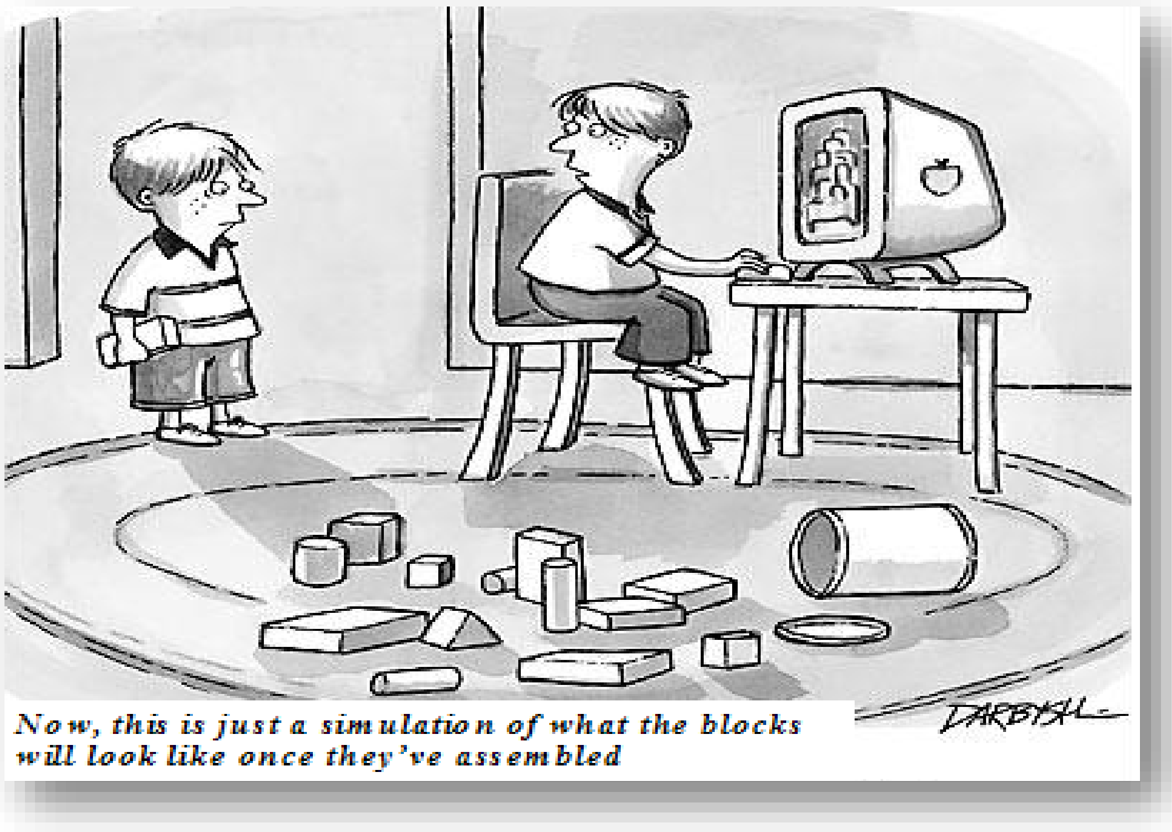
ECE569
Module 20



- Thread Divergence

Memory Model Revised

- **Thread**
  - Local memory
- **Threads in a Block**
  - Shared memory
    - __synchthreads()
- **Kernel: Thread Blocks**
  - Global memory
  - Between two kernel launches
    - Implicit barrier

Writing Efficient CUDA Programs

- **High arithmetic intensity**
  - Minimize time spent on memory
  - Put data in faster memory
    - Utilize shared memory, have threads cooperate for data access
  - Use coalesced global memory accesses
- **Avoid Thread Divergence**

__synchthreads()

- **Must be executed by all threads in a block**
- **If placed in an “If” statement**
  - Either all threads execute the path that includes the __synchtreads() or none
- **For an “if-else” statement**
  - If each path has synchthreads()
    - Either all threads execute the “if” part or all of them execute the “Else” part
    - If one thread hits the if part and another thread hits the else part they wait at two different barrier points!
      - FOREVER!

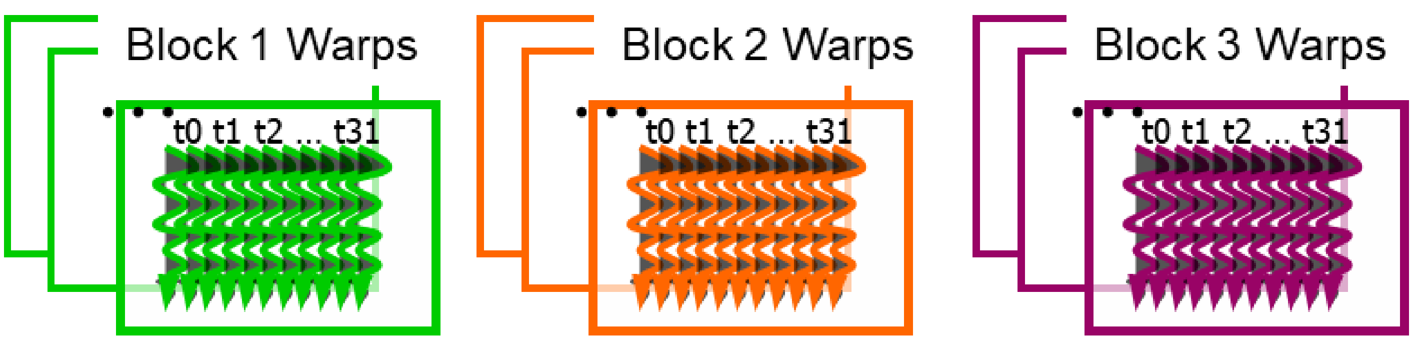
Thread Divergence

```
__global__ void odd_even(int n, int* x)
{
	int i = threadIdx.x + blockDim.x * blockIdx.x;
	if( (i & 0x01) == 0 )
	{
		x[i] = x[i] + 1;
	}
	else
	{
		x[i] = x[i] + 2;
	}
}
```

- Half the threads (even i) in the warp execute the **if** clause, the other half (odd i) the **else** clause
- Performance decreases with degree of divergence in warps

Warp Divergence

- Threads are executed in warps of 32, with all threads in the warp executing the same instruction at the same time
- What happens if different threads in a warp need to do different things?
  - CUDA will generate correct code to handle this, but to understand the performance you need to understand what CUDA does with it



Warp Divergence

- GPUs have predicated instructions which are **carried out only if a logical flag is true.**

    p: a = b + c; // computed only if p is true

- For the previous example, all threads compute the logical predicate and two predicated instructions

```
        p = (i & 0x01)
p:      x[i] = x[i] + 1; // single instruction
!p:     x[i] = x[i] + 2;
```

all threads execute both conditional branches, so **execution cost is sum of both branches** => potentially large loss of performance

Examples

- **Example kernel statement with divergence:**
  - if (threadIdx.x > 2) { }
  - two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- **Example without divergence:**
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Example: Vector addition

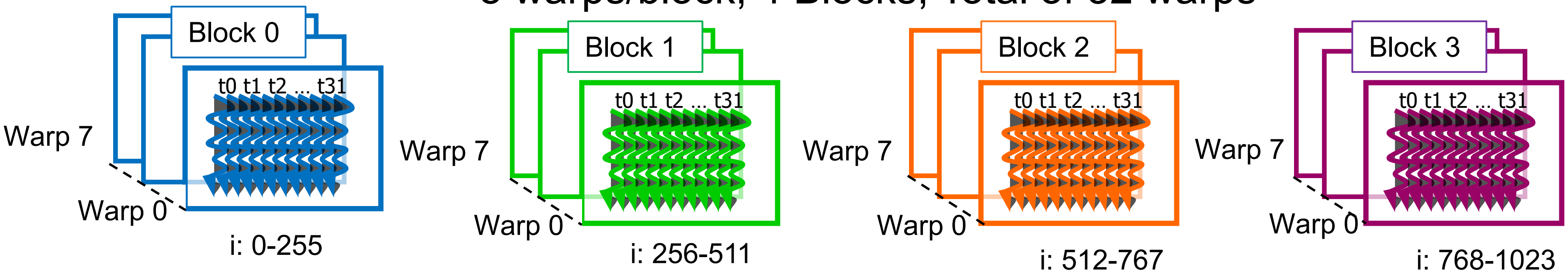
```
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if(i<n)
   C[i] = A[i] + B[i];
}
```

Assume n is 1000, block size is 256 threads

What is the ratio of warps observing control divergence with respect to the total number of warps?

Example: Vector addition (n=1000, 256 threads/block)

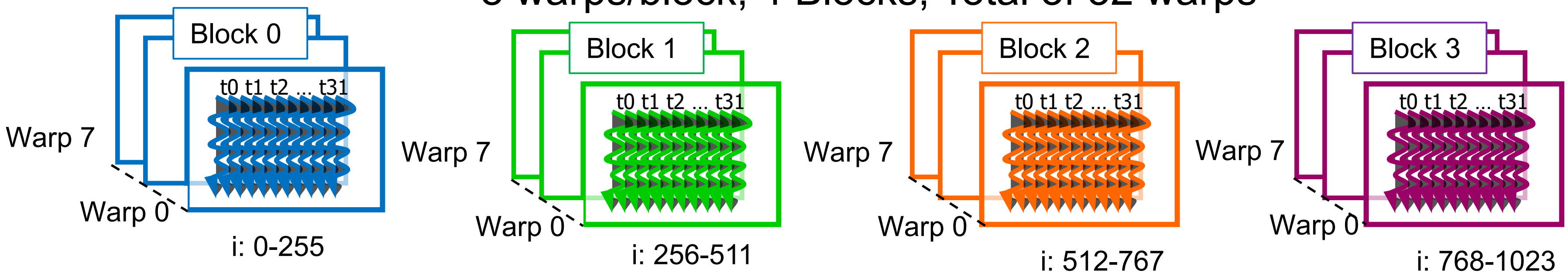
8 warps/block, 4 Blocks, Total of 32 warps




```
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n)
   C[i] = A[i] + B[i];
```

Example: Vector addition (n=1000, 256 threads/block)

8 warps/block, 4 Blocks, Total of 32 warps




```
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n)
   C[i] = A[i] + B[i];
```

Blocks 0-2 ➔ All threads in each warp take “If”

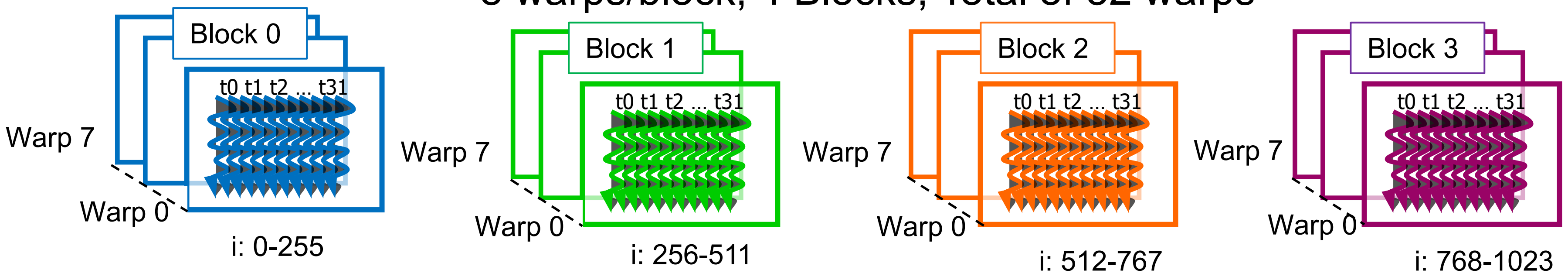
Block 3 ➔ Warps 0-6: all threads take “If”

Warp 7: 8 threads (992-999) take “if”

24 threads (1000-1023) don’t take ”if”

Example: Vector addition (n=1000, 256 threads/block)

8 warps/block, 4 Blocks, Total of 32 warps




```
int i = threadIdx.x + blockDim.x * blockIdx.x;
if(i<n)
   C[i] = A[i] + B[i];
```

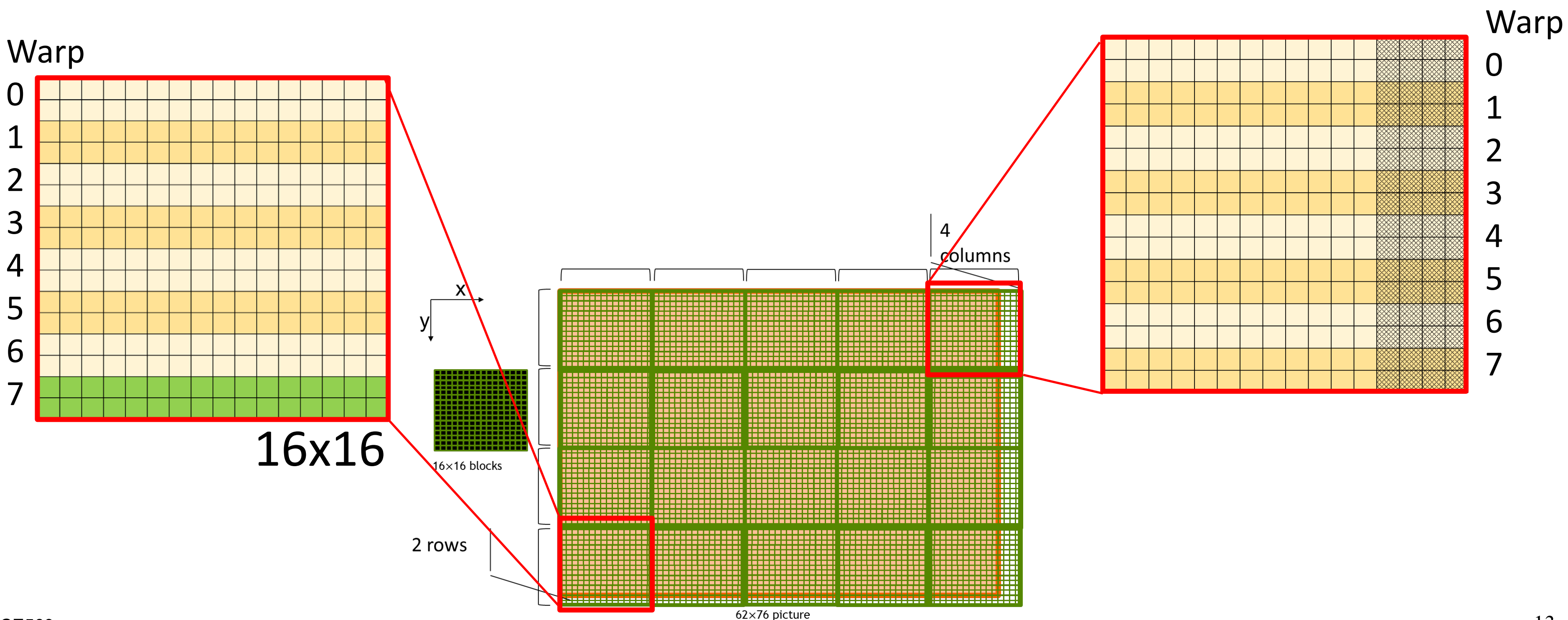
Blocks 0-2 ➔	All threads in each warp take "If"
Block 3 ➔	Warps 0-6: all threads take "If"
	Warp 7: 8 threads (992-999) take "if"
	24 threads (1000-1023) don't take "if"

1 out of 32 warps has control divergence

Less than 3% performance hit

Warp Divergence in PictureKernel

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                    int height, int width){
int Row = blockIdx.y*blockDim.y + threadIdx.y;
int Col = blockIdx.x*blockDim.x + threadIdx.x;
if ((Row < height) && (Col < width))
   d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];}
```



Warp Divergence

- In worst case, effectively lose factor 32× in performance if one thread needs expensive branch, while rest do nothing
- Typical example: PDE application with boundary conditions
  - if boundary conditions are **cheap**, loop over all nodes and branch as needed for boundary conditions
  - if boundary conditions are **expensive**, use two kernels: first for interior points, second for boundary points

Warp Divergence

- Another example: processing a long list of elements where, depending on run-time values, a few require very expensive processing
- GPU implementation approach?

Warp Divergence

- Another example: processing a long list of elements where, depending on run-time values, a few require very expensive processing
- GPU implementation approach
  - first process list to build two sub-lists of “simple” and “expensive” elements
  - then process two sub-lists separately

Divergence in a For loop

```
__global__ void use_shared_memory_GPU(float *array)
{
    int i, index = threadIdx.x;
    float average, sum = 0.0f;

   __shared__ float sh_arr[32];
   sh_arr[index] = array[index];
   __syncthreads();
    //find average of all previous elements
    for (i=0; i<index; i++) {
       sum += sh_arr[i];
    }
    average = sum / (index + 1.0f);
    array[index] = average;
    // since array[] is in global memory, this change will be seen
    // by the host (and potentially other thread blocks, if any)

}
int main(int argc, char **argv)
{
    /*  First, call a kernel that shows using shared memory */
    use_shared_memory_GPU<<<1, 32>>>(d_arr);
}
```

Reading

- CUDA Programming Guide
  - Section 5.4.2: control flow and predicates
  - Section 5.4.3: synchronization
  - Appendix B.5: __threadfence() and variants
  - Appendix B.6: __syncthreads() and variants
  - Appendix B.13: warp voting

Next

- **Putting it all together**
  - Matrix multiplication