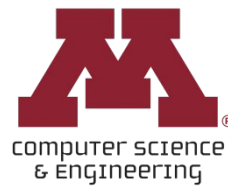


CSCI 5451: Introduction to Parallel Computing

Lecture 20: CUDA Memory Architecture



Announcements (11/12)

- ❑ HW3 + Project PDF release pushed back
 - Monitor slack this afternoon



Matrix Multiplication (MM) Kernel

- ❑ Multiply $P = MN$
- ❑ Assume M, N are *Width x Width* matrices

```
01  __global__ void MatrixMulKernel(float* M, float* N,  
02                                  float* P, int Width) {  
03      int row = blockIdx.y*blockDim.y+threadIdx.y;  
04      int col = blockIdx.x*blockDim.x+threadIdx.x;  
05      if ((row < Width) && (col < Width)) {  
06          float Pvalue = 0;  
07          for (int k = 0; k < Width; ++k) {  
08              Pvalue += M[row*Width+k]*N[k*Width+col];  
09          }  
10          P[row*Width+col] = Pvalue;  
11      }  
12  }
```



Naive MM Memory Limitations

- ❑ We focus on just the multiplication portion of the kernel
- ❑ How many floating point operations do we perform for every byte we load?
 -

```
for (int k = 0; k < Width; ++k) {  
    Pvalue += M[row*Width+k] * N[k*Width+col];  
}
```



Naive MM Memory Limitations

- ❑ We focus on just the multiplication portion of the kernel
- ❑ How many floating point operations do we perform for every byte we load?
 - 8 Bytes (4 from M, 4 from N) loaded
 - 2 FLOPs performed (1 add, 1 multiply)
 - .25 FLOPs per Byte (FLOP/B or OP/B)
- ❑ The term FLOP/B or OP/B is also called the *arithmetic intensity* or *compute intensity*

```
for (int k = 0; k < Width; ++k) {  
    Pvalue += M[row*Width+k] * N[k*Width+col];  
}
```



Naive MM Memory Limitations

- ❑ On the A100, when using fp32, we have 19,500 GFLOPs
- ❑ If we use tensor cores, this increases to 156,000 GFLOPs
- ❑ The memory bandwidth is 1,600 GB/s
- ❑ Why is this a problem?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Naive MM Memory Limitations

- ❑ On the A100, when using fp32, we have 19,500 GFLOPs
- ❑ If we use tensor cores, this increases to 156,000 GFLOPs
- ❑ The memory bandwidth is 1,600 GB/s
- ❑ Why is this a problem?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Naive MM Memory Limitations

- ❑ On the A100, when using fp32, we have 19,500 GFLOPs
- ❑ If we use tensor cores, this increases to 156,000 GFLOPs
- ❑ The memory bandwidth is 1,600 GB/s
- ❑ Why is this a problem?
 - With .25 FLOPs/B, we max out at 400 GFLOPs
 - This is significantly less than 19,500 GFLOPs and even more significantly less than 156,000 GFLOPs

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Naive MM Memory Limitations

- ❑ When execution speed is limited by the memory bandwidth, we call the program *memory bound*
- ❑ In order to produce a program which is not memory bound, we need to increase the arithmetic intensity
- ❑ In this example, to no longer be memory bound we have to increase the arithmetic intensity to at least $19,500/1,600 \approx 12.2$

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Caution on Published Specs

- ❑ The image at right is an example of a *specification (spec)* published for the A100
- ❑ NVIDIA often releases these for their new models to demonstrate performance capabilities
- ❑ Based on the previous slides what is an issue with this?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Caution on Published Specs

- ❑ The image at right is an example of a *specification (spec)* published for the A100
- ❑ NVIDIA often releases these for their new models to demonstrate performance capabilities
- ❑ Based on the previous slides what is an issue with this?
 - The demonstrated speeds are only achievable given that the program has a high enough arithmetic intensity

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Roofline Models

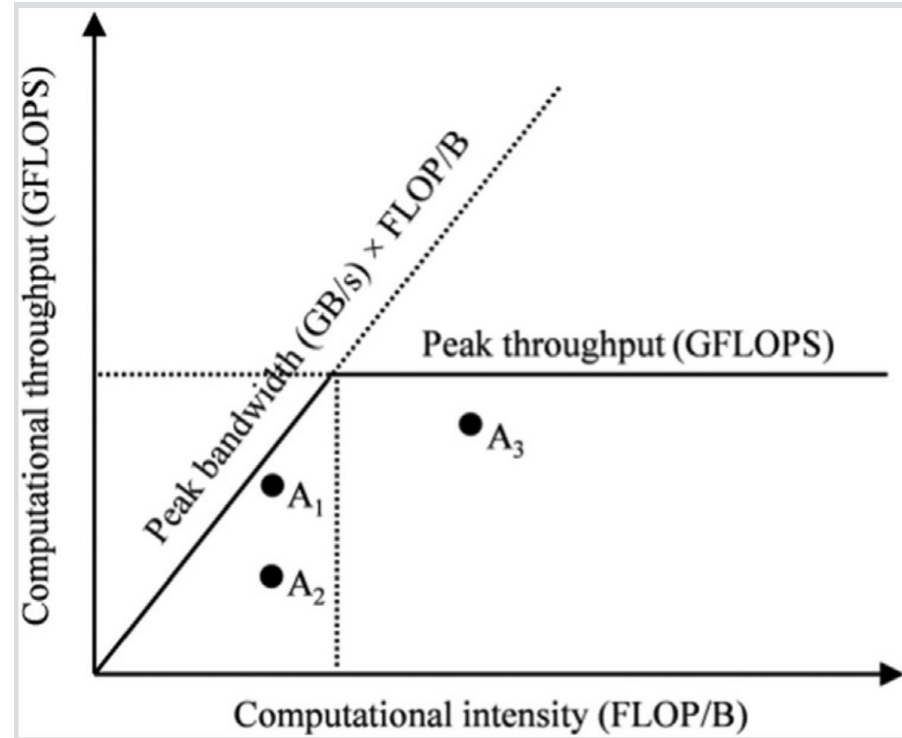
Once we are no longer memory bound,
can we continue to increase the
execution throughput (FLOP/second)?



Roofline Models

Once we are no longer memory bound, can we continue to increase the execution throughput (FLOP/second)?

- ❑ No. The execution throughput values of the previous slide are a hard upper limit.
- ❑ This is represented visually using what are called roofline models



Memory Types

- ❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA
- ❑ CUDA has 5 main memory types
 - Per-thread Register
 - Per-thread Local
 - Per-threadblock Shared
 - Per-grid Global
 - Per-grid Constant



Memory Types

- ❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA
- ❑ CUDA has 5 main memory types

- **Per-thread Register**

- Per-thread Local
- Per-threadblock Shared
- Per-grid Global
- Per-grid Constant



- Each thread has some register space in its subpartition
- This is the lowest latency form of memory storage
- This corresponds to the Register File above
- They are private to each thread
- Very small amount of storage



Memory Types

❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA

❑ CUDA has 5 main memory types

- Per-thread Register
- **Per-thread Local**
- Per-threadblock Shared
- Per-grid Global
- Per-grid Constant



- Local storage holds variables which don't fit in the registers
- These are also private to each thread
- Each thread makes use of the full memory hierarchy based on the size of the object (L1 cache in the SM, L2 cache GPU-wide, HBM). If the per-thread objects grow in size, then you have to use a larger cache.



Memory Types

❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA

❑ CUDA has 5 main memory types

- Per-thread Register
- Per-thread Local
- **Per-threadblock Shared**
- Per-grid Global
- Per-grid Constant



- Stored on-chip SRAM located in the SM
- Is exclusively in the L1 cache
- All threads in a block have access to this memory
- This memory has limitations, so you cannot overuse or your program will slow down and/or fail
- Relatively small size



Memory Types

❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA

❑ CUDA has 5 main memory types

- Per-thread Register
- Per-thread Local
- Per-threadblock Shared
- **Per-grid Global**
- Per-grid Constant

- Visible to all threads in a grid
- Very slow
- Uses full memory hierarchy - so the access can be cached, but this is contingent on previous accesses
- Very large size



Memory Types

❑ In order to improve performance, we need to have a clearer understanding of memory with CUDA

❑ CUDA has 5 main memory types

- Per-thread Register
- Per-thread Local
- Per-threadblock Shared
- Per-grid Global
- **Per-grid Constant**

- Visible to all threads in a grid
- These are cached, but read only
- Lookups can be quite fast
- The current limitation on constant memory is 65,536 bytes



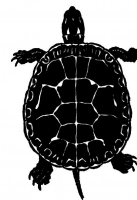
Memory Loading Review



```
fadd r1, r2, r3
```

VS

```
load r2, r4, offset  
fadd r1, r2, r3
```



- ❑ We always want to load things which are closer to the actual execution units
 - Data in registers is best, then L1 cache, then L2, then HBM
- ❑ Loading from further down the memory hierarchy takes more time & vastly more energy
- ❑ Thus we have a tension between small size of fast memory (registers) and large size of slow memory (HBM)

How do we manipulate each of these datatypes in CUDA?



Five Memory Types In Practice

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Grid
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application



Five Memory Types In Practice

```
__global__ void kernel_register_example() {  
    int a = threadIdx.x;    // automatic scalar → stored in a register  
    int b = a * 2;  
}
```

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application



Five Memory Types In Practice

```
__global__ void kernel_local_array_example() {  
    int arr[8];           // automatic array → stored in local memory  
    arr[threadIdx.x % 8] = threadIdx.x;  
}
```

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application



Five Memory Types In Practice

```
__global__ void kernel_shared_example() {  
    __shared__ int sharedVar[32]; // one per block  
    sharedVar[threadIdx.x] = threadIdx.x;  
    __syncthreads();  
}
```

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Grid
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application



Five Memory Types In Practice

```
__device__ int GlobalVar;    // single copy in global memory (application lifetime)

__global__ void kernel_global_example() {
    GlobalVar = blockIdx.x; // visible to all threads and kernels
}
```

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application



Five Memory Types In Practice

```
__constant__ int ConstVar = 42; // stored in constant memory

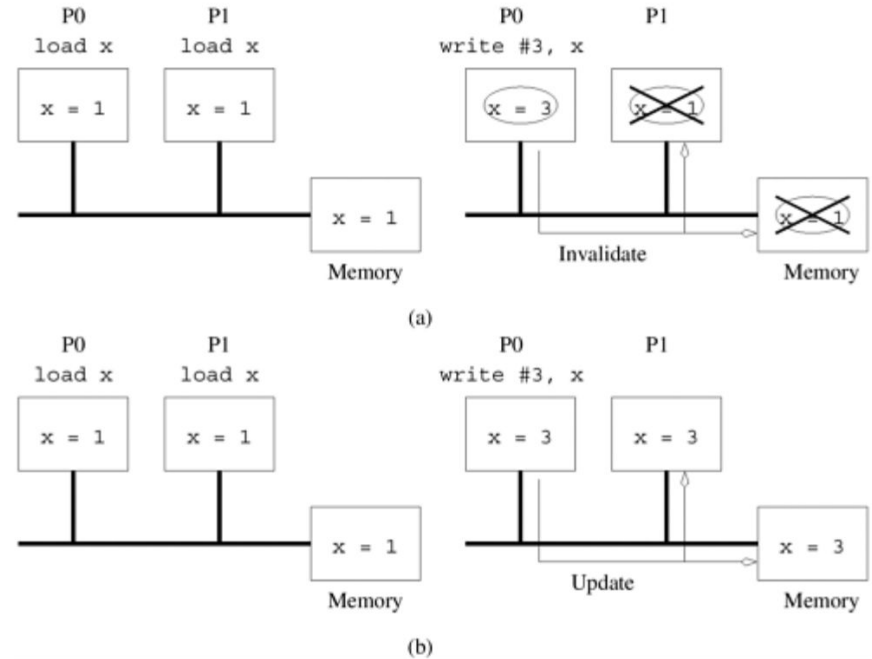
__global__ void kernel_constant_example() {
    int val = ConstVar; // all threads read from constant cache
}
```

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application



GPU Cache Coherence

- ❑ In one of our earlier lectures, we discussed Cache Coherence on Shared Address Machines
- ❑ We discussed how writes have to propagate through memory into the caches of other cores
- ❑ Invalidate and Update protocols are common ways of achieving this
- ❑ How is this done in GPUs?



GPU Cache Coherence

- ❑ The per-thread memories (registers/local) are private & thus do not need to have any shared state with other threads
- ❑ The constant variables are read-only
- ❑ The per-threadblock variables in L1 cache **do not** have any cache coherence protocols
- ❑ Global per-grid variables are shared
 - L1 cache is not updated → threads only know a variable has changed on a read operation
 - All threads have write access → there is no locking mechanism across threads on these variables



How can we use what we have learned of the memory hierarchy to improve our earlier Matrix Multiplication kernel?

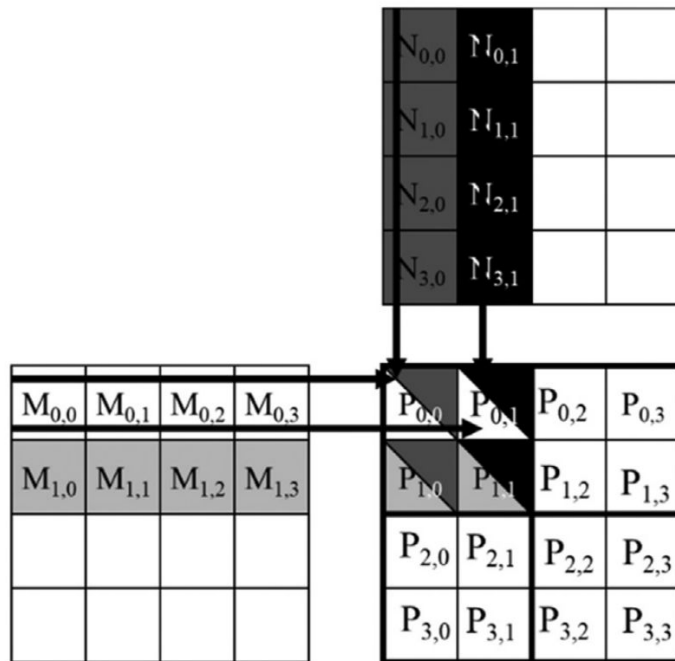


Tiling Motivation

```
dim3 threadsPerBlock(2, 2);
dim3 numBlocks(2, 2);
MatrixMulKernel<<<numBlocks,
threadsPerBlock>>>(M, N, P, Width);
```

- ❑ Suppose we prepare our matrix multiplication algorithm with 2x2 blocks of 2x2 threads
- ❑ Within a block, how many redundant loads of a row of M occur? A column of N?

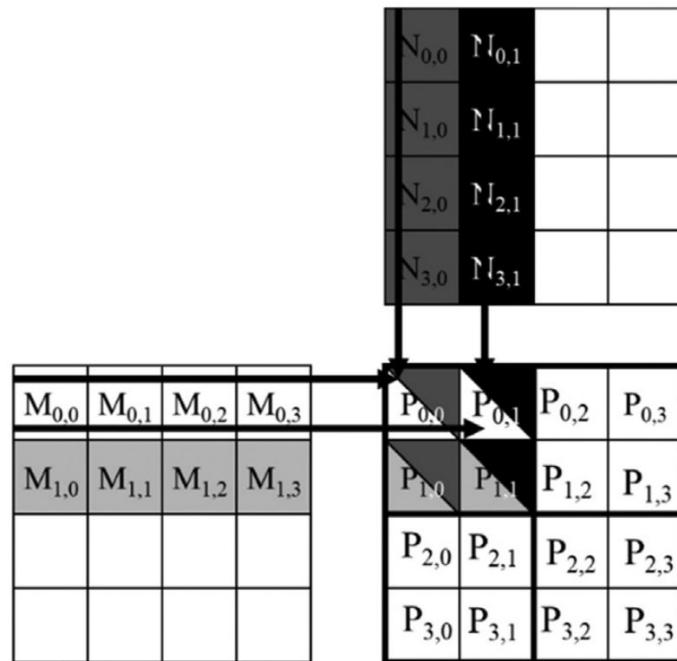
○



Tiling Motivation

```
dim3 threadsPerBlock(2, 2);
dim3 numBlocks(2, 2);
MatrixMulKernel<<<numBlocks,
threadsPerBlock>>>(M, N, P, Width);
```


- ❑ Suppose we prepare our matrix multiplication algorithm with 2x2 blocks of 2x2 threads
- ❑ Within a block, how many redundant loads of a row of M occur? A column of N?
 - 2
- ❑ Tiling is based on the idea that we should reduce these redundant loads by using per-threadblock *shared* memory



Tiling Motivation

Removing redundant accesses will increase the compute intensity of our program → For every byte we load, we are computing more.

Access order



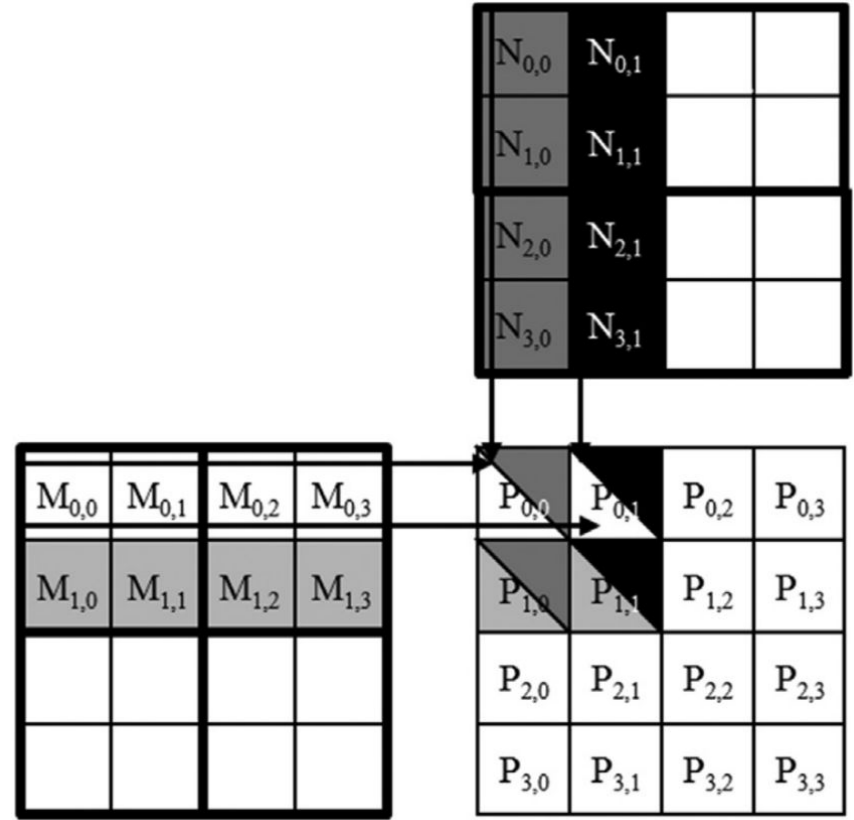
thread _{0,0}	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread _{0,1}	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread _{1,0}	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread _{1,1}	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$



Tiling Strategy

Given that shared memory is not infinite (it is limited by the size of the partition in the L1 cache), we must follow the below general strategy

- Repeat the following until complete
 - Load in small chunk in parallel
 - Compute in parallel



Matrix Multiply Tiling (2x2 block with 2x2 thread)

Block 0
Tiling Steps
(threads
execute in
parallel)

	Phase 0			Phase 1		
thread _{0,0}	$\mathbf{M}_{0,0}$ ↓ Mds _{0,0}	$\mathbf{N}_{0,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	$\mathbf{M}_{0,2}$ ↓ Mds _{0,0}	$\mathbf{N}_{2,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	$\mathbf{M}_{0,1}$ ↓ Mds _{0,1}	$\mathbf{N}_{0,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	$\mathbf{M}_{0,3}$ ↓ Mds _{0,1}	$\mathbf{N}_{2,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	$\mathbf{M}_{1,0}$ ↓ Mds _{1,0}	$\mathbf{N}_{1,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	$\mathbf{M}_{1,2}$ ↓ Mds _{1,0}	$\mathbf{N}_{3,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	$\mathbf{M}_{1,1}$ ↓ Mds _{1,1}	$\mathbf{N}_{1,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	$\mathbf{M}_{1,3}$ ↓ Mds _{1,1}	$\mathbf{N}_{3,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}
			time →			



Matrix Multiply Tiling (2x2 block with 2x2 thread)

	Phase 0			Phase 1		
thread _{0,0}	$\mathbf{M}_{0,0}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{0,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\text{PValue}_{0,0} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,0}$	$\mathbf{M}_{0,2}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{2,0}$ ↓ $\mathbf{Nds}_{0,0}$	$\text{PValue}_{0,0} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,0}$
thread _{0,1}	$\mathbf{M}_{0,1}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{0,1}$ ↓ $\mathbf{Nds}_{0,1}$	$\text{PValue}_{0,1} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,1}$	$\mathbf{M}_{0,3}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{2,1}$ ↓ $\mathbf{Nds}_{0,1}$	$\text{PValue}_{0,1} +=$ $\mathbf{Mds}_{0,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{0,1} * \mathbf{Nds}_{1,1}$
thread _{1,0}	$\mathbf{M}_{1,0}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{1,0}$ ↓ $\mathbf{Nds}_{1,0}$	$\text{PValue}_{1,0} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{1,1} * \mathbf{Nds}_{1,0}$	$\mathbf{M}_{1,2}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{3,0}$ ↓ $\mathbf{Nds}_{1,0}$	$\text{PValue}_{1,0} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,0} +$ $\mathbf{Mds}_{1,1} * \mathbf{Nds}_{1,0}$
thread _{1,1}	$\mathbf{M}_{1,1}$ ↓ $\mathbf{Mds}_{1,1}$	$\mathbf{N}_{1,1}$ ↓ $\mathbf{Nds}_{1,1}$	$\text{PValue}_{1,1} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{1,1} * \mathbf{Nds}_{1,1}$	$\mathbf{M}_{1,3}$ ↓ $\mathbf{Mds}_{1,1}$	$\mathbf{N}_{3,1}$ ↓ $\mathbf{Nds}_{1,1}$	$\text{PValue}_{1,1} +=$ $\mathbf{Mds}_{1,0} * \mathbf{Nds}_{0,1} +$ $\mathbf{Mds}_{1,1} * \mathbf{Nds}_{1,1}$

time →



Matrix Multiply Tiling (2x2 block with 2x2 thread)

$\mathbf{M}_{0,0}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{0,0}$ ↓ $\mathbf{Nds}_{0,0}$
$\mathbf{M}_{0,1}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{0,1}$ ↓ $\mathbf{Nds}_{0,1}$
$\mathbf{M}_{1,0}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{1,0}$ ↓ $\mathbf{Nds}_{1,0}$
$\mathbf{M}_{1,1}$ ↓ $\mathbf{Mds}_{1,1}$	$\mathbf{N}_{1,1}$ ↓ $\mathbf{Nds}_{1,1}$

$\mathbf{N}_{0,0}$	$\mathbf{N}_{0,1}$
$\mathbf{N}_{1,0}$	$\mathbf{N}_{1,1}$

$\mathbf{M}_{0,0}$	$\mathbf{M}_{0,1}$
$\mathbf{M}_{1,0}$	$\mathbf{M}_{1,1}$



Matrix Multiply Tiling (2x2 block with 2x2 thread)

	Phase 0			Phase 1		
thread _{0,0}	$\mathbf{M}_{0,0}$ ↓ Mds _{0,0}	$\mathbf{N}_{0,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	$\mathbf{M}_{0,2}$ ↓ Mds _{0,0}	$\mathbf{N}_{2,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	$\mathbf{M}_{0,1}$ ↓ Mds _{0,1}	$\mathbf{N}_{0,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	$\mathbf{M}_{0,3}$ ↓ Mds _{0,1}	$\mathbf{N}_{2,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	$\mathbf{M}_{1,0}$ ↓ Mds _{1,0}	$\mathbf{N}_{1,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	$\mathbf{M}_{1,2}$ ↓ Mds _{1,0}	$\mathbf{N}_{3,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	$\mathbf{M}_{1,1}$ ↓ Mds _{1,1}	$\mathbf{N}_{1,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	$\mathbf{M}_{1,3}$ ↓ Mds _{1,1}	$\mathbf{N}_{3,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time →



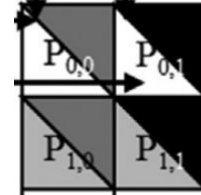
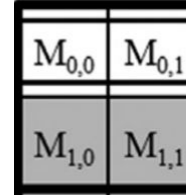
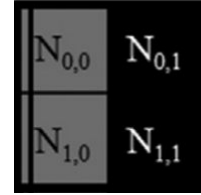
Matrix Multiply Tiling (2x2 block with 2x2 thread)

$\begin{aligned} &PValue_{0,0} += \\ &Mds_{0,0} * Nds_{0,0} + \\ &Mds_{0,1} * Nds_{1,0} \end{aligned}$
--

$\begin{aligned} &PValue_{0,1} += \\ &Mds_{0,0} * Nds_{0,1} + \\ &Mds_{0,1} * Nds_{1,1} \end{aligned}$
--

$\begin{aligned} &PValue_{1,0} += \\ &Mds_{1,0} * Nds_{0,0} + \\ &Mds_{1,1} * Nds_{1,0} \end{aligned}$
--

$\begin{aligned} &PValue_{1,1} += \\ &Mds_{1,0} * Nds_{0,1} + \\ &Mds_{1,1} * Nds_{1,1} \end{aligned}$
--



Matrix Multiply Tiling (2x2 block with 2x2 thread)

	Phase 0			Phase 1		
thread _{0,0}	$\mathbf{M}_{0,0}$ ↓ Mds _{0,0}	$\mathbf{N}_{0,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	$\mathbf{M}_{0,2}$ ↓ Mds _{0,0}	$\mathbf{N}_{2,0}$ ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	$\mathbf{M}_{0,1}$ ↓ Mds _{0,1}	$\mathbf{N}_{0,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	$\mathbf{M}_{0,3}$ ↓ Mds _{0,1}	$\mathbf{N}_{2,1}$ ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	$\mathbf{M}_{1,0}$ ↓ Mds _{1,0}	$\mathbf{N}_{1,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	$\mathbf{M}_{1,2}$ ↓ Mds _{1,0}	$\mathbf{N}_{3,0}$ ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	$\mathbf{M}_{1,1}$ ↓ Mds _{1,1}	$\mathbf{N}_{1,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	$\mathbf{M}_{1,3}$ ↓ Mds _{1,1}	$\mathbf{N}_{3,1}$ ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}
time —————→						



Matrix Multiply Tiling (2x2 block with 2x2 thread)

$\mathbf{M}_{0,2}$ ↓ $\mathbf{Mds}_{0,0}$	$\mathbf{N}_{2,0}$ ↓ $\mathbf{Nds}_{0,0}$
$\mathbf{M}_{0,3}$ ↓ $\mathbf{Mds}_{0,1}$	$\mathbf{N}_{2,1}$ ↓ $\mathbf{Nds}_{0,1}$
$\mathbf{M}_{1,2}$ ↓ $\mathbf{Mds}_{1,0}$	$\mathbf{N}_{3,0}$ ↓ $\mathbf{Nds}_{1,0}$
$\mathbf{M}_{1,3}$ ↓ $\mathbf{Mds}_{1,1}$	$\mathbf{N}_{3,1}$ ↓ $\mathbf{Nds}_{1,1}$

$\mathbf{M}_{0,2}$	$\mathbf{M}_{0,3}$
$\mathbf{M}_{1,2}$	$\mathbf{M}_{1,3}$

$\mathbf{N}_{2,0}$	$\mathbf{N}_{2,1}$
$\mathbf{N}_{3,0}$	$\mathbf{N}_{3,1}$



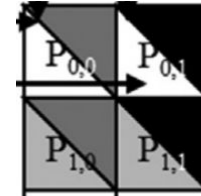
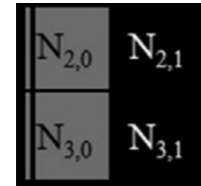
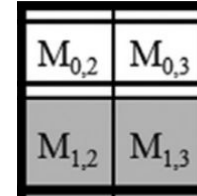
Matrix Multiply Tiling (2x2 block with 2x2 thread)

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}
	time					



Matrix Multiply Tiling (2x2 block with 2x2 thread)

$\begin{aligned} \text{PValue}_{0,0} &+= \\ &\text{Mds}_{0,0} * \text{Nds}_{0,0} + \\ &\text{Mds}_{0,1} * \text{Nds}_{1,0} \end{aligned}$
$\begin{aligned} \text{PValue}_{0,1} &+= \\ &\text{Mds}_{0,0} * \text{Nds}_{0,1} + \\ &\text{Mds}_{0,1} * \text{Nds}_{1,1} \end{aligned}$
$\begin{aligned} \text{PValue}_{1,0} &+= \\ &\text{Mds}_{1,0} * \text{Nds}_{0,0} + \\ &\text{Mds}_{1,1} * \text{Nds}_{1,0} \end{aligned}$
$\begin{aligned} \text{PValue}_{1,1} &+= \\ &\text{Mds}_{1,0} * \text{Nds}_{0,1} + \\ &\text{Mds}_{1,1} * \text{Nds}_{1,1} \end{aligned}$



Tiled MM Kernel

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28      P[Row*Width + Col] = Pvalue;
29
30
31 }
```



Tiled MM Kernel

ph=0

$M_{0,0}$ ↓ $Mds_{0,0}$	$N_{0,0}$ ↓ $Nds_{0,0}$
$M_{0,1}$ ↓ $Mds_{0,1}$	$N_{0,1}$ ↓ $Nds_{0,1}$
$M_{1,0}$ ↓ $Mds_{1,0}$	$N_{1,0}$ ↓ $Nds_{1,0}$
$M_{1,1}$ ↓ $Mds_{1,1}$	$N_{1,1}$ ↓ $Nds_{1,1}$

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27
28      }
29      P[Row*Width + Col] = Pvalue;
30
31 }
```



Tiled MM Kernel

ph=0

$P_{0,0} +=$
 $M_{0,0} * N_{0,0} +$
 $M_{0,1} * N_{1,0}$

$P_{0,1} +=$
 $M_{0,0} * N_{0,1} +$
 $M_{0,1} * N_{1,1}$

$P_{1,0} +=$
 $M_{1,0} * N_{0,0} +$
 $M_{1,1} * N_{1,0}$

$P_{1,1} +=$
 $M_{1,0} * N_{0,1} +$
 $M_{1,1} * N_{1,1}$

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27
28      }
29      P[Row*Width + Col] = Pvalue;
30
31 }
```



Tiled MM Kernel

ph=1

$M_{0,2}$ ↓ $Mds_{0,0}$	$N_{2,0}$ ↓ $Nds_{0,0}$
$M_{0,3}$ ↓ $Mds_{0,1}$	$N_{2,1}$ ↓ $Nds_{0,1}$
$M_{1,2}$ ↓ $Mds_{1,0}$	$N_{3,0}$ ↓ $Nds_{1,0}$
$M_{1,3}$ ↓ $Mds_{1,1}$	$N_{3,1}$ ↓ $Nds_{1,1}$

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28      P[Row*Width + Col] = Pvalue;
29
30
31 }
```



Tiled MM Kernel

ph=1

$P_{0,0} +=$
 $M_{0,0} * N_{0,0} +$
 $M_{0,1} * N_{1,0}$

$P_{0,1} +=$
 $M_{0,0} * N_{0,1} +$
 $M_{0,1} * N_{1,1}$

$P_{1,0} +=$
 $M_{1,0} * N_{0,0} +$
 $M_{1,1} * N_{1,0}$

$P_{1,1} +=$
 $M_{1,0} * N_{0,1} +$
 $M_{1,1} * N_{1,1}$

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27
28      }
29      P[Row*Width + Col] = Pvalue;
30
31 }
```



Tiled MM Kernel

Given an arbitrary
TILE_WIDTH, and arbitrary
Width, what must the kernel
invocation look like (i.e. what
are the arguments to the
kernel)?

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28      P[Row*Width + Col] = Pvalue;
29
30
31 }
```



Tiled MM Kernel

Given an arbitrary
TILE_WIDTH, and arbitrary
Width, what must the kernel
invocation look like (i.e. what
are the arguments to the
kernel)?

```
int sz = ceil(Width/TILE_WIDTH)
dim3 a(sz, sz);
dim3 b(TILE_WIDTH, TILE_WIDTH);
MatrixMulKernel<<<a, b>>>(M, N, P,
                          Width);
```

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28
29      P[Row*Width + Col] = Pvalue;
30
31 }
```



Tiled MM Kernel

What is the best possible
TILE_WIDTH we can
practically choose?

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28      P[Row*Width + Col] = Pvalue;
29
30
31 }
```



Tiled MM Kernel

What is the best possible
TILE_WIDTH we can
practically choose?

32

Let TILE_WIDTH=2

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27      }
28      P[Row*Width + Col] = Pvalue;
29
30
31 }
```



Tiling on the A100

- ❑ Recall that our earlier compute intensity was .25FLOP/B
- ❑ What is our new compute intensity with a TILE_WIDTH of 2?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Tiling on the A100

- ❑ Recall that our earlier compute intensity was .25FLOP/B
- ❑ What is our new compute intensity with a TILE_WIDTH of 2?
 - .5FLOP/B
- ❑ 4, 8, 16?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Tiling on the A100

- ❑ Recall that our earlier compute intensity was .25FLOP/B
- ❑ What is our new compute intensity with a TILE_WIDTH of 2?
 - .5FLOP/B
- ❑ 4, 8, 16?
 - $4 \rightarrow 1\text{FLOP/B}$
 - $8 \rightarrow 2\text{FLOP/B}$
 - $16 \rightarrow 4\text{FLOP/B}$
- ❑ New Execution throughput with 1,600 GB/s and TILE_WIDTH=16?
 -

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



Tiling on the A100

- ❑ Recall that our earlier compute intensity was .25FLOP/B
- ❑ What is our new compute intensity with a TILE_WIDTH of 2?
 - .5FLOP/B
- ❑ 4, 8, 16?
 - $4 \rightarrow 1\text{FLOP/B}$
 - $8 \rightarrow 2\text{FLOP/B}$
 - $16 \rightarrow 4\text{FLOP/B}$
- ❑ New Execution throughput with 1,600 GB/s and TILE_WIDTH=16?
 - 6,400 GFLOPS

	Peak Performance
Transistor Count	54 billion
Die Size	826 mm ²
FP64 CUDA Cores	3,456
FP32 CUDA Cores	6,912
Tensor Cores	432
Streaming Multiprocessors	108
FP64	9.7 teraFLOPS
FP64 Tensor Core	19.5 teraFLOPS
FP32	19.5 teraFLOPS
TF32 Tensor Core	156 teraFLOPS 312 teraFLOPS*
BFLOAT16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
FP16 Tensor Core	312 teraFLOPS 624 teraFLOPS*
INT8 Tensor Core	624 TOPS 1,248 TOPS*
INT4 Tensor Core	1,248 TOPS 2,496 TOPS*
GPU Memory	40 GB
GPU Memory Bandwidth	1.6 TB/s
Interconnect	NVLink 600 GB/s PCIe Gen4 64 GB/s
Multi-Instance GPUs	Various Instance sizes with up to 7MIGs @5GB
Form Factor	4/8 SXM GPUs in HGX A100
Max Power	400W (SXM)



How can we apply boundary checking to this kernel?



Boundary Checking

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27
28      }
29      P[Row*Width + Col] = Pvalue;
30
31  }
```



Boundary Checking

```
// Loop over the M and N tiles required to compute P element
float Pvalue = 0;
for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTH); ++ph) {

    // Collaborative loading of M and N tiles into shared memory
    if ((Row < Width) && (ph*TILE_WIDTH+tx) < Width)
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
    else Mds[ty][tx] = 0.0f;
    if ((ph*TILE_WIDTH+ty) < Width && Col < Width)
        Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
    else Nds[ty][tx] = 0.0f;
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; ++k) {
        Pvalue += Mds[ty][k] * Nds[k][tx];
    }
    __syncthreads();
}
if (Row < Width) && (Col < Width)
    P[Row*Width + Col] = Pvalue;
```

```
02  __global__ void matrixMulKernel(float* M, float* N, float* P, int Width) {
03
04      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
05      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
06
07      int bx = blockIdx.x;  int by = blockIdx.y;
08      int tx = threadIdx.x; int ty = threadIdx.y;
09
10      // Identify the row and column of the P element to work on
11      int Row = by * TILE_WIDTH + ty;
12      int Col = bx * TILE_WIDTH + tx;
13
14      // Loop over the M and N tiles required to compute P element
15      float Pvalue = 0;
16      for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
17
18          // Collaborative loading of M and N tiles into shared memory
19          Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
20          Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
21          __syncthreads();
22
23          for (int k = 0; k < TILE_WIDTH; ++k) {
24              Pvalue += Mds[ty][k] * Nds[k][tx];
25          }
26          __syncthreads();
27
28      }
29      P[Row*Width + Col] = Pvalue;
30
31  }
```



A100 MatMul Kernel Occupancy

- ❑ Consider the Kernel we just developed in lecture on an A100
- ❑ The A100 has a limitation of 164KB of shared memory/SM, with a maximum of 2048 threads/SM
- ❑ Therefore, the maximum shared memory per thread is $164\text{KB}/2048 = 82\text{B/thread}$
- ❑ Each block in our kernel has $(\text{TILE_WIDTH} * \text{TILE_WIDTH})$ threads

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```



A100 MatMul Kernel Occupancy

- ❑ Consider the Kernel we just developed in lecture on an A100
- ❑ The A100 has a limitation of 164KB of shared memory/SM, with a maximum of 2048 threads/SM
- ❑ Therefore, the maximum shared memory per thread is $164\text{KB}/2048 = 82\text{B}/\text{thread}$
- ❑ Each block in our kernel has $(\text{TILE_WIDTH} * \text{TILE_WIDTH})$ threads

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];  
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

$$\begin{aligned} & (4\text{B} * \text{TILE_WIDTH}^2 + \\ & 4\text{B} * \text{TILE_WIDTH}^2) \\ & \text{per block} \end{aligned}$$

$$\begin{aligned} & (8\text{B} * \text{TILE_WIDTH}^2) / \\ & (\text{TILE_WIDTH}^2) \\ & = \\ & 8\text{B}/\text{thread} \end{aligned}$$

Not limited by
shared memory



A100 General Example

- ❑ Suppose instead that we have a kernel which has threadblocks which each use 32KB of memory
- ❑ Additionally, assume each threadblock contains 256 threads
- ❑ Recall that the limitation is 82B/thread



A100 General Example

- ❑ Suppose instead that we have a kernel which has threadblocks which each use 32KB of memory
- ❑ Additionally, assume each threadblock contains 256 threads
- ❑ Recall that the limitation is 82B/thread

$$\begin{aligned} & (32\text{KB/block}) / \\ & (256 \text{ threads/block}) \\ & = \\ & 132 \text{ B/thread} \end{aligned}$$

Limited by
Memory



A note on cuBLAS/CUTLASS

- ❑ For many algorithms, you can use existing kernels from libraries such as cuBLAS or CUTLASS
- ❑ These algorithms have many more optimizations (some of which we will cover in more detail)
- ❑ They are highly tuned to the specific hardware (implementations may differ based on A100 vs. H100 vs. etc.)
- ❑ In practical settings for many common algorithms, you should start here, then work to build your own



Examples

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: Analyze the elements that are accessed by each thread and see whether there is any commonality between threads.



Examples

4. Assuming that capacity is not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.
5. For our tiled matrix-matrix multiplication kernel, if we use a 32×32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?
6. Assume that a CUDA kernel is launched with 1000 thread blocks, each of which has 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?
7. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?



Examples

- 8.** Consider performing a matrix multiplication of two input matrices with dimensions $N \times N$. How many times is each element in the input matrices requested from global memory when:
- a.** There is no tiling?
 - b.** Tiles of size $T \times T$ are used?



Examples

a. How many versions of the variable `i` are there?

b. How many versions of the array `x[]` are there?

c. How many versions of the variable `y_s` are there?

d. How many versions of the array `b_s[]` are there?

e. What is the amount of shared memory used per block (in bytes)?

f. What is the floating-point to global memory access ratio of the kernel (in OP/B)?

```
01  __global__ void foo_kernel(float* a, float* b) {
02      unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
03      float x[4];
04      __shared__ float y_s;
05      __shared__ float b_s[128];
06      for(unsigned int j = 0; j < 4; ++j) {
07          x[j] = a[j*blockDim.x*gridDim.x + i];
08      }
09      if(threadIdx.x == 0) {
10          y_s = 7.4f;
11      }
12      b_s[threadIdx.x] = b[i];
13      __syncthreads();
14      b[i] = 2.5f*x[0] + 3.7f*x[1] + 6.3f*x[2] + 8.5f*x[3]
15            + y_s*b_s[threadIdx.x] + b_s[(threadIdx.x + 3)%128];
16  }
17  void foo(int* a_d, int* b_d) {
18      unsigned int N = 1024;
19      foo_kernel <<< (N + 128 - 1)/128, 128 >>>(a_d, b_d);
20  }
```

