GPU Programming
II – Memory and Synchronization

Stefan BORNHOFEN

```
#define N (1024*1024)
#define M 512
__global__ void add (float *a, float *b, float *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}

int main(void) {
        int size = N * sizeof(float);
        int *a, *b, *c, *d_a, *d_b, *d_c;
        a = (float*)malloc(size); random_vector (a, N);
        b = (float*)malloc(size); random_vector (b, N);
        c = (float*)malloc(size);
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        add<<<N/M,M>>>(d_a, d_b, d_c);
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Synchronous vs. Asynchronous calls

cudaMalloc(), cudaFree()
- synchronous : blocks the CPU until the allocation is complete

cudaMemcpy()
- synchronous : blocks the CPU until the copy is complete
- Copy begins when all preceding CUDA calls have completed

cudaDeviceSynchronize()
- synchronous : blocks the CPU until all preceding CUDA calls have completed

cudaMallocAsync(), cudaFreeAsync()
- asynchronous: does not block the CPU

cudaMemcpyAsync()
- asynchronous: does not block the CPU (pinned memory only!)

kernel <<<…>>> ()
- asynchronous : control returns to the CPU immediately
- CPU needs to synchronize before consuming the result

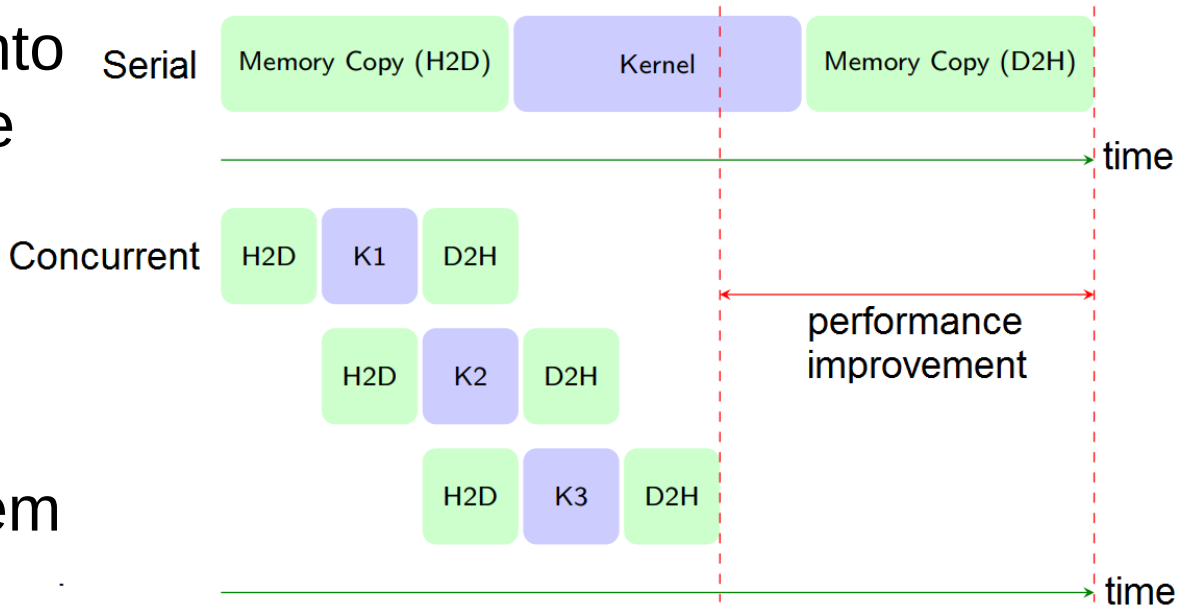
Streams

Streams introduce task parallelism. Streams are work queues to express concurrency between different tasks:

- host to device memory copies (pinned memory only!)
- kernel execution (allows concurrent kernels if there are enough GPU resources)
- device to host memory copies (pinned memory only!)

All tasks launched into the same stream are executed in order. Default stream: 0.

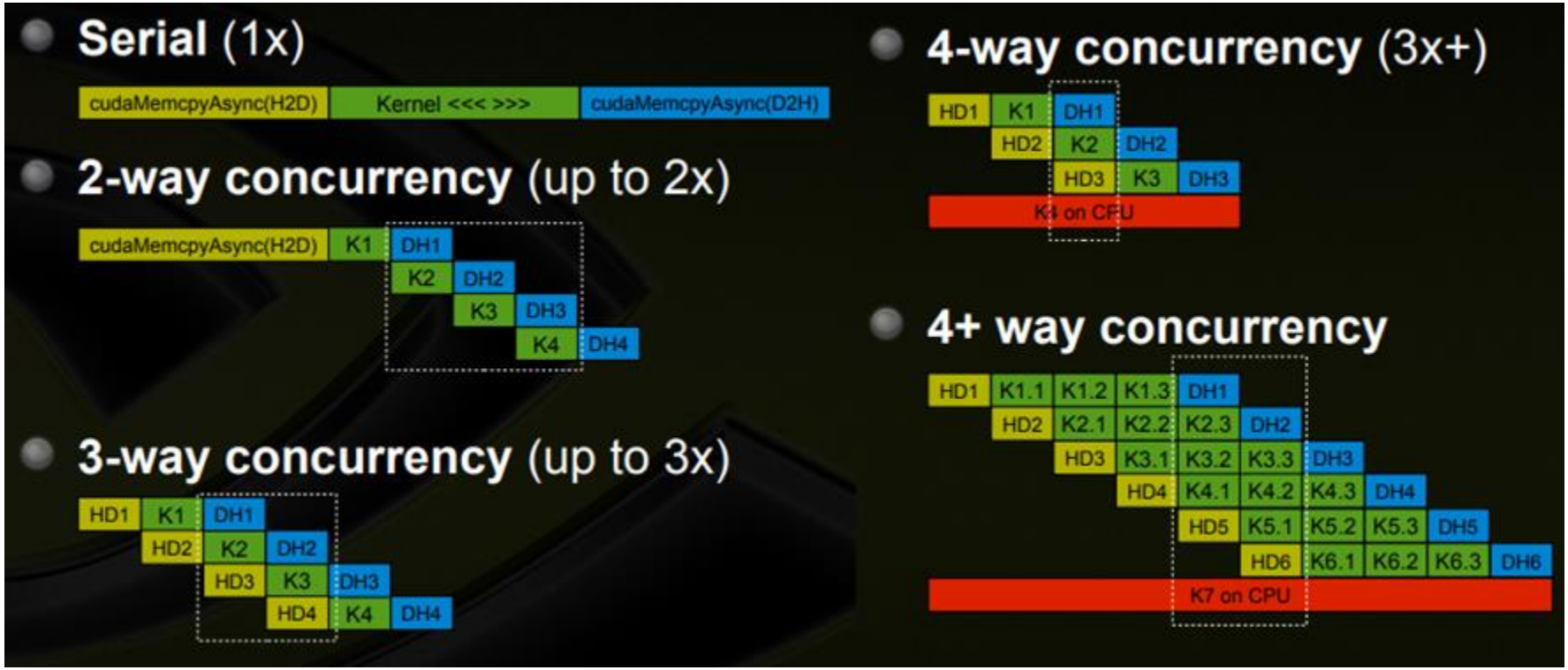
To overlap different tasks just launch them in different streams.



Amount of concurrency

Fermi architecture can simultaneously support (compute capability 2.0+)

- Up to 16 CUDA kernels on GPU
- 2 cudaMemcpyAsyncs (must be in different directions)
- Computation on the CPU



Streams - Howto

Declare

```
cudaStream_t stream[N]; // array of N streams
```

Initialize

```
for (int k=0;k<N;k++) cudaStreamCreate(&stream[k]);
```

Launch

```
for (int k=0;k<N;k++) { // do 1/Nth of the work
  // remember to use pinned host memory
  cudaMemcpyAsync(... cudaMemcpyHostToDevice, stream[k]);
  kernel <<<nbB, nbT, 0, stream[k] >>>(...);
  cudaMemcpyAsync(... cudaMemcpyDeviceToHost, stream[k]);
}
```

Destroy

```
for (int k=0;k<N;k++) cudaStreamDestroy(stream[k]);
```

Memory Model

Registers
- Per thread
- Data lifetime = thread lifetime

Shared memory
- Per thread block on-chip memory
- Data lifetime = block lifetime

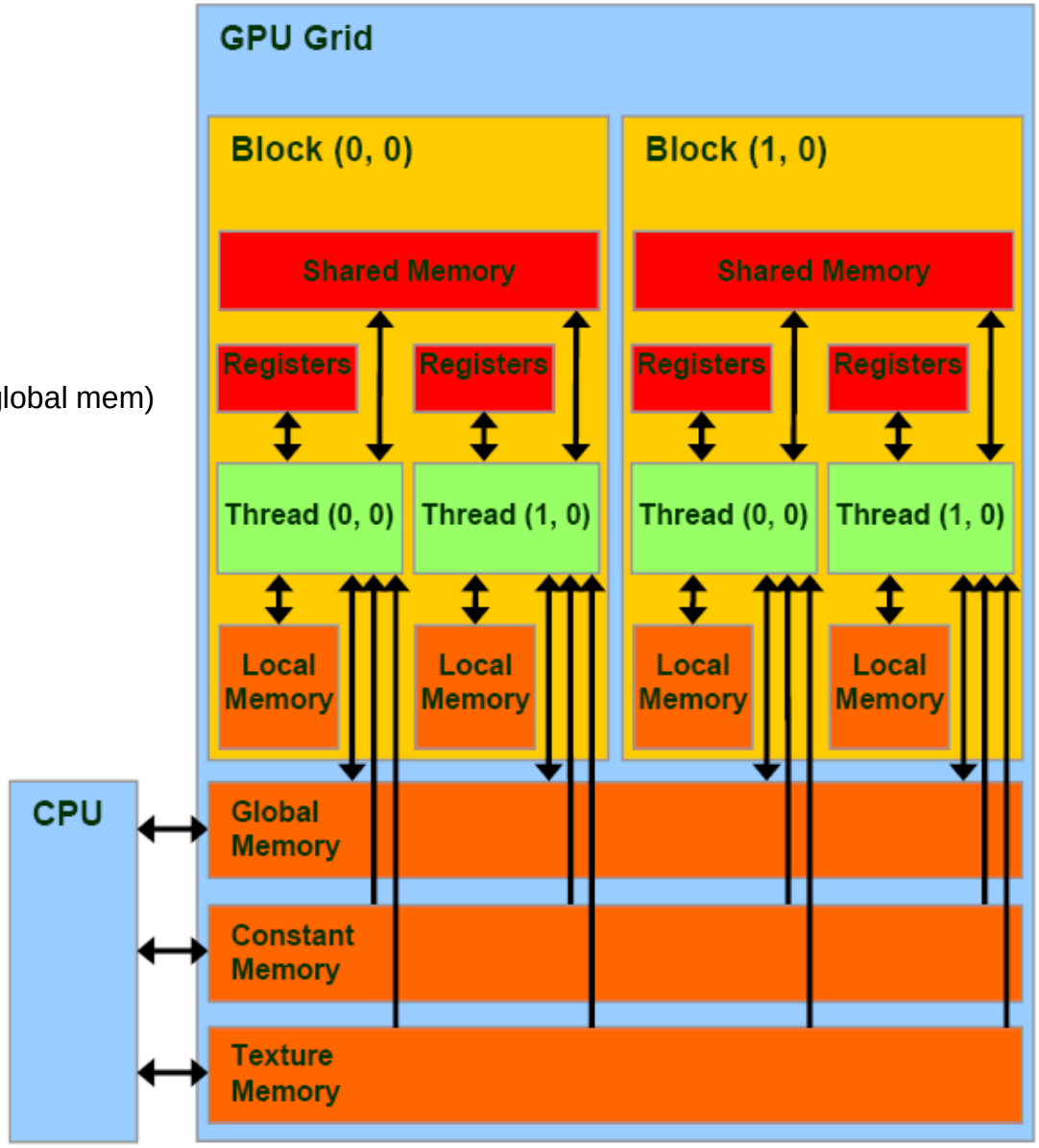
Local memory
- Per thread off-chip memory (physically part of global mem)
- Data lifetime = thread lifetime

Global memory
- Slow and uncached(1.0), cached(2.0)
- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation

Constant memory
- Read only, cached
- Short latency and high bandwidth when all threads access the same location

Texture memory
- Read-only, cached
- Accessible by all threads as well as host (CPU)
- Data lifetime = from allocation to deallocation
- Some nice features (wrap modes, etc)

CPU memory ("host memory")
- Not directly accessible by CUDA threads



Local memory

- Physically a part of the global memory
- Should rather be called thread-local global memory
- Slow & uncached
- Visible only to one single thread
- Used to store whatever does not fit into the registers: the compiler makes use of local memory when it determines that there is not enough register space to hold your variables

Constant memory

- Physically a part of the global memory
- Read-only but cached
- Cache hits do not need to re-access the global memory
- Size needs to be known at compile time
- Use Constant memory in the following cases:
  - Data is read-only (will not change during the execution)
  - Many threads have to access the same data

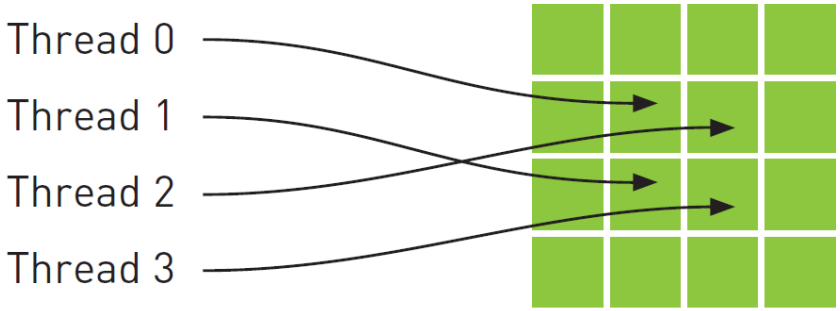
- Use cudaMemcpyToSymbol() to copy from host memory to constant memory on the GPU.

Constant memory: example

```
define SIZE 512
//declare constant memory
__constant__ float cm[SIZE]; // no cudaMalloc!

__global__ void mykernel (float* d_in, float* d_out)
{
	int index = blockIdx.x*blockDim.x + threadIdx.x;
	float x = d_in[index];
	for (int j=0; j<SIZE; j++)
		x += cm[j];
	d_out[index]=x;
}

int main()
{
	...
	cudaMemcpyToSymbol (cm, hostdata, SIZE*sizeof(float));
	...
	mykernel<<<N/M,M>>>(d_in, d_out);
	...
}
```

Texture memory

- Like constant memory, texture memory is read-only cached memory that can improve performance and reduce memory traffic.
- The cache is optimized for 2D spatial locality:



- Arithmetically, the four addresses shown are not consecutive, but the texture cache is designed to accelerate access patterns such as this one.

Data access performance

1st place: Registers
2nd place: Shared Memory
3rd place: Constant Memory
4th place: Texture Memory
Tie for last place: Local Memory and Global Memory

Exercise

Time three kernels that access memory values a huge number of times. The values are stored in

```
#define SIZE 4096
__constant__ float d_idata_const[SIZE];
```

a) the global memory

```
__global__ void testglobalmem(float* d_idata, float* d_odata) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      float x = 0;
      for (int j = 0; j<SIZE; j++) x += d_idata[j];
      d_odata[i] = x;
}
```

b) the constant memory

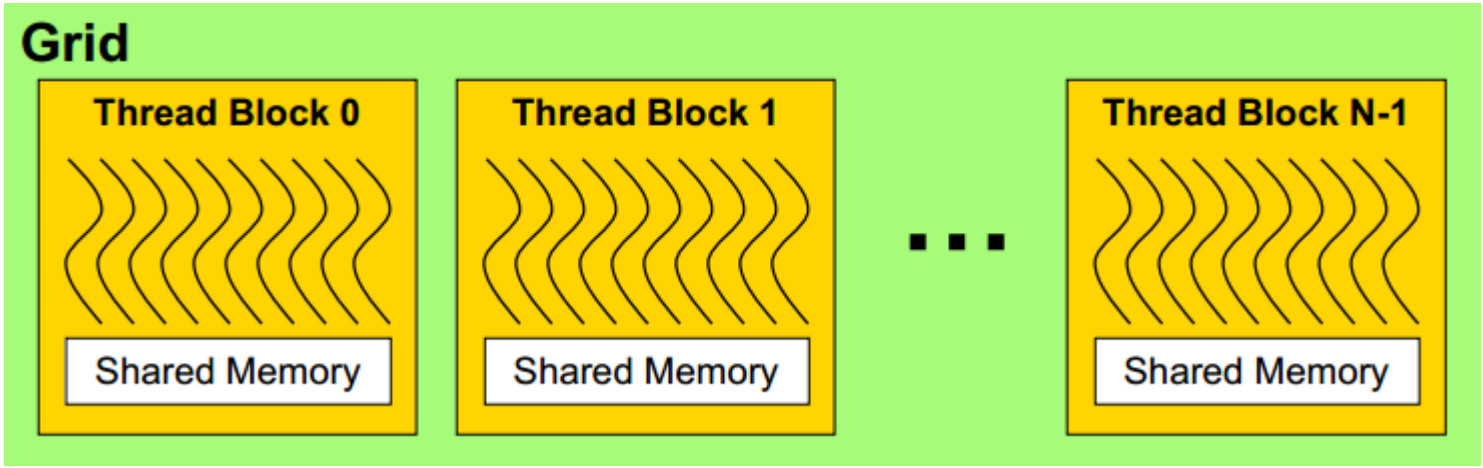
```
__global__ void testconstmem(float* d_odata) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      float x = 0;
      for (int j = 0; j<SIZE; j++) x += d_idata_const[j];
      d_odata[i] = x;
}
```

c) a register (kernel parameter)

```
__global__ void testregister(float f, float* d_odata) {
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      float x = 0;
      for (int j = 0; j<SIZE; j++) x += f;
      d_odata[i] = x;
}
```

Shared Memory

- Extremely fast, on-chip memory
- Not visible to threads in other blocks running in parallel
- “User-managed cache”



Shared Memory

Size known at compile time:

```
__global__ void kernel(...) {
      __shared__ float sm[SIZE];
      ...
}
int main(void) {
      kernel<<<N,M>>>(...);
}
```

Size known at run time:

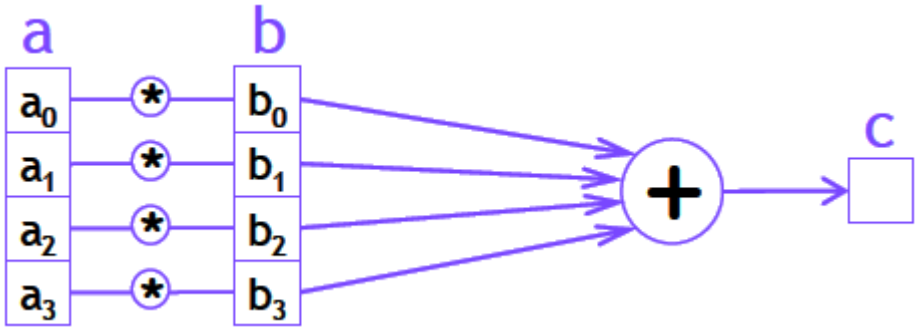
```
__global__ void kernel(...) {
      extern __shared__ float sm[];
      ...
}
int main(void) {
      smBytes = SIZE*sizeof(float);
      kernel<<<N,M,smBytes>>>(...);
}
```

Thread Cooperation

- A block of threads can cooperate via shared memory and synchronization.
- Threads may want to
  - Share results to avoid redundant computation
    - Case study: dot product
  - Cooperate on memory accesses (bandwidth reduction)
    - Case study: matrix multiplication

Case study: Dot Product

- Unlike vector addition, dot product is a reduction from two vectors to a scalar.




```
__global__ void dot (float *a, float *b, float *c) {
    // Each thread computes a pairwise product
    float temp = a[threadIdx.x] * b[threadIdx.x];
}
```

Can't compute the final sum:
Each thread's copy of "temp" is private. What can we do?

Case study: Dot Product

```
#define N 512

__global__ void dot (float *a, float *b, float *c) {
  __shared__ float sm[N];
  sm[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

  // thread 0 sums the pairwise products
  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<N; i++)
      sum += sm[i];
    *c = sum;
  }
}
```

Case study: Dot Product

```
#define N 512

__global__ void dot (float *a, float *b, float *c) {
  __shared__ float sm[N];
  sm[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

  // thread 0 sums the pairwise products
  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<N; i++)
      sum += sm[i];
    *c = sum;
  }
}
```

Attention: Read-before-Write Hazard!

Thread 0 might start summing up before other threads write their product!

Case study: Dot Product

```
#define N 512

__global__ void dot (float *a, float *b, float *c) {
  __shared__ float sm[N];
  sm[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];

  __syncthreads();

 // thread 0 sums the pairwise products
  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<N; i++)
      sum += sm[i];
    *c = sum;
  }
}
```

Threads in the block wait until all threads have hit the __syncthreads() barrier. Threads are only synchronized within a block.

Exercise

Implement the GPU dot product and compare its result to a CPU version (« golden model »).

Scalable dot product

How can we handle dot products when the size of the vector exceeds 512?

Scalable dot product (1)

Looping threads for one block:

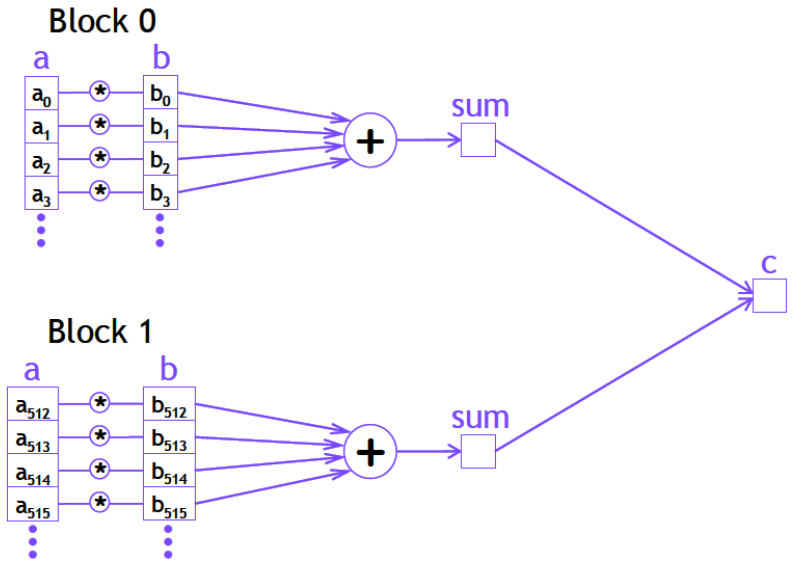
```
#define N (4*1024*1024)
#define NBTHREADS 512

__global__ void dot (float *a, float *b, float *c) {
  __shared__ float sm[NBTHREADS];
  sm[threadIdx.x] = 0;
  for (int i=threadIdx.x; i<N; i+=blockDim.x)
    sm[threadIdx.x] += a[i] * b[i];

  __syncthreads();

  // thread 0 sums the pairwise products
  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<NBTHREADS; i++)
      sum += sm[i];
    *c = sum;
  }
}
```

But a single block will not saturate the GPU…

Scalable dot product (2)

Multiple blocks

- Each block computes a partial dot product
- And then contributes its sum to the final result



Scalable dot product (2)

Idea

Running the final reduction on the host.

```
#define N (4*1024*1024)
#define NBTHREADS 512

int main {
 ...
 dot_tmp<<< N/NBTHREADS,NBTHREADS >>>(a,b,d_dot);
 // you can also choose a hybrid solution: less blocks with more loops

 cudaMemcpy(dot, d_dot, size, cudaMemcpyDeviceToHost));
 c = CPU_dot_finalize(dot);
 ...
 return 0;
}
```

Scalable dot product (3)

Another idea

Can the blocks add themselves up?

```
#define N (4*1024*1024)
#define NBTHREADS 512

__global__ void dot(float *a, float *b, float *c) {
  __shared__ float sm[NBTHREADS];
  sm[threadIdx.x] = 0;
  int index = blockIdx.x*blockDim.x+threadIdx.x;
  for (int i=index; i<N; i+=blockDim.x*gridDim.x)
    sm[threadIdx.x] += a[i] * b[i];

  __syncthreads();

  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<NBTHREADS; i++)
      sum += sm[i];
    *c += sum;
  }
}
```

Scalable dot product (3)

Another idea

Can the blocks add themselves up?

```
#define N (4*1024*1024)
#define NBTHREADS 512

__global__ void dot(float *a, float *b, float *c) {
  __shared__ float sm[NBTHREADS];
  sm[threadIdx.x] = 0;
  int index = blockIdx.x*blockDim.x+threadIdx.x;
  for (int i=index; i<N; i+=blockDim.x*gridDim.x)
    sm[threadIdx.x] += a[i] * b[i];

  __syncthreads();

  if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<NBTHREADS; i++)
      sum += sm[i];
    *c += sum;
  }
}
```

But we have a race condition!

Race condition

- Program behavior depends upon relative timing of two or more event sequences
- What actually happens to execute the line `*c += sum;`
  - Read value at address c
  - Add sum to value
  - Write result to address c
- What if two threads are trying to do this at the same time?

Scalable dot product (3)

```
#define N (4*1024*1024)
#define NBTHREADS 512

__global__ void dot(float *a, float *b, float *c) {
  __shared__ float sm[NBTHREADS];
  sm[threadIdx.x] = 0;
  int index = blockIdx.x*blockDim.x+threadIdx.x;
  for (int i=index; i<N; i+=blockDim.x*gridDim.x)
    sm[threadIdx.x] += a[i] * b[i];

  __syncthreads();

 if (threadIdx.x==0) {
    float sum = 0;
    for (int i= 0; i<NBTHREADS; i++)
      sum += sm[i];
    atomicAdd(c, sum);
  }
}
```

Atomic operations

Read-modify-write uninterruptible

- CUDA 1.1 or higher
- Ensure correct results when multiple threads modify the same memory space,
- Several atomic operations are available :
  - atomicAdd()
  - atomicSub()
  - atomicMin()
  - atomicMax()
  - atomicInc()
  - atomicDec()
  - atomicExch()
  - …

But

- Execution order is not defined
- Atomics are slower than normal accesses, especially when many threads attempt to perform atomic operations on a small number of locations

Avoid atomics if possible and try to use shared memory and structured algorithms.

Exercise

Implement the scalable dot product (3) with looping threads and atomic additions.
Compare different combinations of nbBlocks/nbLoops:

```
datasize = 32*1024*1024
nbthreads = 512
N = datasize/nbthreads
```

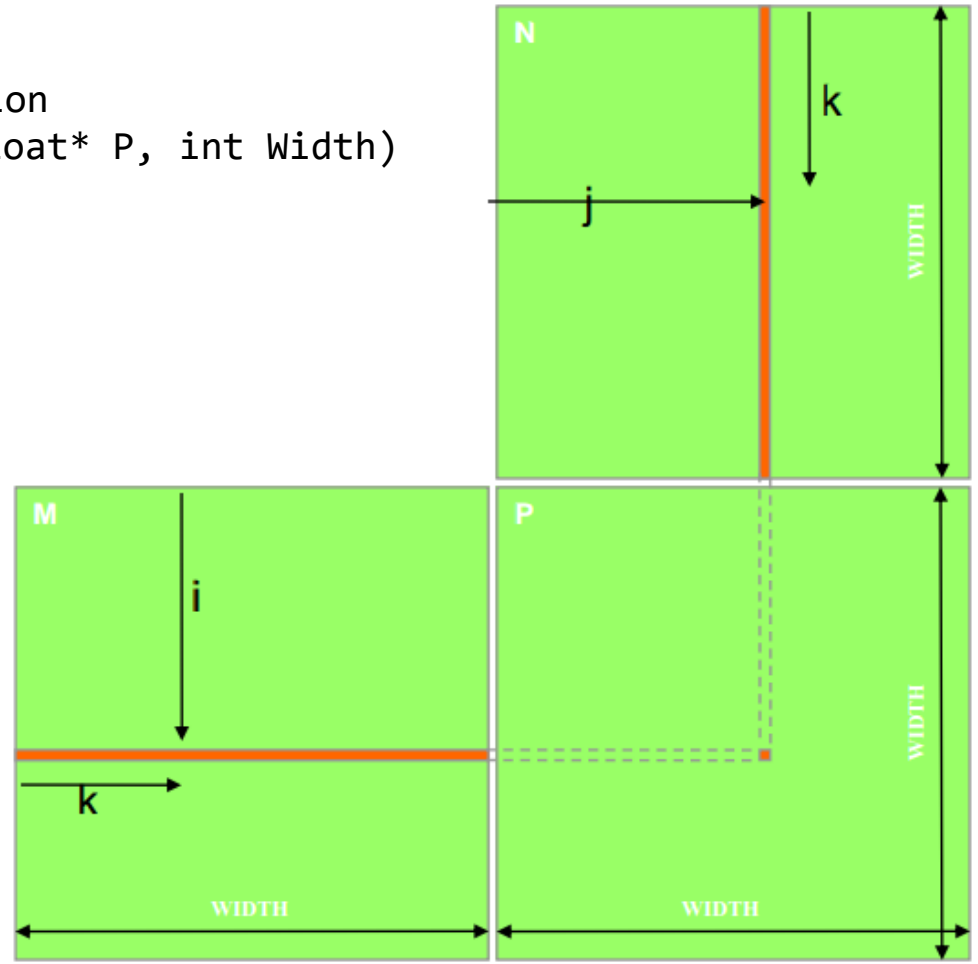
- 1 block, N loops
- 2 blocks, N/2 loops
- … (hybrid solutions)
- N/2 blocks, 2 loops
- N blocks, 1 loop

Which combination works best?

Case study: Matrix multiplication

P = M * N of size WIDTH x WIDTH

```
// CPU version of matrix multiplication
void matrixMul(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;
    }
}
```



Exercise

Implement an equivalent GPU matrix multiplication. For now, we consider only small matrices that fit into one block (512 threads, so max width = 22)

Compare its result and its performance to the CPU version. To estimate the error, use the root-mean-square deviation:

```
float comparef (float* a, float* b, int n) {
  double diff = 0.0;
  for (int i=0;i<n;i++)
    diff += (a[i]-b[i])*(a[i]-b[i]);
    return (float)(sqrt(diff/(double)n));
}
```

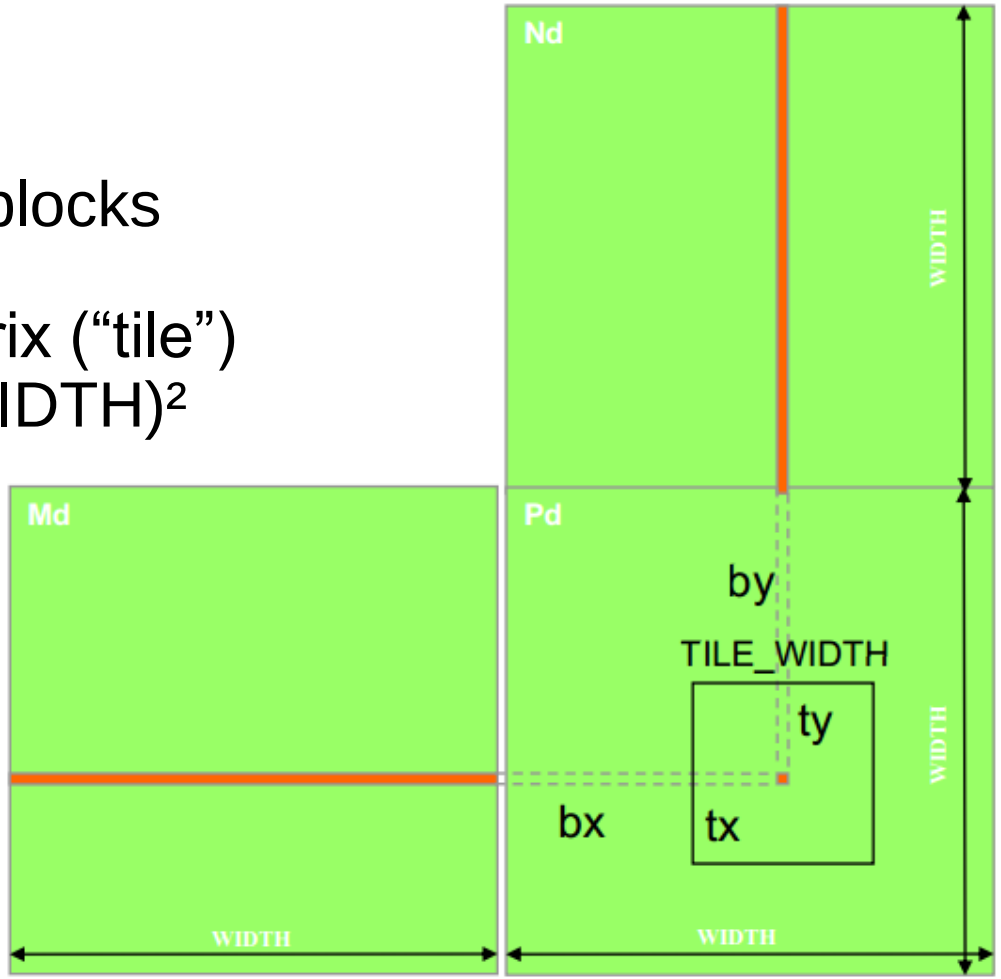
Scalable Matrix multiplication

How about big matrices?

Scalable Matrix multiplication

Solution: Tiling

- Generate a 2D Grid of (WIDTH/TILE_WIDTH)² blocks
- Each block computes a (TILE_WIDTH)² sub-matrix (“tile”)
- Each block has (TILE_WIDTH)² threads



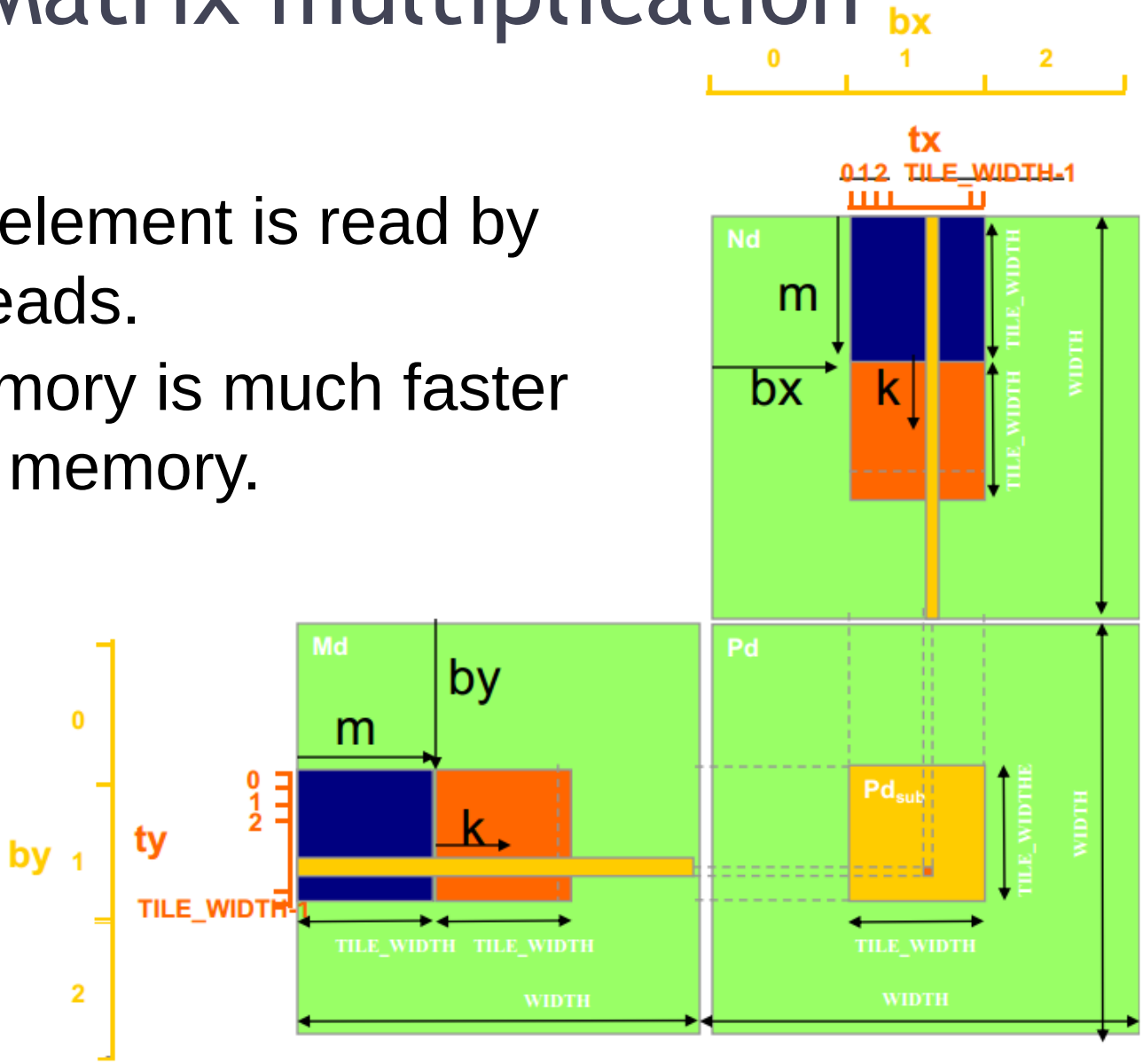
Exercise

Implement the tiled GPU matrix multiplication.

Compare its result and performance to the CPU version.

Scalable Matrix multiplication

- Each input element is read by WIDTH threads.
- Shared memory is much faster than global memory.



Scalable Matrix multiplication

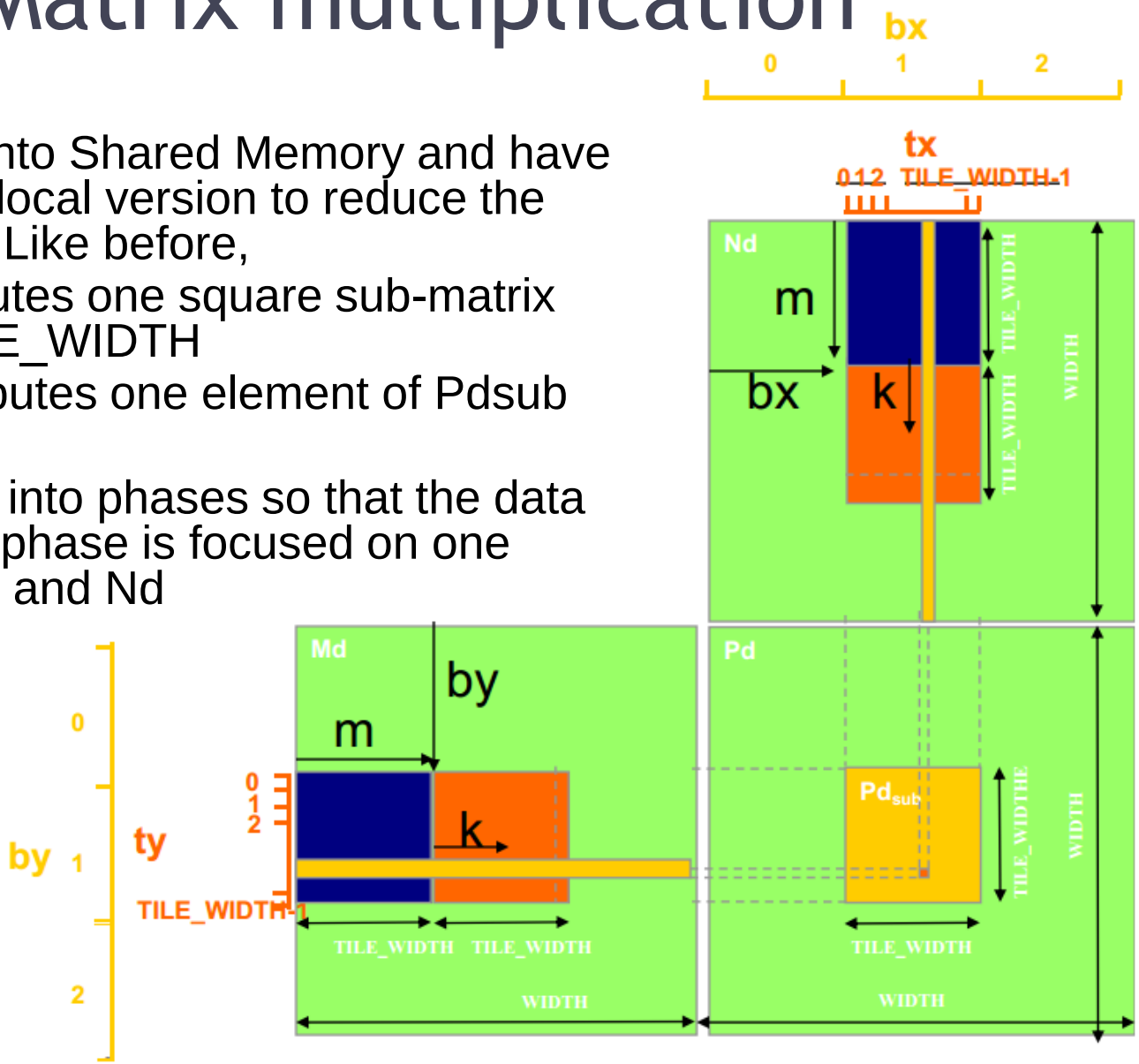
Idea

Load the elements into Shared Memory and have the threads use the local version to reduce the memory bandwidth. Like before,

- Each block computes one square sub-matrix Pdsub of size TILE_WIDTH
- Each thread computes one element of Pdsub

BUT

- Break up the sum into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



```
__global__ void matrixMul(float* Md, float* Nd, float* Pd, int Width)
{
  __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
  __shared__float Nds[TILE_WIDTH][TILE_WIDTH];
  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
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;

  float Pvalue = 0;
  // loop over the Md and Nd tiles required to compute the Pd element
  for (int m = 0; m < Width/TILE_WIDTH; ++m)
  {
    // collaborative loading of Md and Nd tiles into shared memory
    Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
    __syncthreads();

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

Exercise

Implement GPU matrix multiplication using shared memory.