

CUDA Programming Exercises

Scott B Baden & Didem Unat

May 4, 2010



UC San Diego

[**simula** . research laboratory]

This afternoon?

- Compiling CUDA programs
- Lab #1 : Increment Array
 - Simple CUDA Kernel
 - Kernel Launch and Kernel Parameters
 - Host Memory Management
 - Device Memory Allocation
 - Host-Device Memory Copy
- Lab #2 : Matrix Multiplication
 - 2D Blocks
 - Shared memory utilization
 - Registers

GPU Machines at Simula

Ssh to machine called: **oslo.simula.no**

Then, login to one of these machines

- If your user ID 1-3
 - **gpu-1.ndlab.net**
 - GeForce 8800GT (G92) - Compute 1.1 - CUDA 3.0
- If your user ID 4-6
 - **gpu-2.ndlab.net**
 - GeForce 8800GT (G92) - Compute 1.1 - CUDA 3.0
- If your user ID 7-9
 - **gpu-3.ndlab.net**
 - GeForce 8800GT (G92) - Compute 1.1 - CUDA 3.0
- If your user ID 10-12
 - **gpu-4.ndlab.net**
 - GeForce 8800GT (G92) - Compute 1.1 - CUDA 3.0

Before you start

- Add these lines to the *.bashrc* or *.bash_profile*

Export Path and Library Path for CUDA

```
PATH=$PATH:/usr/local/cuda/bin
```

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64
```

```
export PATH
```

```
export LD_LIBRARY_PATH
```

Compiling CUDA

- Any source file containing CUDA language code must be compiled with NVCC
 - It separates code running on the host from the code running on the GPU
 - Creates PTX (Parallel Thread eXecution) and CPU code
 - Then PTX code is compiled into device-specific binary object
- Device Emulation Mode
 - Nvcc -deviceemu
 - Runs on the host using the CUDA runtime
 - Useful for debugging
 - Executes sequentially

Lab #1: Increment Array

- Code in
`/home/dunat/public/incrArr`
- Simple CUDA Kernel
- Kernel Launch and Kernel Parameters
- Host Memory Management
 - Device Memory Allocation
 - Host-Device Memory Copy

Lab #1 : Increment Array

```
void increment_Array_On_Host(float* A, int N)
{
    int i;
    for (i=0; i< N; i++)
        A[i] = A[i] + 1.f;
}
```

On Host

```
__global__ void increment_On_Device(float *A, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N)
        A[idx] = A[idx]+1.0f;
}
```

On GPU

Lab #1 : Kernel Launch

```
void increment_Array_On_Host(float* A, int N)
{
    int i;
    for (i=0; i< N; i++)
        A[i] = A[i] + 1.f;
}
```

Function Call:

```
increment_Array_On_Host(A, N);
```

On Host

```
__global__ void increment_Array_On_Device(float *A, int N)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if (idx<N)
```

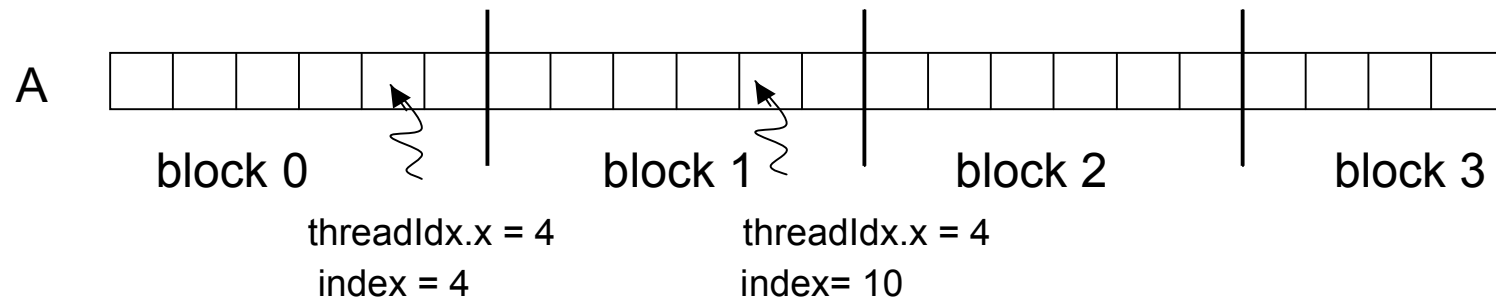
Kernel Call:

```
increment_Array_On_Device<<<nBlocks, blockSize>>> (A_d, N);
```

On GPU

Lab #1 : Thread Assignment

- Each thread increments one element in A
- A thread uses *block Id*, *block size* and *thread Id* parameters to determine its assignment.
- `__global__` keyword precedes the kernel declarations

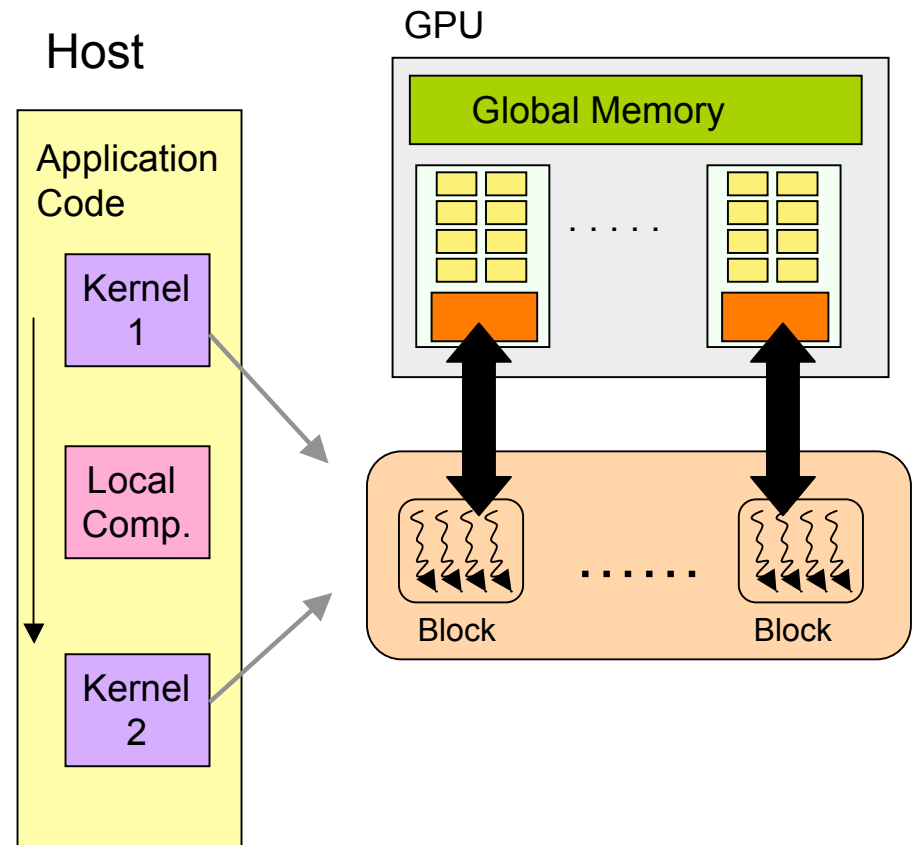


```
__global__ void increment_On_Device(float *A, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        A[idx] = A[idx] + 1.f;
}
```

On GPU

Lab #1: Managing Memory

- *Increment Array* Kernel is offloaded to the GPU
- Array *A* resides on the main memory NOT in the device memory.
 - Allocate storage on the device memory
 - Copy data over the device before kernel launch



Lab #1: Managing Memory on the Host

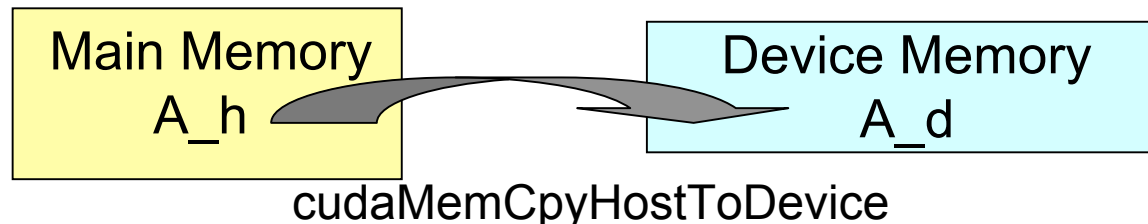
```
float *A_h, *B_h;      // pointers to host memory
float *A_d;           // pointer to device memory
```

```
//Allocate memory on the device for A
cudaMalloc((void **) &A_d, size);
```

```
for (i=0; i<N; i++)
    A_h[i] = (float)i; // init host data
```

```
//Copy host array to the device array
cudaMemcpy(a_d, a_h, sizeof(float)*N,
           cudaMemcpyHostToDevice);
```

```
//set block size
int bSize = 64;
//compute number of blocks
int nBlocks = N/bSize + (N%bSize == 0?0:1);
```



Lab #1 : Kernel Launch on the Host

```
//kernel launch  
Increment_On_Device <<< nBlocks, bSize >>> (A_d, N);
```

```
//synchronize to make sure kernel is complete  
cudaThreadSynchronize();
```

```
// Retrieve result from device and store in B_h  
cudaMemcpy(B_h, A_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
```

```
// check results  
for (i=0; i<N; i++)  
    assert(A_h[i] == A_d[i]);
```

```
// cleanup  
free(A_h);  
free(A_h);  
cudaFree(A_d);
```

Lab #1 : Copy back to the Host

```
//kernel launch
Increment_On_Device <<< nBlocks, bSize >>> (A_d, N);

//synchronize to make sure kernel is complete
cudaThreadSynchronize();

// Retrieve result from device and store in B_h
cudaMemcpy(B_h, A_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++)
    assert(A_h[i] == A_d[i]);

// cleanup
free(A_h);
free(A_h);
cudaFree(A_d);
```

Excercises-1

- Run the increment array benchmark
- Report performance on the GPU for
 - Various # repetitions
 - Compare performance with the host
- Determine the data transfer rate between host and node.
- For the fixed block size, and a fixed value of N ,
 - increase the number of repetitions (by doubling) until the running time per point stabilizes.
- Next, run for 1 iteration. Using these two timings, solve for the transfer rate.

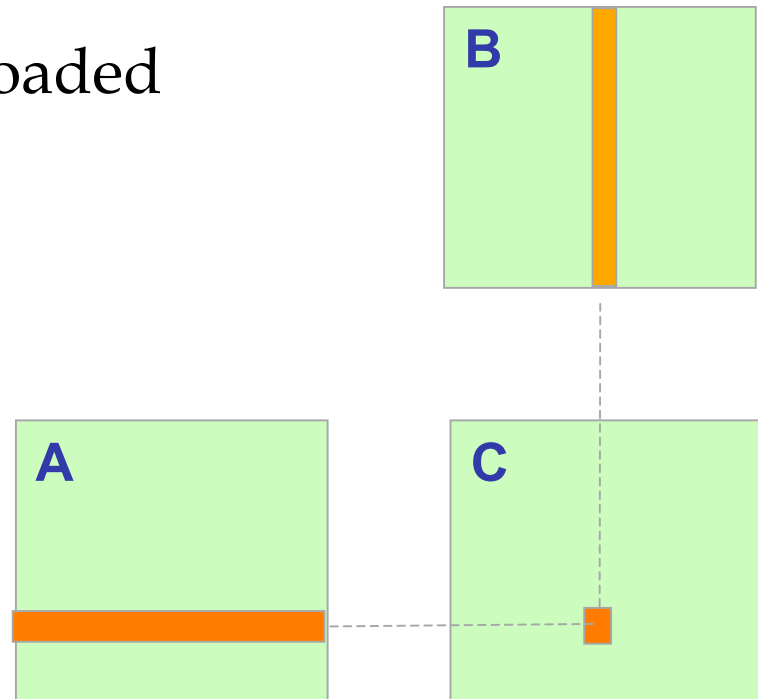
Lab #2 : Matrix Multiply

- code in /home/dunat/public/matrixMul
- $C = A * B$ (N by N square matrices)
- Naïve Host Code

```
for (unsigned int i = 0; i<N; i++){  
    for (unsigned int j = 0; j<N; j++) {  
        float sum = 0;  
  
        for (unsigned int k = 0; k<n; k++)  
            sum += A[i * n + k] * B[k * n + j];  
  
        C[i * n + j] = (float) sum;  
    }  
}
```

Lab #2 : Naïve Matrix Multiply

- Each thread computes one element of C
 - Loads a row of matrix A
 - Loads a column of matrix B
 - Computes a dot product
- Every value of A and B is loaded N times from global memory



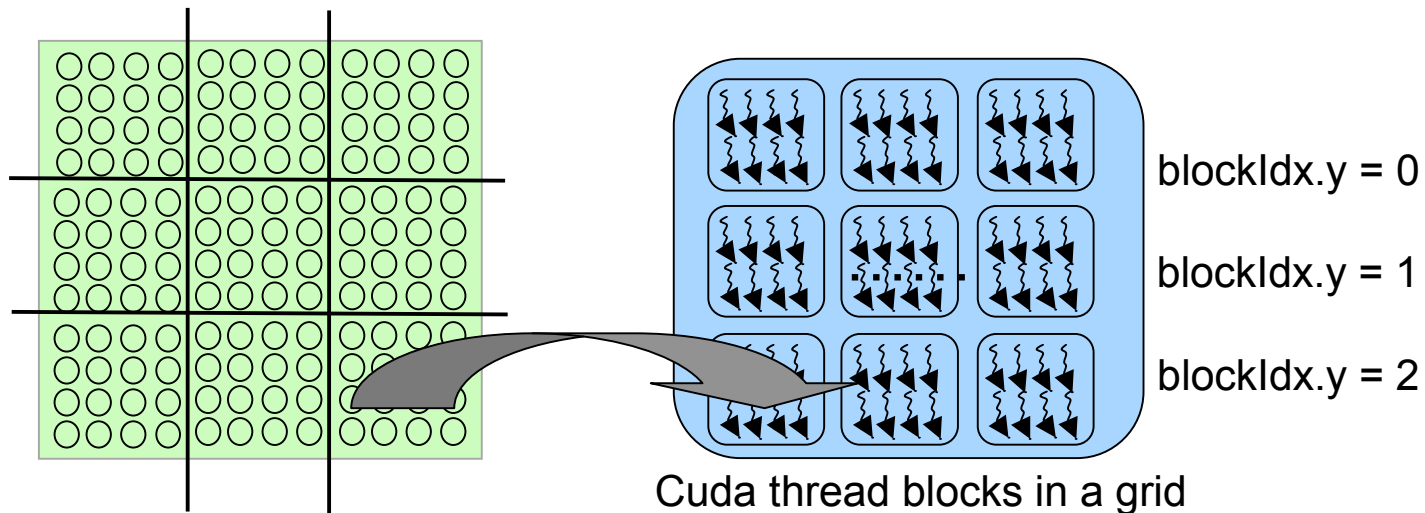
Lab #2 : Device Kernel Function

```
__global__ void  
matMul( float* C, float* A, float* B) {  
  
    int I = blockIdx.x*blockDim.x + threadIdx.x;  
    int J = blockIdx.y*blockDim.y + threadIdx.y;  
    int N = blockDim.y*gridDim.y; // Assume a square matrix  
  
    if ((I < N) && (J < N)){  
        float _c = 0;  
        for (unsigned int k = 0; k < N; k++) { // perform dot product  
            float a = A[I * N + k];  
            float b = B[k * N + J];  
            _c += a * b;  
        }  
        C[I * N + J] = _c;  
    }  
}
```

Lab #2 : 2D Thread Blocks

```
__global__ void  
matMul( float* C, float* A, float* B) {  
  
  Int I = blockIdx.x*blockDim.x + threadIdx.x;  
  Int J = blockIdx.y*blockDim.y + threadIdx.y;  
  Int N = blockDim.y*gridDim.y; // Assume a square matrix
```

- A thread computes global indices from local indices (thread Id x and y) and block parameters
- *Array C* is divided into 2D thread blocks and a thread block consists of 2D threads.



Lab #2 : Code stub on host side

```
unsigned int n2 = n*n*sizeof(FLOAT);  
FLOAT *h_A = (FLOAT*) malloc(n2), *h_B = (FLOAT*) malloc(n2);  
  
// Check that allocations went OK  
assert(h_A); assert(h_B);  
  
genMatrix(h_A, n, n); genMatrix(h_B, n, n); // Initialize matrices  
  
FLOAT *d_A, *d_B, *d_C;  
cudaMalloc((void**) &d_A, n2); ... &d_A ... &d_B  
checkCUDAError("Error allocating device memory arrays");  
  
// copy host memory to device  
cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);  
cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);  
checkCUDAError("Error copying data to device");
```

Lab #2 : Code stub on host side (cont.)

```
// setup execution configurations
dim3 threads(ntx, nty, 1);
dim3 grid(n / threads.x, n / threads.y);

// launch the kernel
matMul<<< grid, threads >>>(d_C, d_A, d_B);

//synchronize to make sure kernel is complete
cudaThreadSynchronize();
// retrieve result
cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);
checkCUDAError("Unable to retrieve result from device");

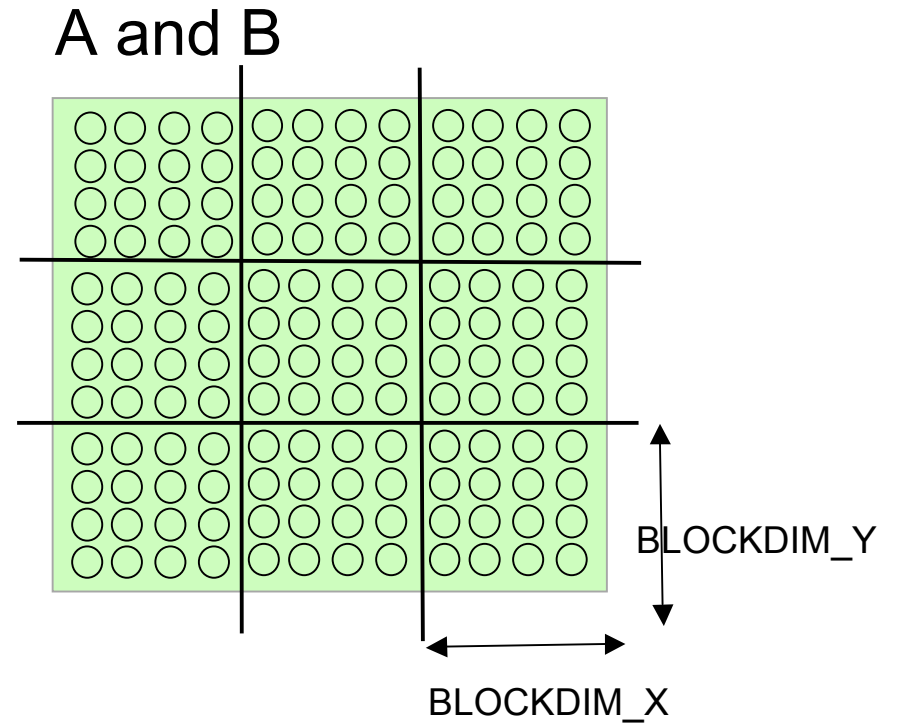
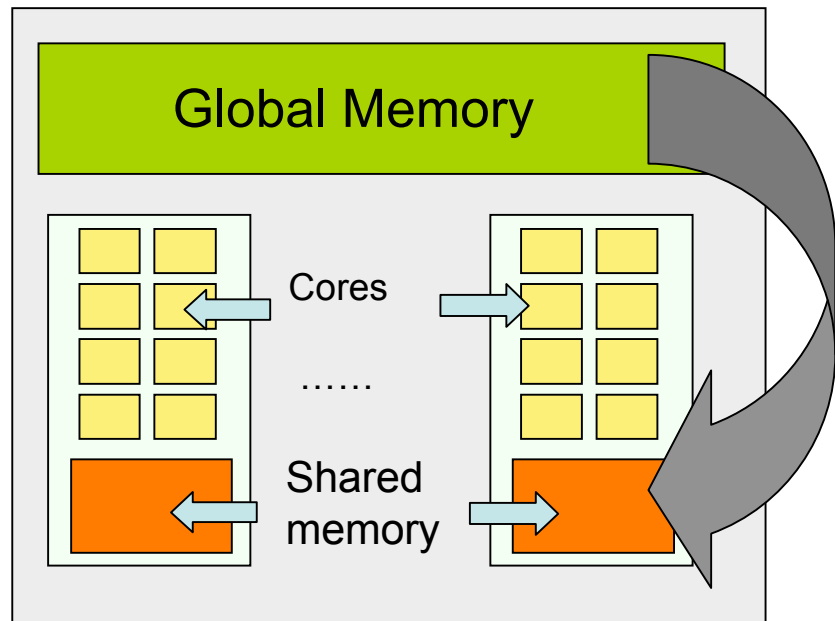
// Free device storage
assert(cudaSuccess ==cudaFree(d_A));
assert(cudaSuccess ==cudaFree(d_B));
assert(cudaSuccess ==cudaFree(d_C));
```

Lab #2 :Performance Improvement

- Problems with the naïve implementation
 - Redundant global memory accesses
 - Each value of A and B are read N times
- Improve the performance with **shared memory**

Shared Memory Blocks

- A and B are in global memory
- Threads cooperate to read a tile size of BLOCKDIM_X and BLOCKDIM_Y into shared memory.



Shared Memory Blocks (cont.)

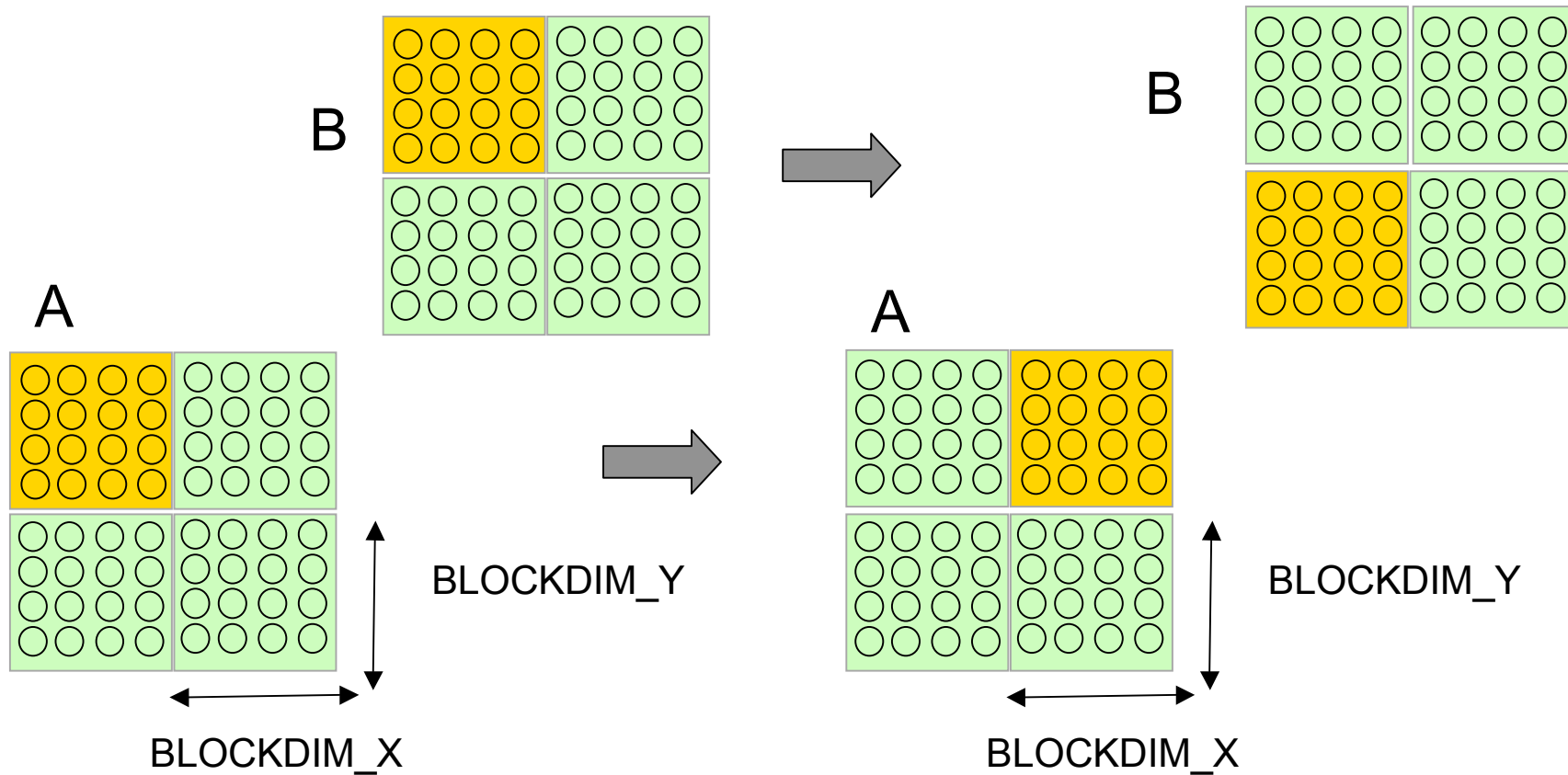
- Define Shared Memory blocks for A and B arrays

```
__shared__ float Ablock[BLOCKDIM_Y][BLOCKDIM_X],
```

```
__shared__ float Bblock[BLOCKDIM_Y][BLOCKDIM_X];
```

- Now, each thread block has a copy of *Ablock* and *Bblock*.
 - Note that data in shared memory is accessible by threads only in the same block.
- Create 2D thread blocks so that each thread loads 1 element of A and element of B in a tile

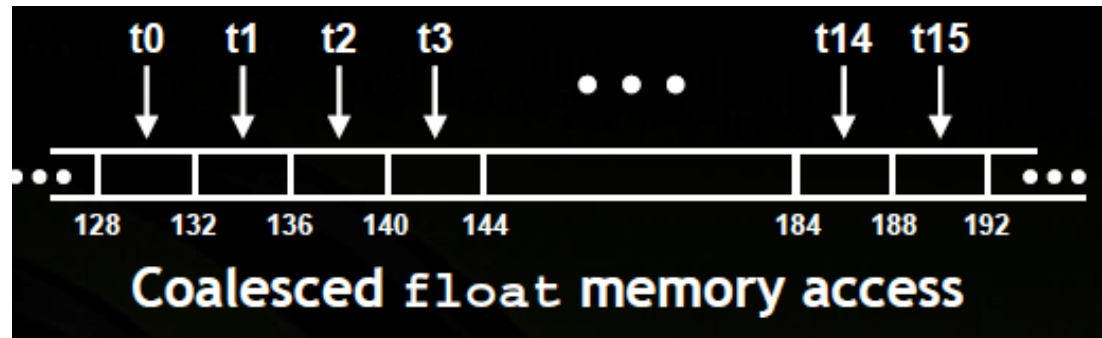
Shared Memory Blocks (cont.)



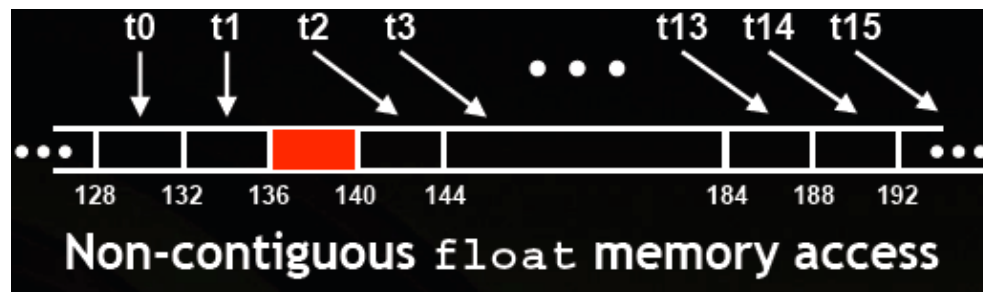
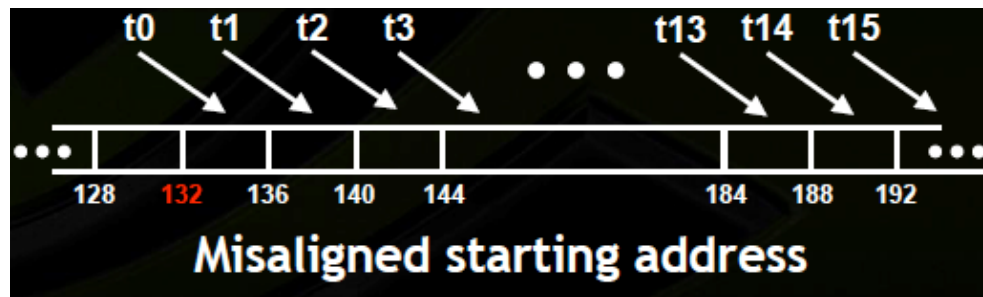
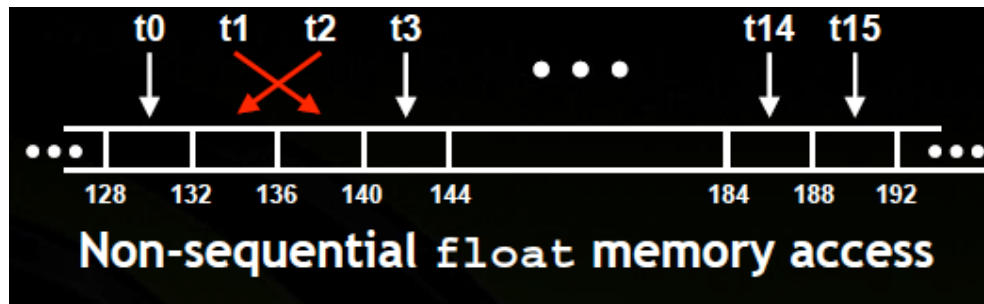
For each $(N/BLOCKDIM_X)$ // number of blocks
Each thread brings one element in a tile of
A and B into shared memory

Memory Coalescing

- Global memory accessed will be coalesced into a single access if:
 - The size of the memory element accessed by each thread is either 4,8, or 16 bytes
 - The elements form a contiguous block of memory
 - The Nth element is accessed by the Nth thread in the half-warp
 - The address of the first element is aligned to 16 times the element's size.



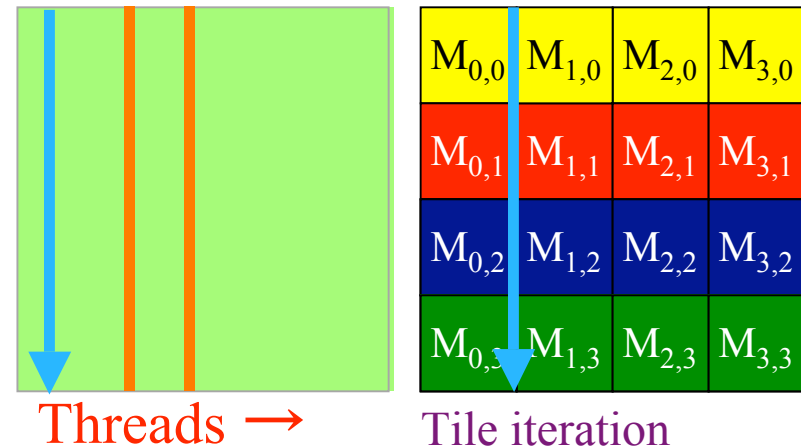
Non-coalesced Memory Accesses



Coalesced Memory Accesses

- Memory banks: consecutive addresses across threads can be read very quickly
- Accesses organized by half warps (16 threads)

Uncoalesced	8.07
Coalesced	83.8



Nvidia Corp.

Lab #2 : Performance Comparison

- On Tesla C1060
- $N = 512$
- Single threaded host (unblocked): 0.60 Gflop/s
- Different thread geometries

Geometry	16x16	8 × 8	4 × 4
Uncoalesced	9.20	8.90	8.26
Coalesced	128	53.8	12.5

Naive

Geometry	2 × 256	2 × 128	2 × 64	2 × 32	4 × 128	4 × 64
Gflops	8.85	7.69	6.36	6.30	4.61	3.96

Synchronization

```
__shared__ float Ablock[BLOCKDIM_Y][BLOCKDIM_X],
__shared__ float Bblock[BLOCKDIM_Y][BLOCKDIM_X];

//compute global indices from thread id and block ids

For each N/BLOCKDIM_X //number of blocks
{
    __syncthreads();

    //read an element of a tile of A and B into shared memory

    __syncthreads();
    .....
}
```

- Synchronize before accessing shared memory to make sure all threads have completed
- No global synchronization- only threads within a thread block synchronize.

Pseudo-code

```
__shared__ float Ablock[BLOCKDIM_Y][BLOCKDIM_X],
__shared__ float Bblock[BLOCKDIM_Y][BLOCKDIM_X];

//compute global indices from thread id and block ids

Float c;

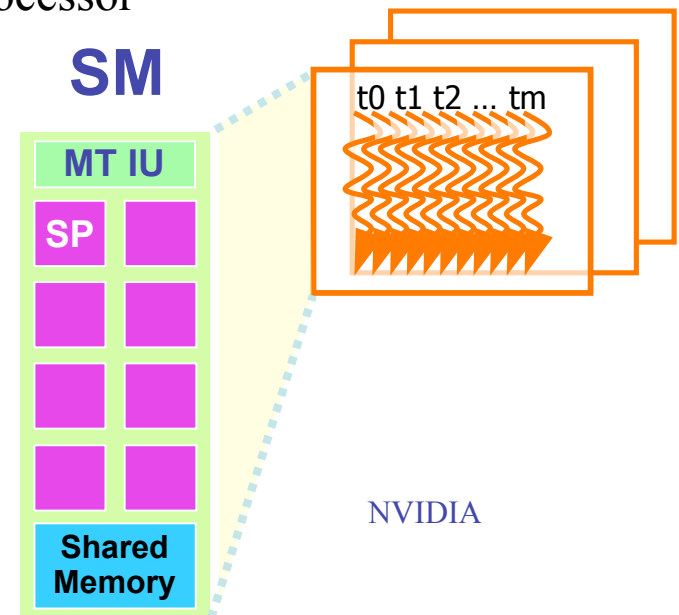
For each N/BLOCKDIM_X //number of blocks {
    __syncthreads();
    //read an element of a block in A into shared memory
    //read an element of a block in B into shared memory
    __syncthreads();

    For each element in a block
        c+= Ablock[ ][ ] * Bblock[ ][ ];
}

C[ ] = c; // store c in global memory
```

Device Occupancy

- Device occupancy
 - Ratio of the number of active warps to the maximum number of warps supported by a multiprocessor.
- Limited by
 - Amount of shared memory / multiprocessor
 - Number of registers / multiprocessor
 - Maximum number of thread blocks / multiprocessor
- In MatrixMultiply, block size determines
 - Amount of shared memory
 - Number of threads used.
 - As a result, number of registers



Performance Goal

- Using block size of 16×16
- Try to achieve >50 GFlops for $N=1024$



Thank you

- Questions?

Coalesced Memory Accesses

- Memory banks: consecutive addresses across threads can be read very quickly
- Accesses organized by half warps (16 threads)

```
I = blockIdx.y*by + ty;
J = blockIdx.x*bx + tx;
```

```
Ablock[ty][tx] = A[I*N+k*by+tx];
Bblock[ty][tx] = B[J+N*(k*bx+ty)]
```



Nvidia Corp.

Naïvely Blocked

Naïvely blocked:

```
I = blockIdx.x*bx + tx;
```

```
J = blockIdx.y*by + ty;
```

```
a[tx][ty] = A[I*N+k*by+ty];
```

```
b[ty][tx] = B[J+N*(k*bx+tx)];
```

Performance on Tesla C1060	Gflop/s
Coalesced	128
Naively Blocked	9.20

Kernel Parameters

- Dynamic partitioning → underutilized resources
- Blocked matrix multiply: 10 registers
- 16×16 blocks: 2560 registers → 6 blocks
 - 1024 register limit → 4 blocks
(128 Gflops on Tesla C1060)
- 8×8 blocks: 640 registers → 25 blocks
 - 8 block limit → 512 threads
(53.8 Gflops on Tesla C1060)