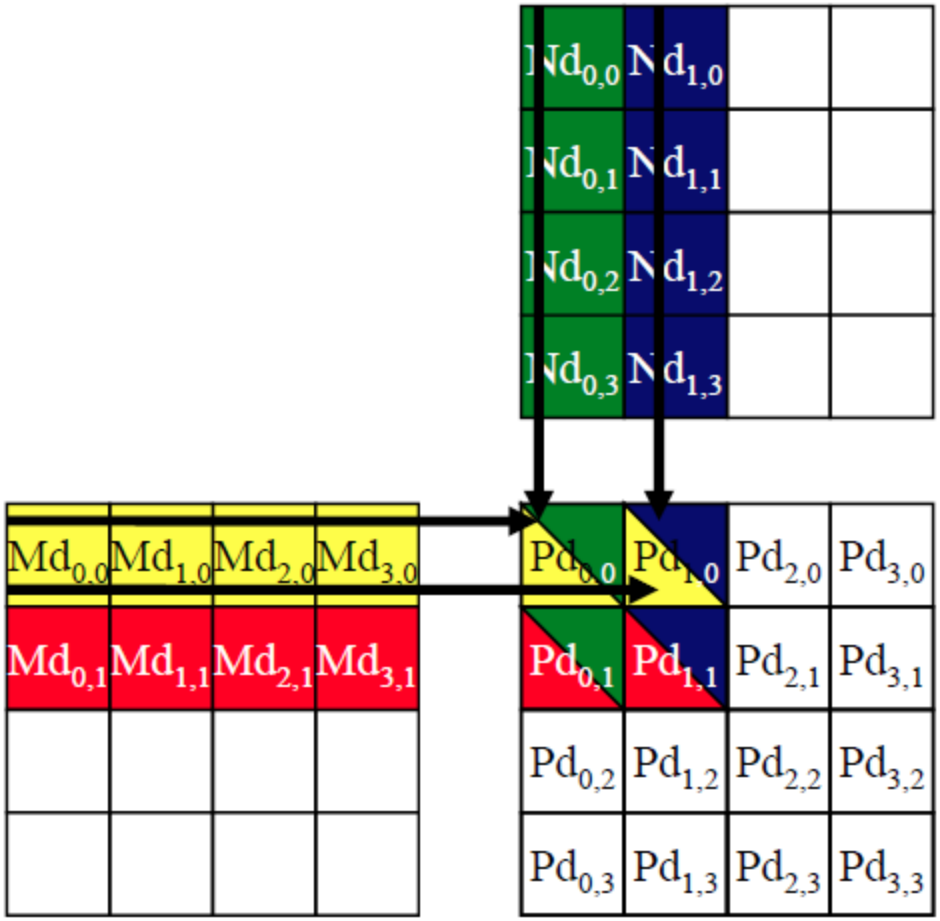



```
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```

Kernel Program

```
void MatrixMultiplication(float* M, float* N, float* P, int Width)
{
   int size = Width * Width * sizeof(float);
   float* Md, Nd, Pd;

1. // Transfer M and N to device memory
   cudaMalloc((void**) &Md, size);
   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMalloc((void**) &Nd, size);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMalloc((void**) &Pd, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer P from device to host
   cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
   // Free device matrices
   cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
   }
```

Calculating a Dot



Kernel Program

```
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
   // 2D Thread ID
   int tx = threadIdx.x;
   int ty = threadIdx.y;

   // Pvalue stores the Pd element that is computed by the thread
   float Pvalue = 0;

   for (int k = 0; k < Width; ++k)
   {
      float Mdelement = Md[ty * Width + k];
      float Ndelement = Nd[k * Width + tx];
      Pvalue += Mdelement * Ndelement;
   }

   // Write the matrix to device memory each thread writes one element
   Pd[ty * Width + tx] = Pvalue;
}
```

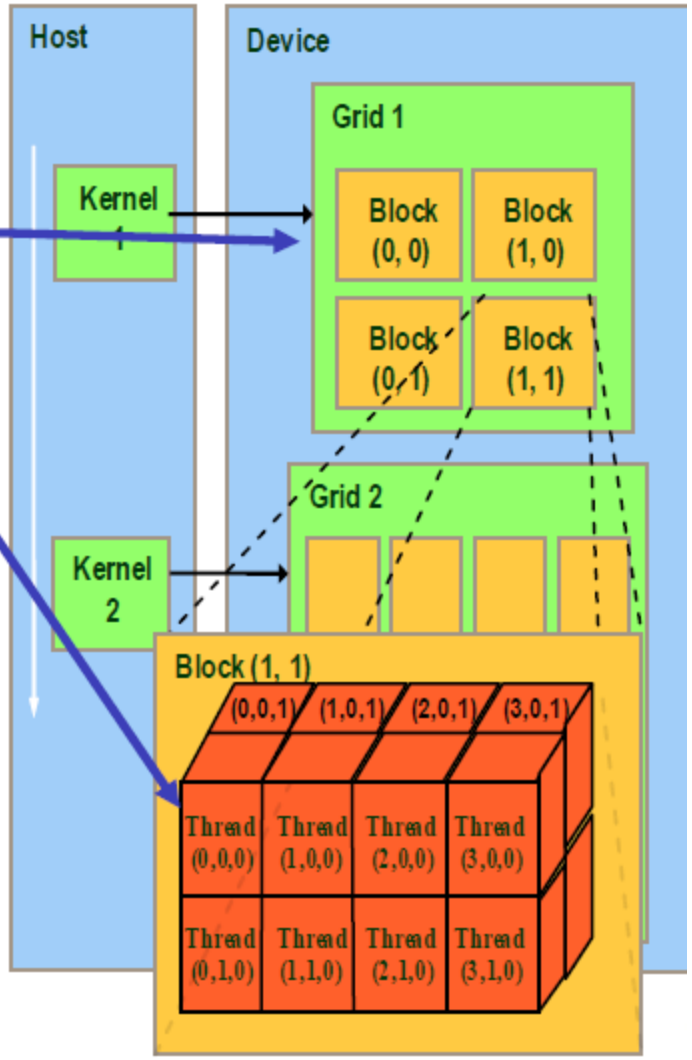
Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together

Thread Blocks

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



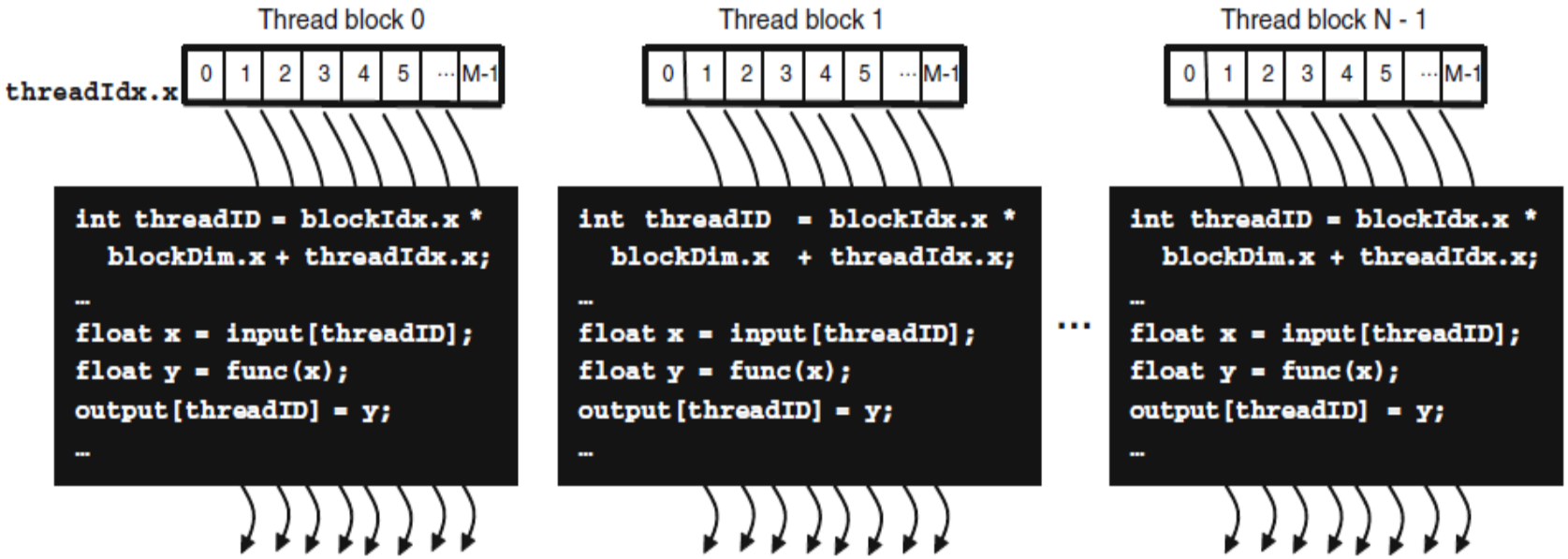
Building Variables

- **dim3 gridDim;**
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- **dim3 blockDim;**
  - Dimensions of the block in threads
- **dim3 blockIdx;**
  - Block index within the grid
- **dim3 threadIdx;**
  - Thread index within the block

Kernel Invocation

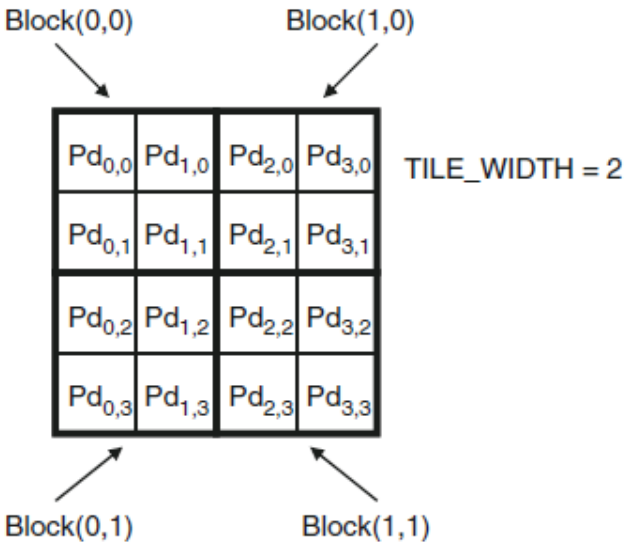
```
// Setup the execution configuration
   dim3 dimGrid(1, 1);
    dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

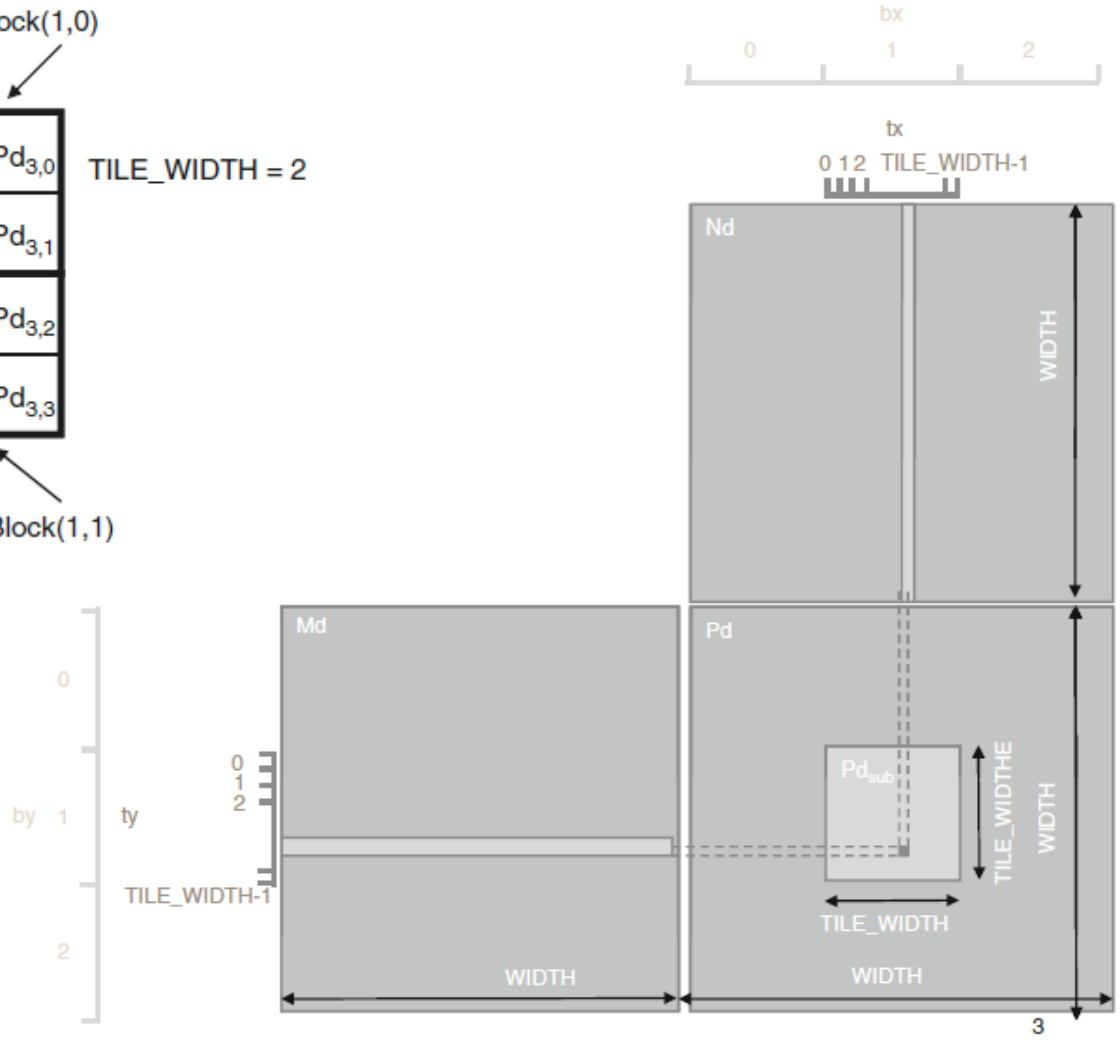
Thread Blocks




```
dim3 dimGrid(128, 1, 1);
dim3 dimBlock(32, 1, 1);
KernelFunction<<<dimGrid, dimBlock>>>(...);
```

Matrix Program





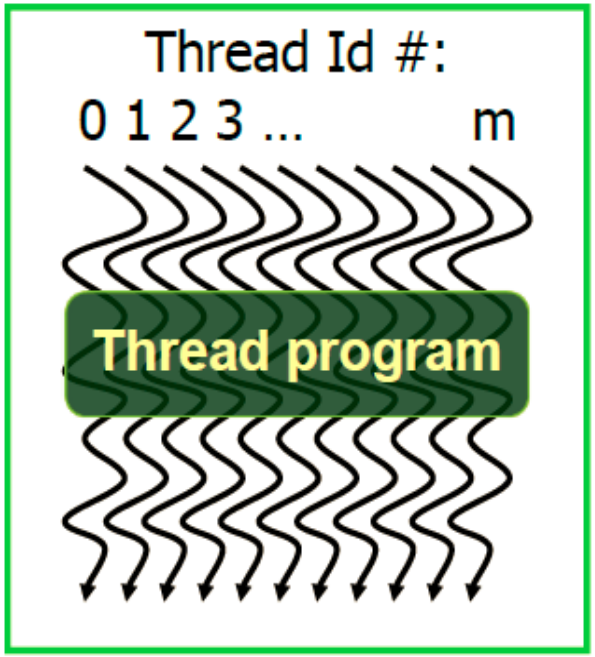
Kernel Program

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the Pd element and M
int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
// Calculate the column idenx of Pd and N
int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

float Pvalue = 0;
// each thread computes one element of the block sub-matrix
for (int k = 0; k < Width; ++k)
  Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

Pd[Row*Width+Col] = Pvalue;
}
```

Characteristics of Thread Blocks

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocs!

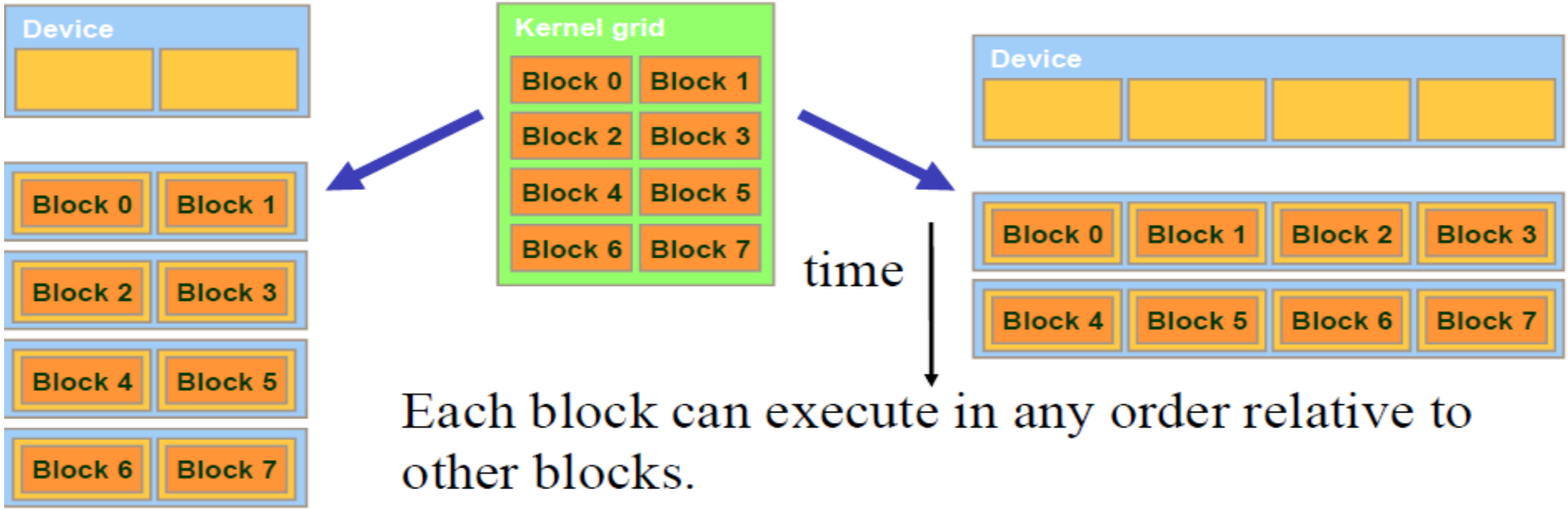
CUDA Thread Block



Courtesy: John Nickolls, NVIDIA

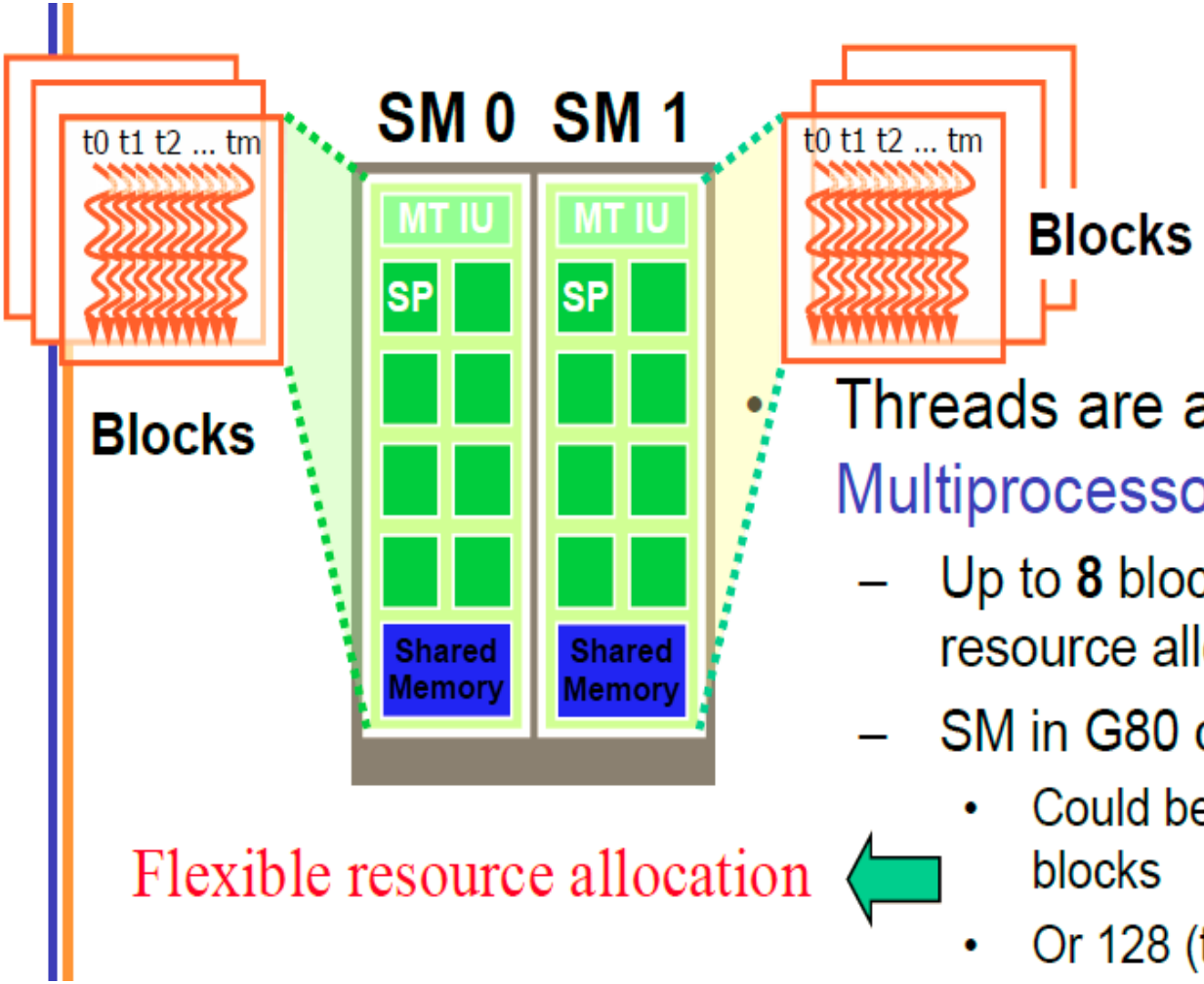
Transparency

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Each block can execute in any order relative to other blocks.

Threads Assignment



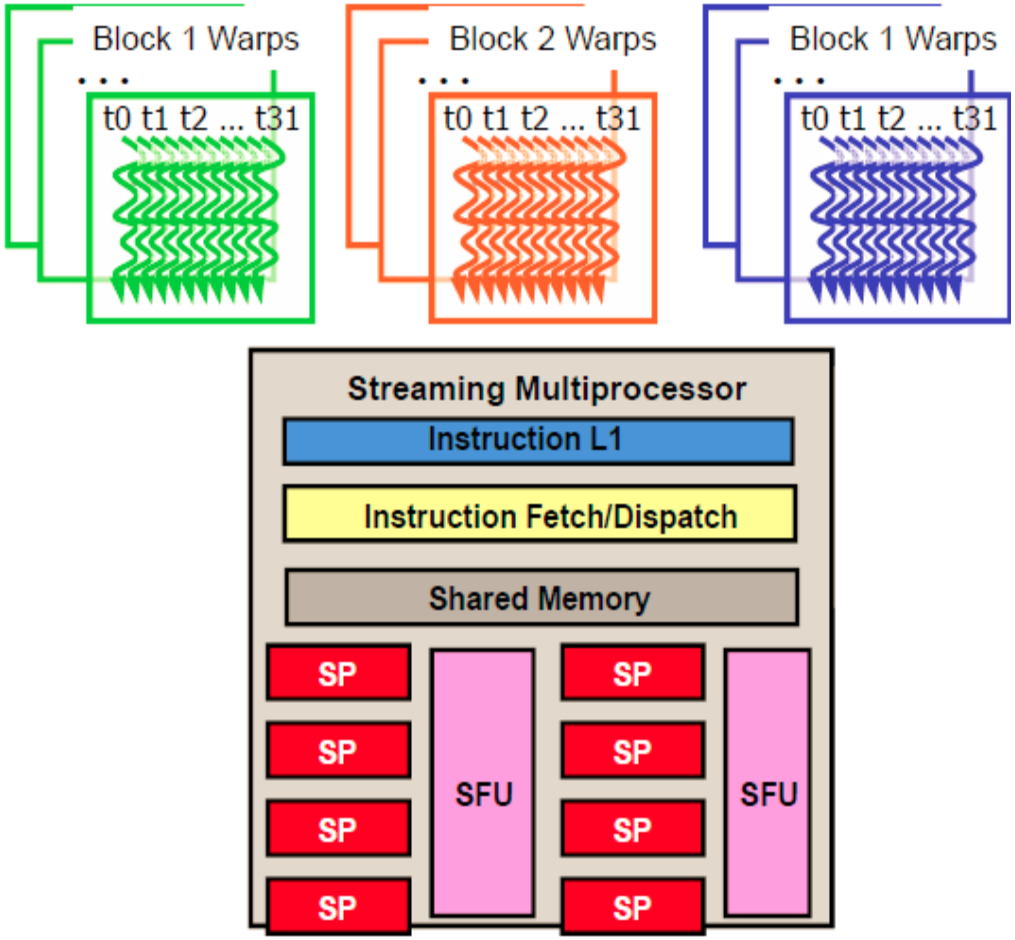
Threads are assigned to Streaming Multiprocessors in block granularity

- Up to **8** blocks to each SM as resource allows
- SM in G80 can take up to **768** threads
  - Could be 256 (threads/block) * 3 blocks
  - Or 128 (threads/block) * 6 blocks, etc.

Flexible resource allocation

Threads Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps



Threads Allocation

•For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 per block?

•For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

•For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.

•For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

Special Functions

- pow, sqrt, cbrt, hypot
- exp, exp2, expm1
- log, log2, log10, log1p
- sin, cos, tan, asin, acos, atan, atan2
- sinh, cosh, tanh, asinh, acosh, atanh
- ceil, floor, trunc, round

Synchronization

- `void __syncthreads();`
- Synchronizes all threads in a block
- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory
- Allowed in conditional constructs only if the conditional is uniform across the entire thread block

Synchronization

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap
- Requires hardware with compute capability 1.1 and above.

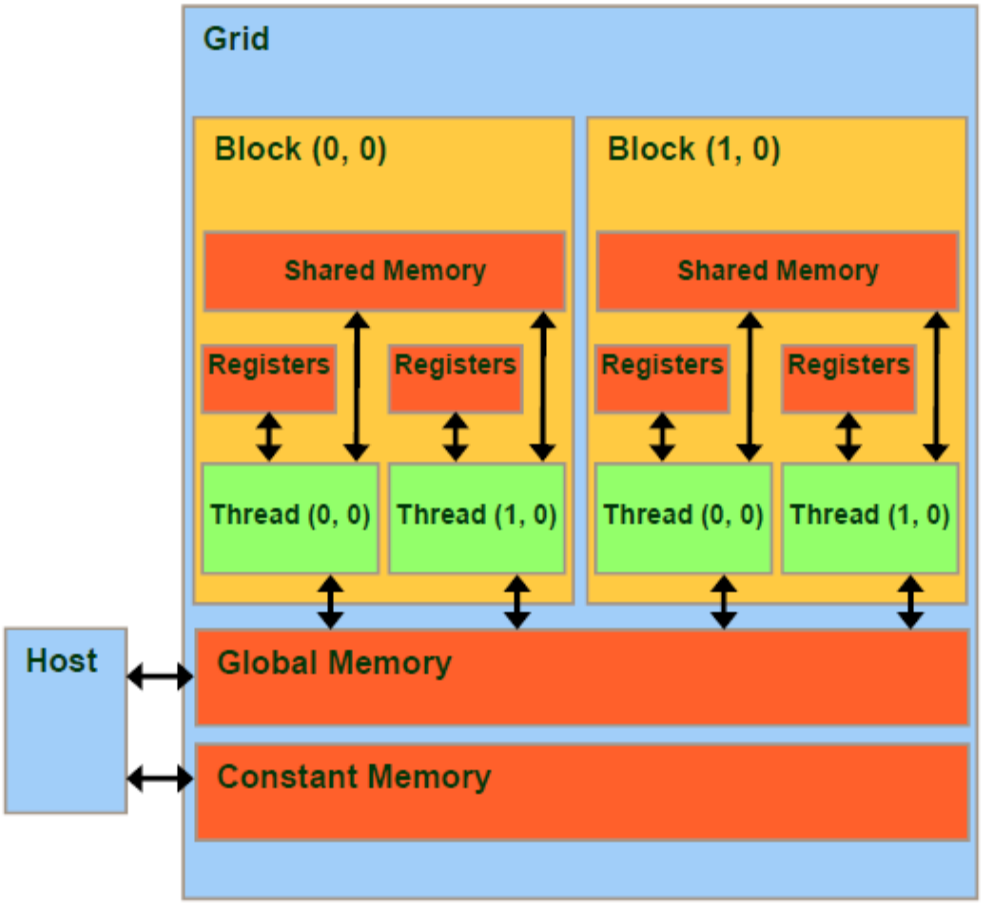
block

Memory Constraints

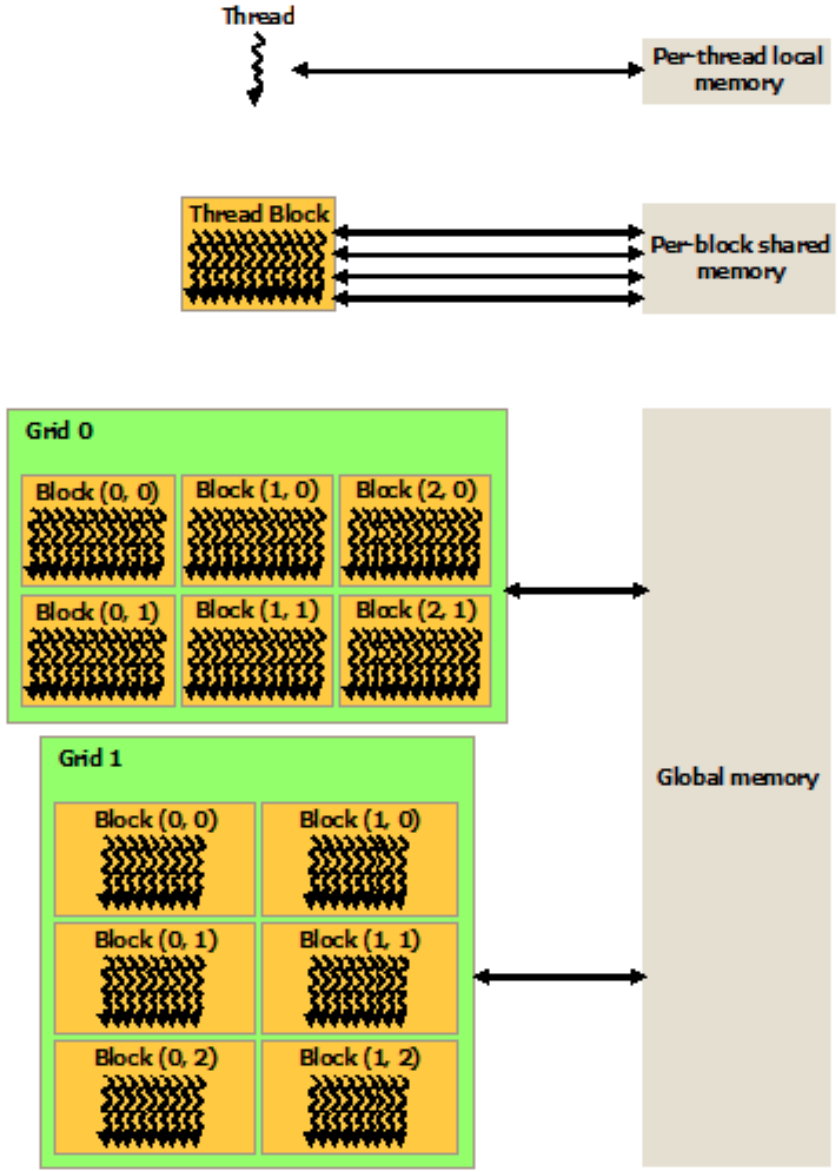
- Compute to Global Memory Access Ratio (CGMA)
  - *Two global memory access required for one multiplication and one addition*
  - *CGMA=1*
- G80 Memory Bandwidth
  - *86.4 GB/s memory bandwidth*
  - *4B per float type*
  - *86.4/4=21.6Gflops/s compute operations*
- G80 Compute Capability
  - *367Gflops/s*
  - *21.6/367=5.8% potential is used*

Memory Types

- Each thread can:
  - Read/write per-thread **registers**
  - Read/write per-thread local memory
  - Read/write per-block **shared memory**
  - Read/write per-grid **global memory**
  - Read/only per-grid **constant memory**



Memory Types



Memory Declaration

Variable declaration	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

- `__device__` is optional when used with `__local__`, `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in local memory

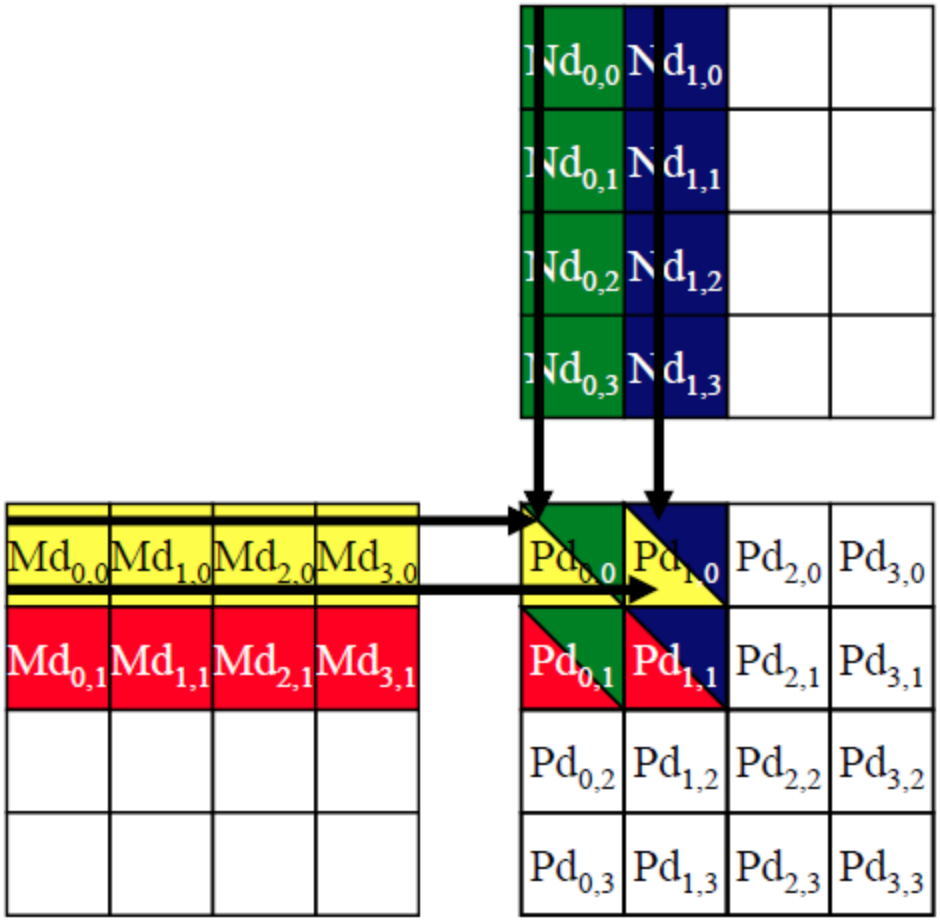
Memory Strategy

- Global memory resides in device memory (DRAM) - much slower access than shared memory
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

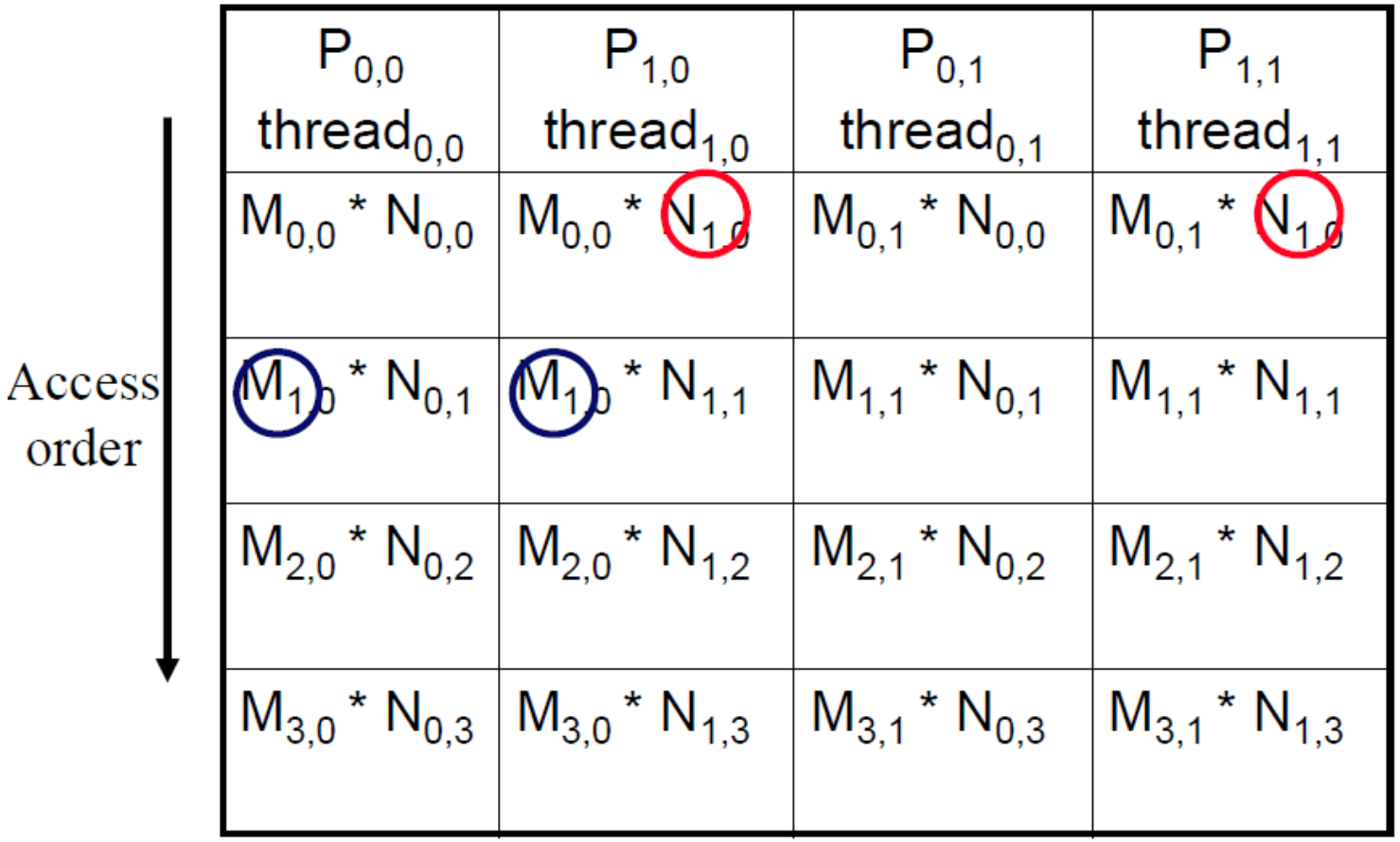
Memory Strategy

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data
- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very fast)
  - R/W inputs/results → global memory (very slow)

Shared Data in Matrix

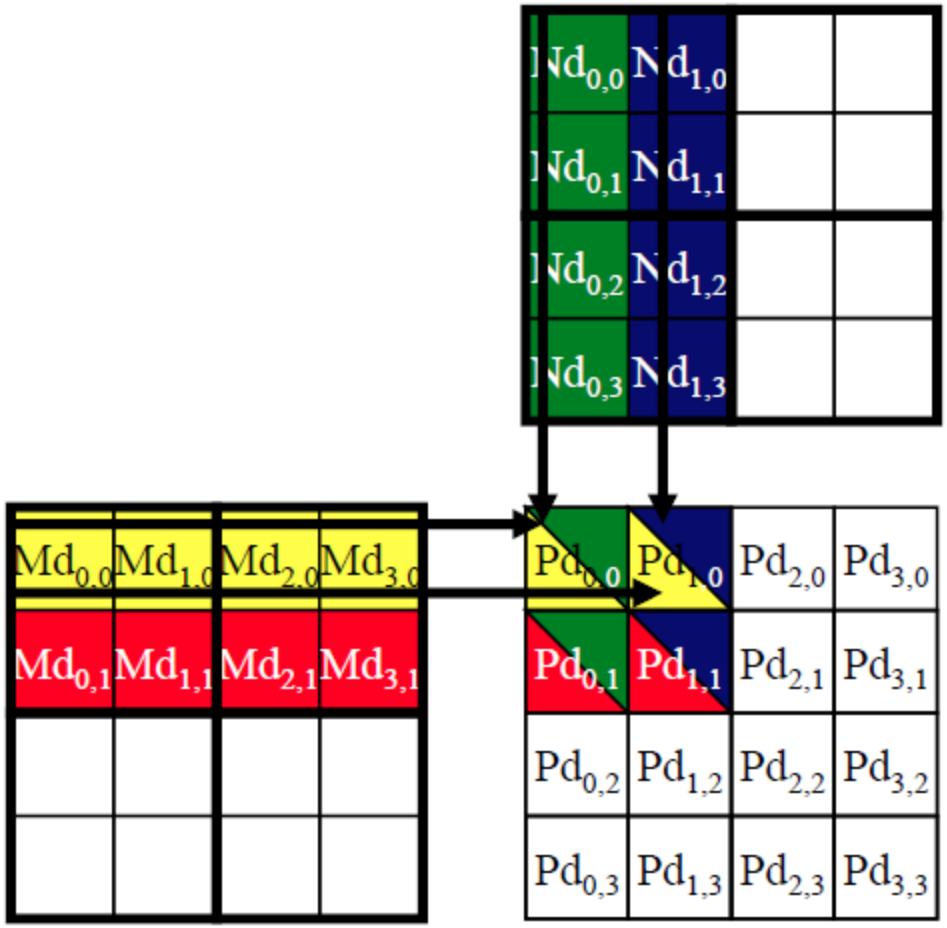


Shared Data in Matrix



Every Md and Nd Element is used twice in a 2x2 tile
Load the data into shared memory and saved for the later use
Save 15 global memory access in a 16x16 tile

Matrix Tiling



Matrix Tiling

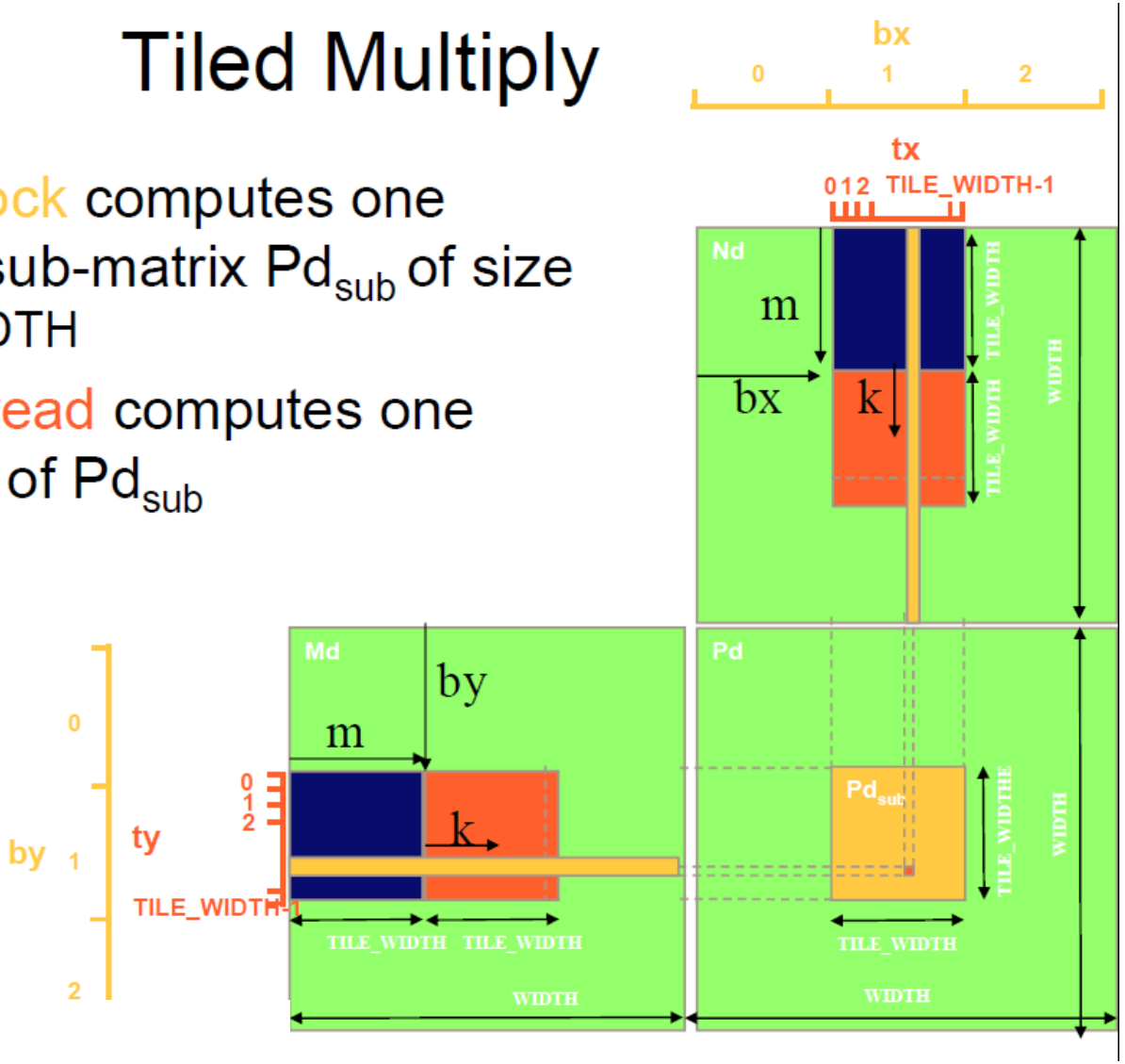
	Phase 1			Phase 2		
$T_{0,0}$	**$Md_{0,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,0}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	**$Md_{2,0}$** ↓ $Mds_{0,0}$	**$Nd_{0,2}$** ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	**$Md_{1,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,0}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	**$Md_{3,0}$** ↓ $Mds_{1,0}$	**$Nd_{1,2}$** ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	**$Md_{0,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,1}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	**$Md_{2,1}$** ↓ $Mds_{0,1}$	**$Nd_{0,3}$** ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	**$Md_{1,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,1}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	**$Md_{3,1}$** ↓ $Mds_{1,1}$	**$Nd_{1,3}$** ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Tiled Multiply

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



Tiling Code

```
// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
dim3 dimGrid(Width  / TILE_WIDTH,
             Width /  TILE_WIDTH);
```

Tiling 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.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   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) {
// Coolaborative 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[Col + (m*TILE_WIDTH + ty)*Width];
11.     __syncthreads();

11.    for (int k = 0; k < TILE_WIDTH; ++k)
12.    Pvalue += Mds[ty][k] * Nds[k][tx];
13.    Synchthreads();
14.   }
13.    Pd[Row*Width+Col] = Pvalue;
}
```

Tiling Impact

- G80 without tiling
  - *367Gflops/s*
  - *21.6/367=5.8% potential is used*
- G80 with 16x16 tiling
  - *G80 has 16KB shard memory*
  - *16*16*2*4=2KB shared memory for each block, can accommodate 8 blocks*
  - *21.6*15=324Gflops/s*
  - *324/367=88% potential is used*
- However G80 only support 768 threads per SM
  - *16*16=256 threads, 768/256=3 blocks*
  - *Only 6KB shared memory is used*
  - *Fermi can support 1536 threads per SM*

Performance