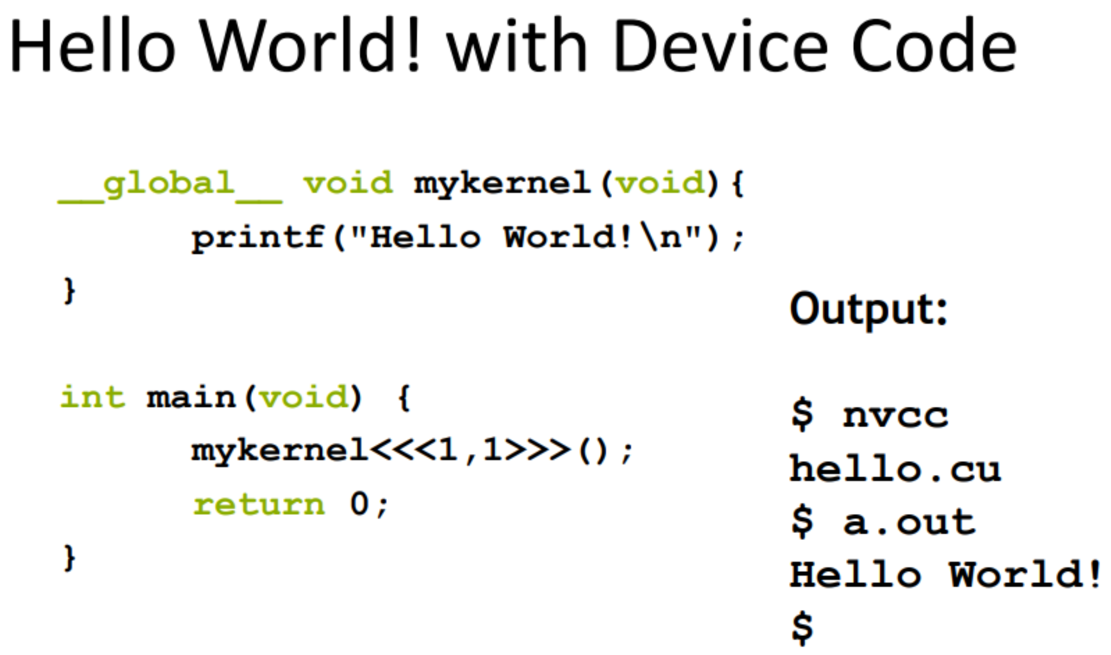
Cuda Programming: Hello world:

- Cuda programming:
  - Kernels are C functions that can only be accessed from the GPU and must have void returns.
- Cuda built-in device variables:
  - gridDim, blockDim, blockIdx, thredIdx.



Hello World! with Device Code

```
__global__ void mykernel(void){
        printf("Hello World!\n");
}

int main(void) {
        mykernel<<<1,1>>>();
        return 0;
}
```

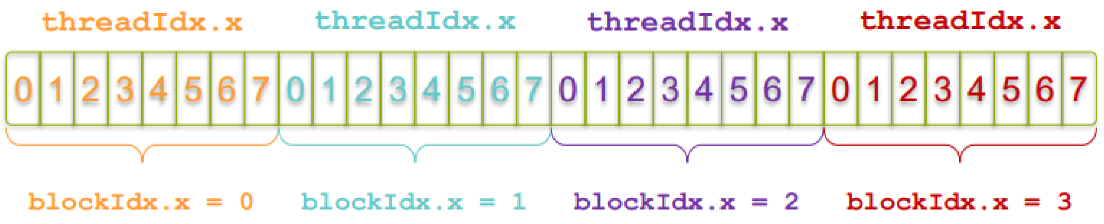
Output:

```
$ nvcc
hello.cu
$ a.out
Hello World!
$
```

Cuda programming: sum of 2 arrays:

- GPU computing is about massive parallelism
- Memory management:
  - Device pointers point to GPU memory
  - Host pointers point to CPU memory
- Cuda API for handling device memory:
  - cudaMallor(), cudaFree(), cudaMemcpy()
- Cuda threads:
  - A block can be split into parallel threads.
  - On the device, each thread can execute in parallel.

- Combining blocks and threads (Indexing):

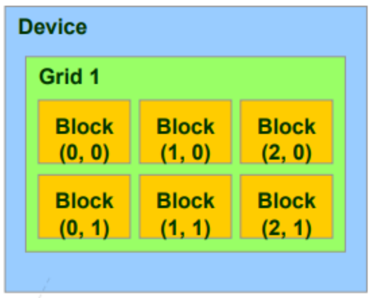


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Formatting the grid as a Matrix

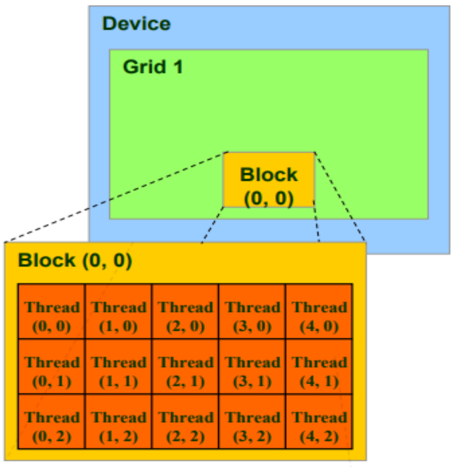
- dim3 grid (3,2);
- kernel<<<grid, 1>>>(...);



- `int index = blockIdx.x + blockIdx.y * gridDim.x;`

Formatting the grid as a Matrix

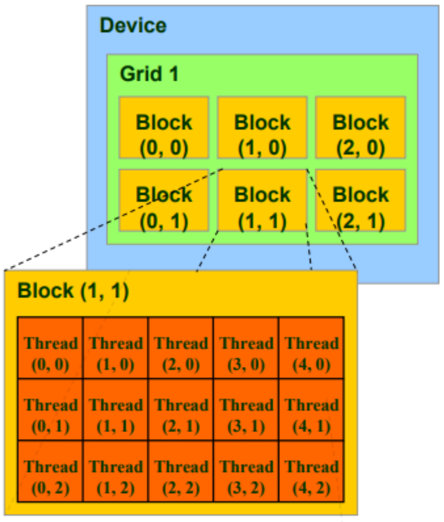
- dim3 threads(5,3);
- kernel<<1, threads>>(…);



```
•   int index = threadIdx.x + threadIdx.y * blockDim.x;
```

Formatting the grid as a Matrix

- dim3 grid(3,2);
- dim3 block(5,3);
- kernel<<<grid, block>>>(…);



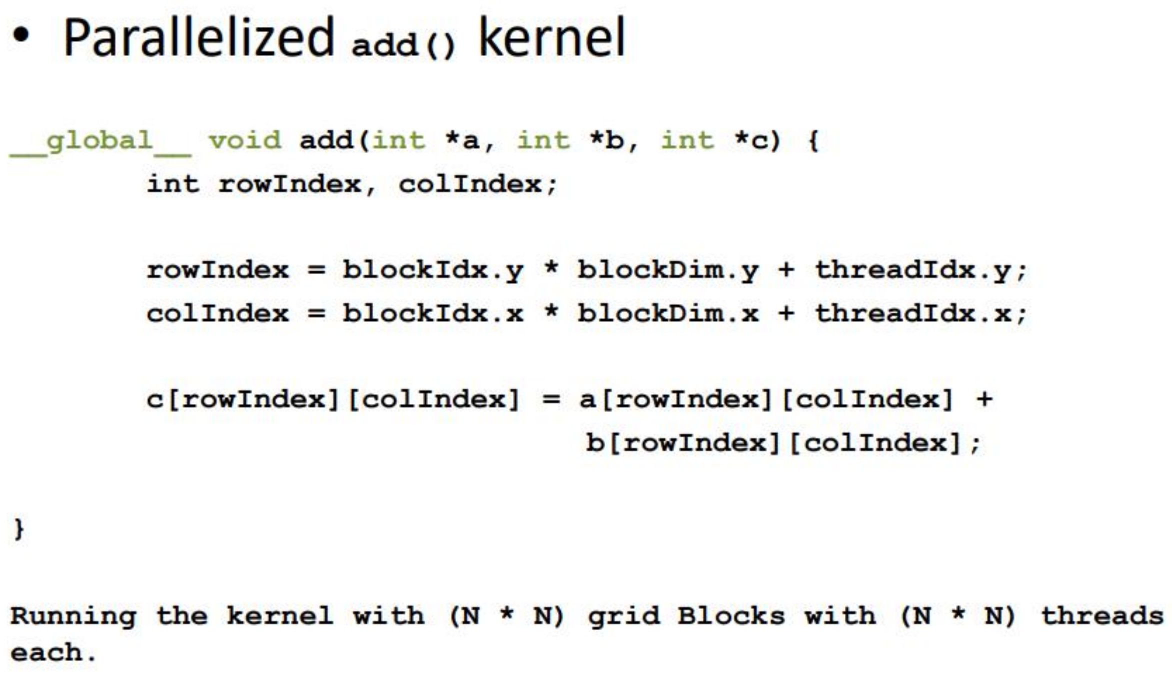
```
•   int index  = ( blockIdx.x + blockIdx.y * gridDim.x )
               * (blockDim.x * blockDim.y)
               + threadIdx.y * blockDim.x + threadIdx.x;
```

Cuda programming: Addition of 2 matrices:

- When adding matrices together, the rowIndex is usually threadIdx.y while the column index is usually threadIdx.x.
- The y is for rows, the x is for columns.
- So the addition code could look like this:

```
rowIndex – threadIdx.y;
colIndex = threadIdx.x

c[rowIndex][colIndex] = a[rowIndex][colIndex] + b[rowIndex][colIndex];
```

- After indexing blocks and threads to make it parallel, the code looks like this:



- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        int rowIndex, colIndex;

        rowIndex = blockIdx.y * blockDim.y + threadIdx.y;
        colIndex = blockIdx.x * blockDim.x + threadIdx.x;

        c[rowIndex][colIndex] = a[rowIndex][colIndex] +
                                b[rowIndex][colIndex];

}
```

```
Running the kernel with (N * N) grid Blocks with (N * N) threads
each.
```

Matrix Addition on the Device: `add()`

- Parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c, int width) {
        int rowIndex, colIndex, width;

        rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
        colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

        for (int i=;0 i < width; i++)
                for (int j=0; j < width; j++)
                        c[rowIndex + i][colIndex + j] =
                                a[rowIndex + i][colIndex  + j] +
                                b[rowIndex + i][colIndex  + j];

}
```

- the same as the previous code but with width.

Cuda programming: product of matrices:

Matrix multiplication

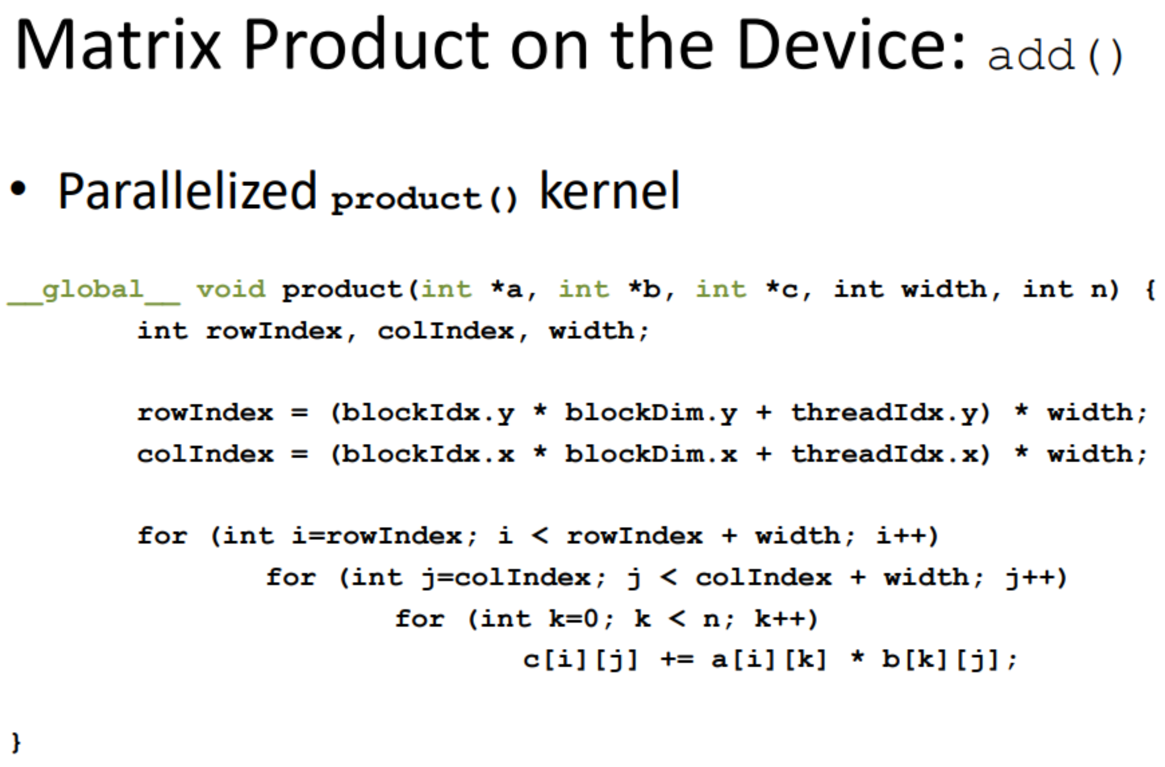
$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 0 & 7 \end{bmatrix} = \begin{bmatrix} 1*5 + 2*0 & 1*6 + 2*7 \\ 3*5 + 4*0 & 3*6 + 4*7 \end{bmatrix} = \begin{bmatrix} 5 & 20 \\ 15 & 46 \end{bmatrix}$$

$$C[\![i]\!][j] = \sum_{k=0}^{n} A[i][k] * B[k][j]$$

Matrix Product on the Device: `add()`

- Parallelized `product()` kernel

```
__global__ void product(int *a, int *b, int *c, int n) {
        int rowIndex, colIndex, width;

        rowIndex = blockIdx.y * blockDim.y + threadIdx.y;
        colIndex = blockIdx.x * blockDim.x + threadIdx.x;

        for (int k=0; k < n; k++)
           c[rowIndex][colIndex] += a[rowIndex][k] * b[k][colIndex];
}
```



Matrix Product on the Device: `add()`

- Parallelized `product()` kernel

```
__global__ void product(int *a, int *b, int *c, int width, int n) {
        int rowIndex, colIndex, width;

        rowIndex = (blockIdx.y * blockDim.y + threadIdx.y) * width;
        colIndex = (blockIdx.x * blockDim.x + threadIdx.x) * width;

        for (int i=rowIndex; i < rowIndex + width; i++)
                for (int j=colIndex; j < colIndex + width; j++)
                        for (int k=0; k < n; k++)
                                c[i][j] += a[i][k] * b[k][j];

}
```

- Same as the previous code but with width

Cuda programming: Querying the device:

cudaGetDeviceCount(int count) → returns the number of devices in the system.

char name[256] → an ASCII string identifying the device

size_t totalGlobalMem → the amount of global memory on the device in bytes

size_t sharedMemPerBlock → the maximum amount of shared memory a single block may use in bytes

int regsPerBlock → the number of 32-but registers available per block

int warpSize → the number of threads in a warp

size_t totalConstMen → the amount of available constant memory

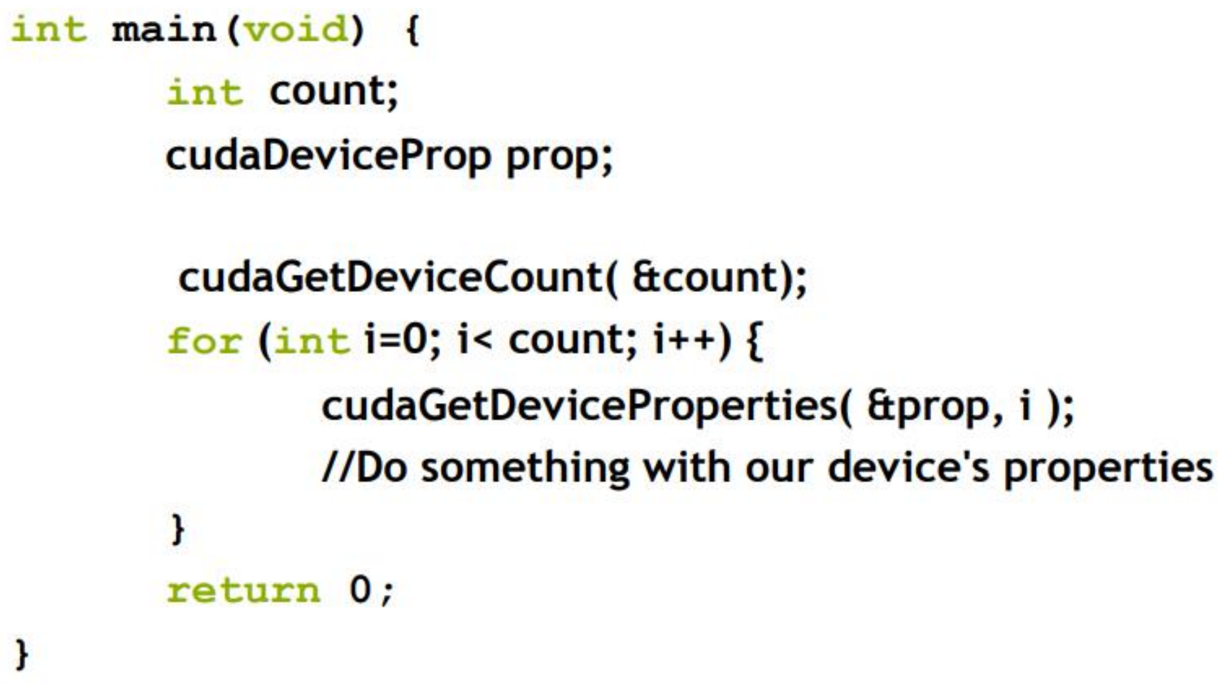
int maxThreadsPerBlock → the maximum number of threads that a block may contain

int maxThreadsDim[3] → the maximum number of threads allowed along each dimension of a block

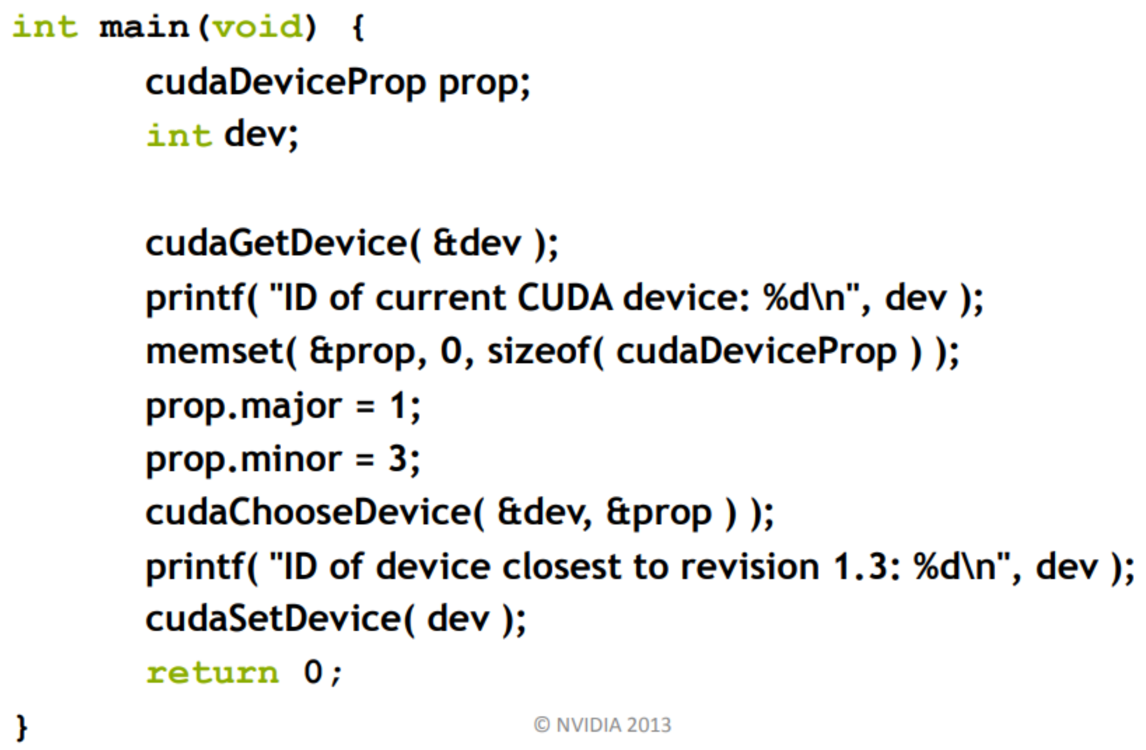
int maxGridSize[3] → the number of blocks allowed along each dimension of a grind

int multiProcessorCount → the number of multiprocessors on the device

int concurrentKernels → a Boolean value representing whether the device supports executing multiple kernels within the same context simultaneously



```
int main(void) {
        int count;
        cudaDeviceProp prop;

         cudaGetDeviceCount( &count);
        for (int i=0; i< count; i++) {
                cudaGetDeviceProperties( &prop, i );
                //Do something with our device's properties
        }
        return 0;
}
```



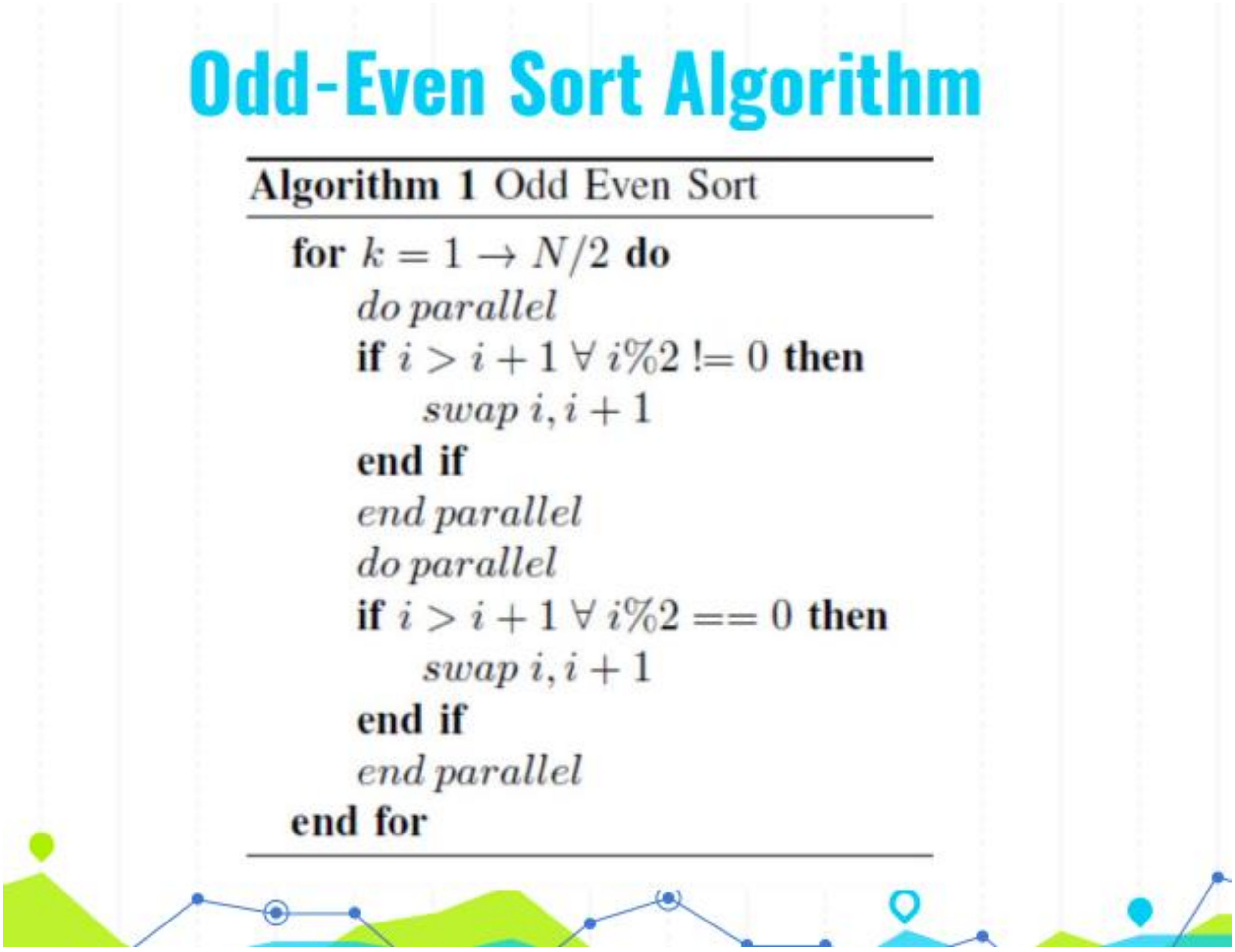
```
int main(void)  {
        cudaDeviceProp prop;
        int dev;

        cudaGetDevice( &dev );
        printf( "ID of current CUDA device: %d\n", dev );
        memset( &prop, 0, sizeof( cudaDeviceProp ) );
        prop.major = 1;
        prop.minor = 3;
        cudaChooseDevice( &dev, &prop ) );
        printf( "ID of device closest to revision 1.3: %d\n", dev );
        cudaSetDevice( dev );
        return  0;
}
```

© NVIDIA 2013

Parallel sorting algorithms:

Odd-even sort:

- The odd-even sort is based on the bubble-sort technique
- Adjacent pairs of items in an array are exchanged if they are found to be out of order
- Total running time is O(log 2N)
- Code for this technique:



Odd-Even Sort Algorithm

Algorithm 1 Odd Even Sort

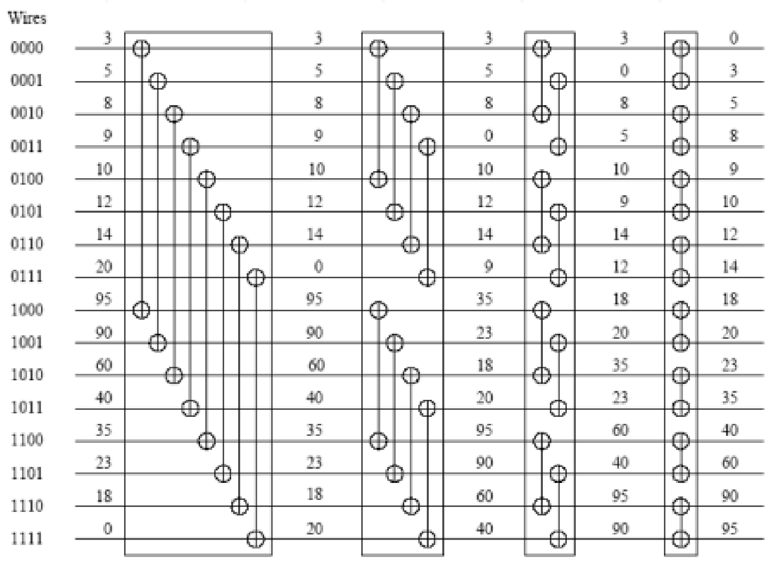
```
for $k = 1 \rightarrow N/2$ do
    do parallel
    if $i > i + 1 \ \forall \ i\%2 \ != 0$ then
        swap $i, i + 1$
    end if
    end parallel
    do parallel
    if $i > i + 1 \ \forall \ i\%2 == 0$ then
        swap $i, i + 1$
    end if
    end parallel
end for
```

Bitonic sort:

- A bitonic sorting network sorts n elements in $O(\log^2 n)$ time
- Bitonic sequence has two tones → increasing and decreasing.

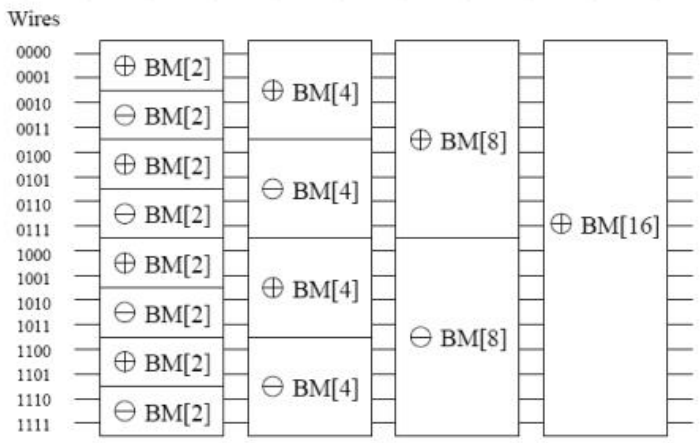
- The kernel of the network is the rearrangement of a bitonic sequence into a sorted

Sorting Networks: Bitonic Sort

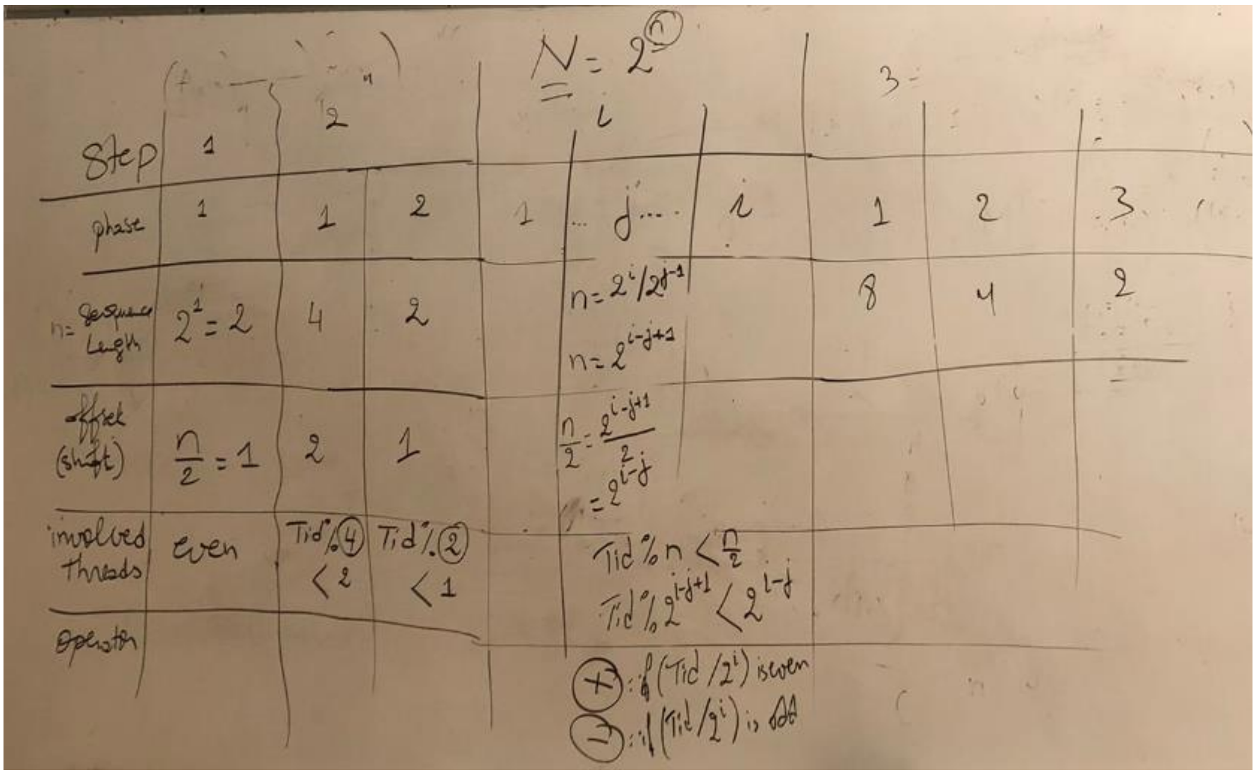


A bitonic merging network for $n = 16$. The entire figure represents a $\oplus$BM[16] bitonic merging network. The network takes a bitonic sequence and outputs it in sorted order.

sequence



A schematic representation of a network that converts an input sequence into a bitonic sequence. In this example, ⊕BM[k] and ⊖BM[k] denote bitonic merging networks of input size k that use ⊕ and ⊖ comparators, respectively. The last merging network (⊕BM[16]) sorts the input. In this example, $n = 16$.



$N = 2^{n}$

Step	1	2		i			3		
phase	1	1	2	1	... j ...	i	1	2	3
n= Sequence Length	$2^1 = 2$	4	2		$n = 2^i/2^{j-1}$ $n = 2^{i-j+1}$		8	4	2
offset (shift)	$\frac{n}{2} = 1$	2	1		$\frac{n}{2} = \frac{2^{i-j+1}}{2} = 2^{i-j}$				
involved threads	even	Tid % 4 < 2	Tid % 2 < 1		Tid % n < $\frac{n}{2}$ Tid % $2^{i-j+1} < 2^{i-j}$				
Operation					⊕ : if (Tid / 2^i) is even ⊖ : if (Tid / 2^i) is odd				

Dynamic parallelism:

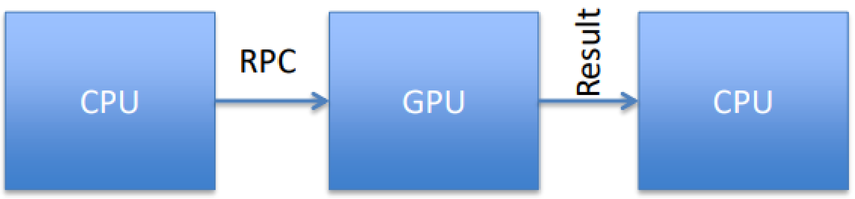
C → Compute

U → Unified

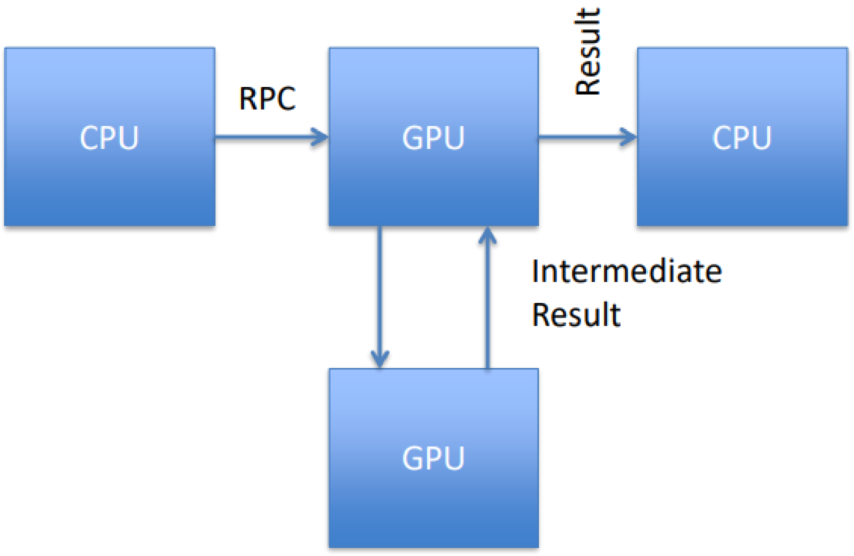
D → Device
A → Architecture

- Compute unified architecture:
  - Hybrid CPU/GPU code
  - Low latency code is running on the CPU while the high latency code is running on the GPU
- Types of parallelism:
  - Task parallelism:
    - Problem is divided to tasks, which are processed independently
  - Data parallelism:
    - Same operation is performed over many data items
  - Pipeline parallelism:
    - Data are flowing through a sequence of stages, which operate concurrently
- Parallelism in GPU:
  - Data parallelism:
    - The same kernel is executed by many threads
    - Thread process one data item
  - Limited task parallelism:
    - Multiple kernels executed simultaneously
    - At most as many kernels as SMP
  - We don’t have:
    - Any means of kernel-wide synchronization
    - Any guarantees that two blocks/kernels will actually run concurrently
- What is dynamic parallelism:
  - The ability to launch new kernels from the GPU:
    - Dynamically
      - Based on run time
    - Simultaneously:
      - From multiple threads at once
    - Independently:
      - Each thread can launch a different grid

Execution Model (Overview)



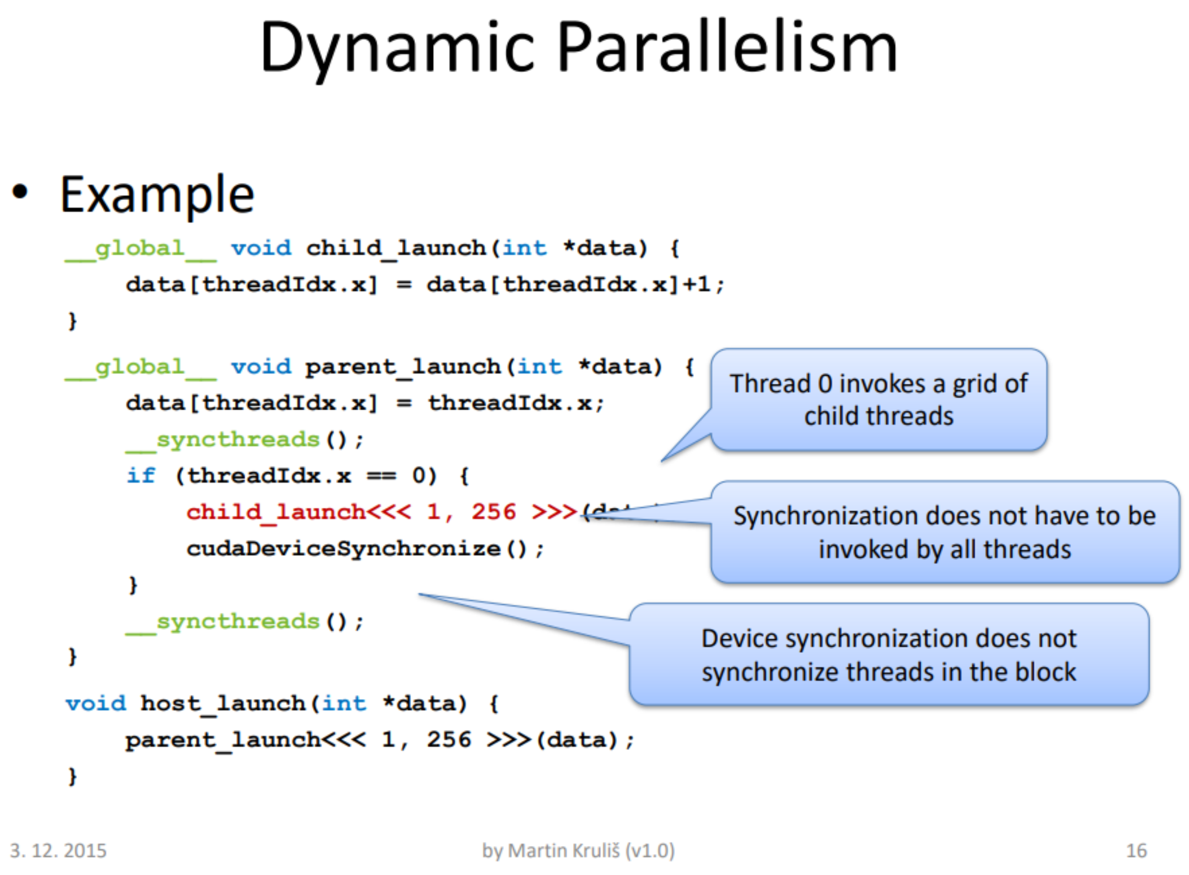
Fermi: Only CPU can generate GPU work



Kepler: GPU can generate work for itself

- Dynamic parallelism:
  - Allows program flow to be controlled by the GPU
  - Allows recursion and subdivision of problems
  - Interesting data is not uniformly distributed
  - Dynamic parallelism can launch additional threads in interesting areas
  - Allows higher resolution in critical areas without slowing down others
- Problem cases:
  - Unsuitable problems for GPUs:
    - Processing irregular data structures
    - Regular structures with irregular processing workload
    - Iterative tasks with explicit synchronization
    - Pipeline-oriented tasks with many simple stages
  - Solutions:
    - Iterative kernel execution
    - Mapping irregular structures to regular grids
    - 2-phase algorithms
- Dynamic parallelism purpose:
  - The device does not need to synchronize with host to issue new work to the device
  - Irregular parallelism may be expressed more easily

- Without dynamic parallelism:
  - Data travels back and forth between the CPU and GPU many times
  - This is because of the inability of the GPU to create more work on itself depending on the data
- With dynamic parallelism:
  - GPU can generate work on itself based on intermediate results, without the involvement of CPU
  - Permits dynamic run time decisions
  - Leaves the CPU free to do other work
- How does dynamic parallelism work?
  - Portions of CUDA runtime are ported to device side:
    - Kernel execution
    - Device synchronization
    - Streams, events, and async memory operations
  - Kernel launches are asynchronous:
    - No guarantee the child kernel starts immediately
    - Synchronization points may cause context switch
  - Block-wise locality of resources:
    - Streams and events are shared in thread block



Dynamic Parallelism

- Example

```
__global__ void child_launch(int *data) {
    data[threadIdx.x] = data[threadIdx.x]+1;
}
__global__ void parent_launch(int *data) {
    data[threadIdx.x] = threadIdx.x;
    __syncthreads();
    if (threadIdx.x == 0) {
        child_launch<<< 1, 256 >>>(data);
        cudaDeviceSynchronize();
    }
    __syncthreads();
}
void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

Thread 0 invokes a grid of child threads

Synchronization does not have to be invoked by all threads

Device synchronization does not synchronize threads in the block

3. 12. 2015 by Martin Kruliš (v1.0) 16

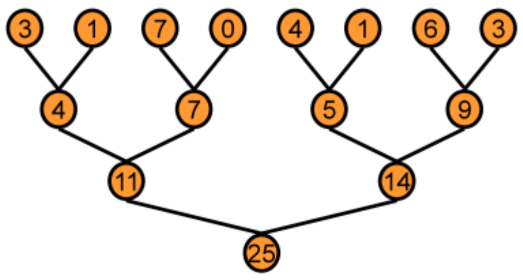
- More on dynamic parallelism:
  - Scheduling can be controlled by streams
  - Launched kernels may execute out-of-order within a stream
  - Named streams can guarantee concurrency
  - Nested dependencies:

        - cudaDeviceSynchronize()
        - can be used inside a kernel
        - synchronizes all launches by any kernel in block
        - does not imply __syncthreads()
    - kernel launch implied memory sync operation
    - child sees state at time of launch
    - parent sees child writes after sync
    - local and shared memory are private, cannot be shared with children

Optimizing parallel reduction in CUDA:

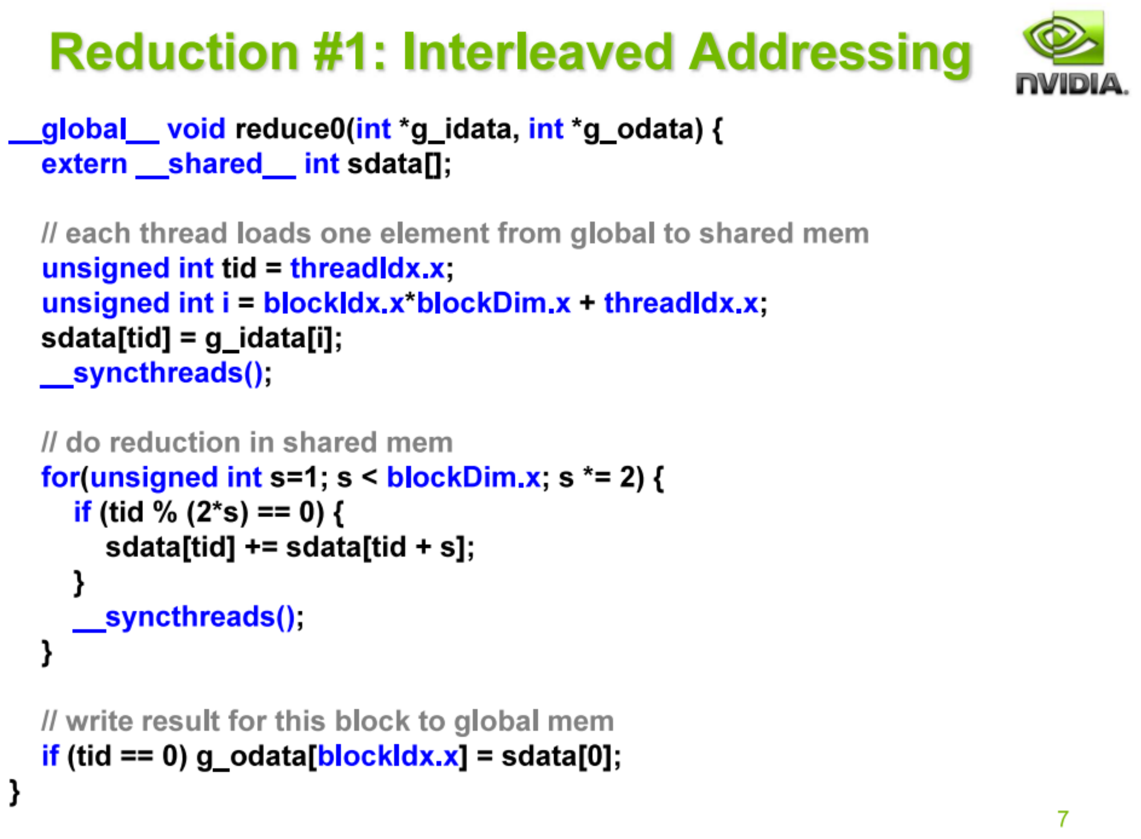
- Parallel reduction:
    - Common and important data parallel primitive
    - Easy to implement in CUDA
    - Serves as a great optimization example
    - It has a tree-based approach used within each thread block

Tree-based approach used within each thread block



- Problem: global synchronization:
    - CUDA has no global synchronization because it’s expensive to build in hardware for GPUs with high processor count
    - The solution is to decompose into multiple kernels:
        - Kernel launches serves as a global synchronization point
        - Kernel launch has negligible HW overhead, low SW overhead

- Reduction 1 → interleaved addressing with divergent branching:

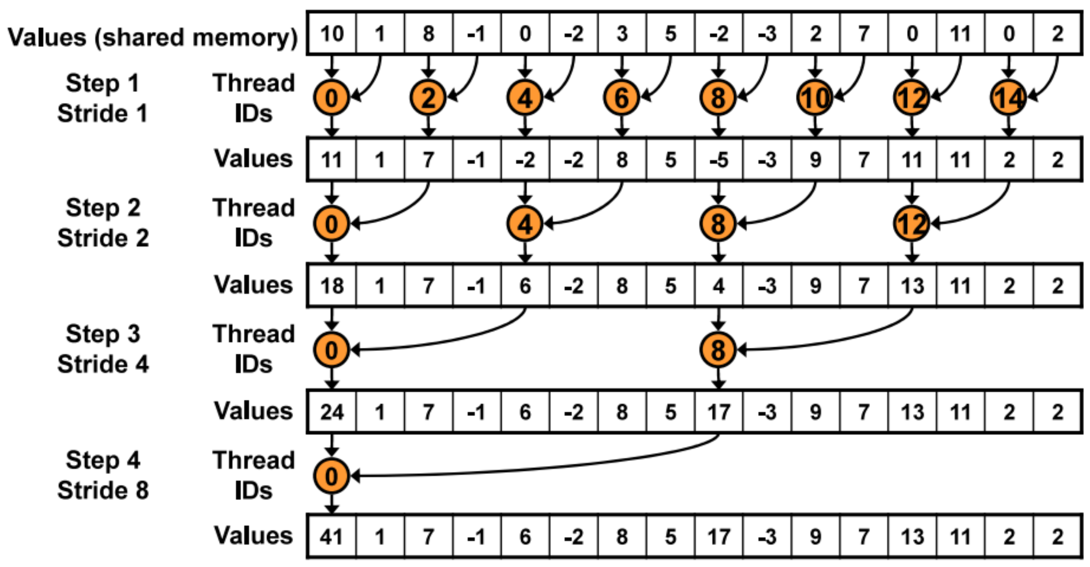


Reduction #1: Interleaved Addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
   extern __shared__ int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

   // do reduction in shared mem
   for(unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

7

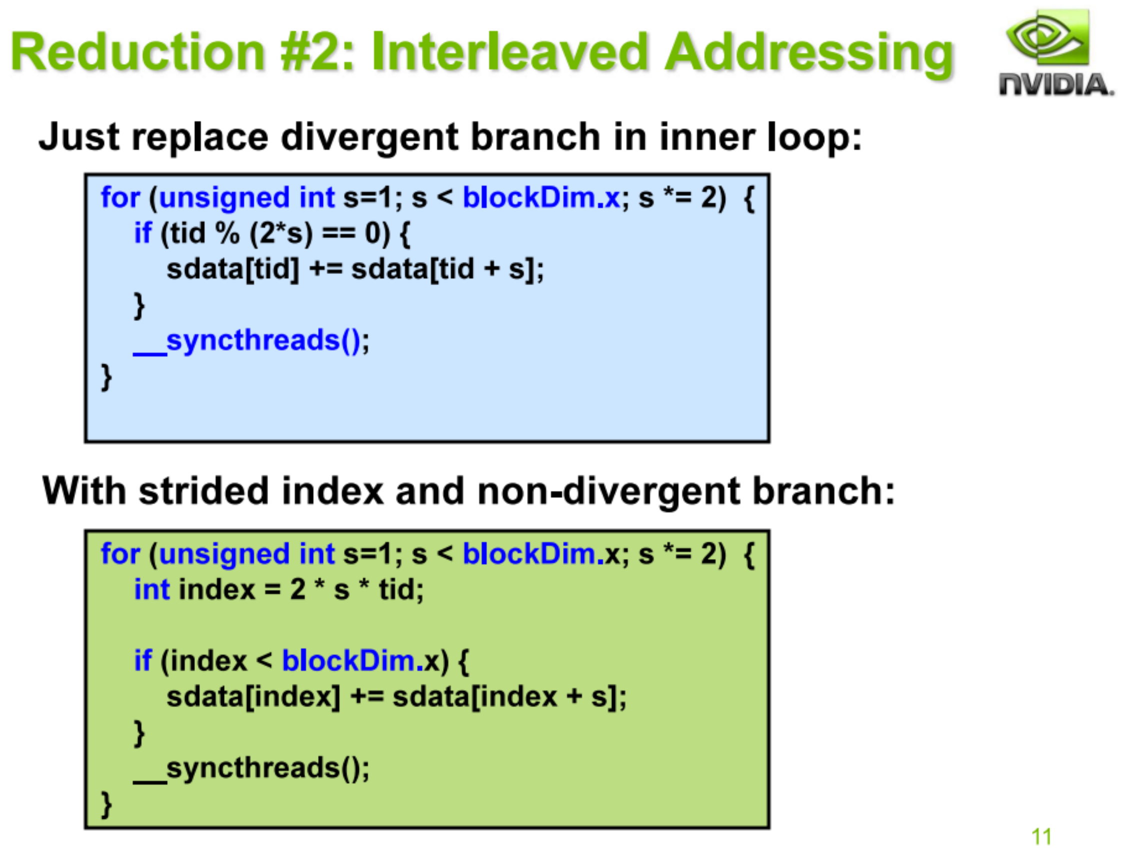
Parallel Reduction: Interleaved Addressing


Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 2 4 6 8 10 12 14
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 4 8 12
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 8
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

8

- The problem with this method is that highly divergent warps are inefficient, and the % operator is very slow

- Reduction 2 → interleaved addressing with bank conflicts:



Reduction #2: Interleaved Addressing

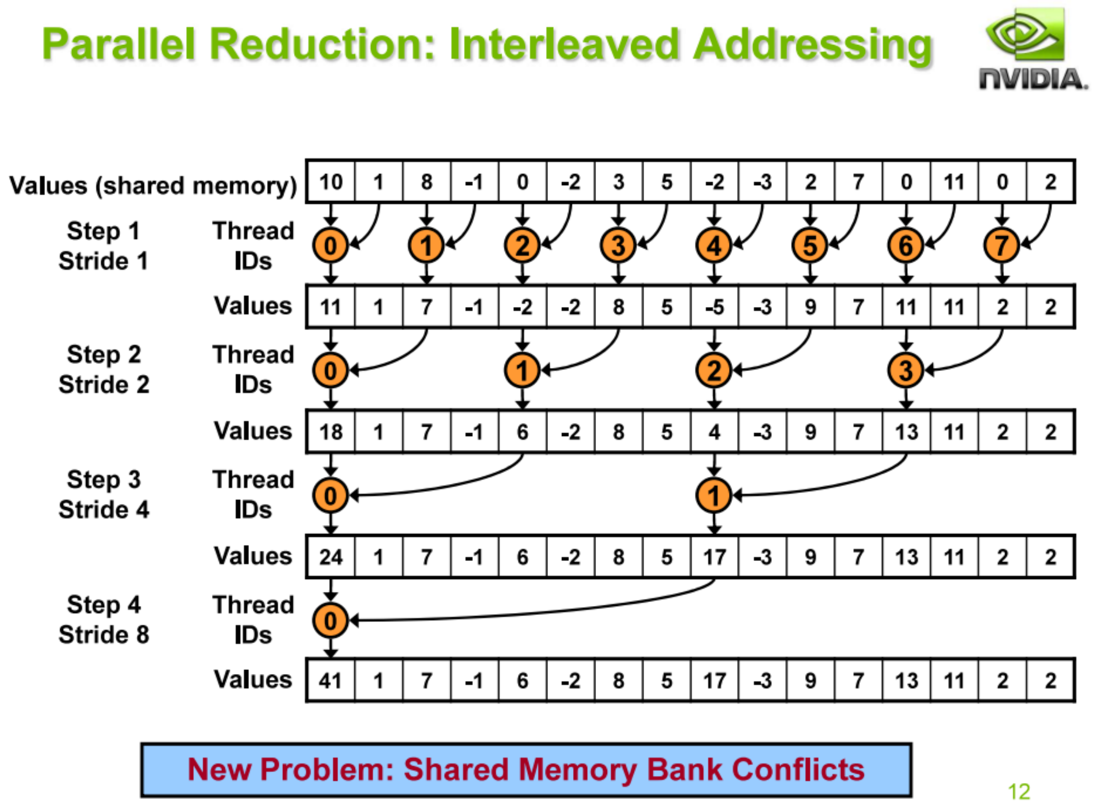
Just replace divergent branch in inner loop:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
   if (tid % (2*s) == 0) {
      sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
}
```

With strided index and non-divergent branch:

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
   int index = 2 * s * tid;

   if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
   }
   __syncthreads();
}
```

11



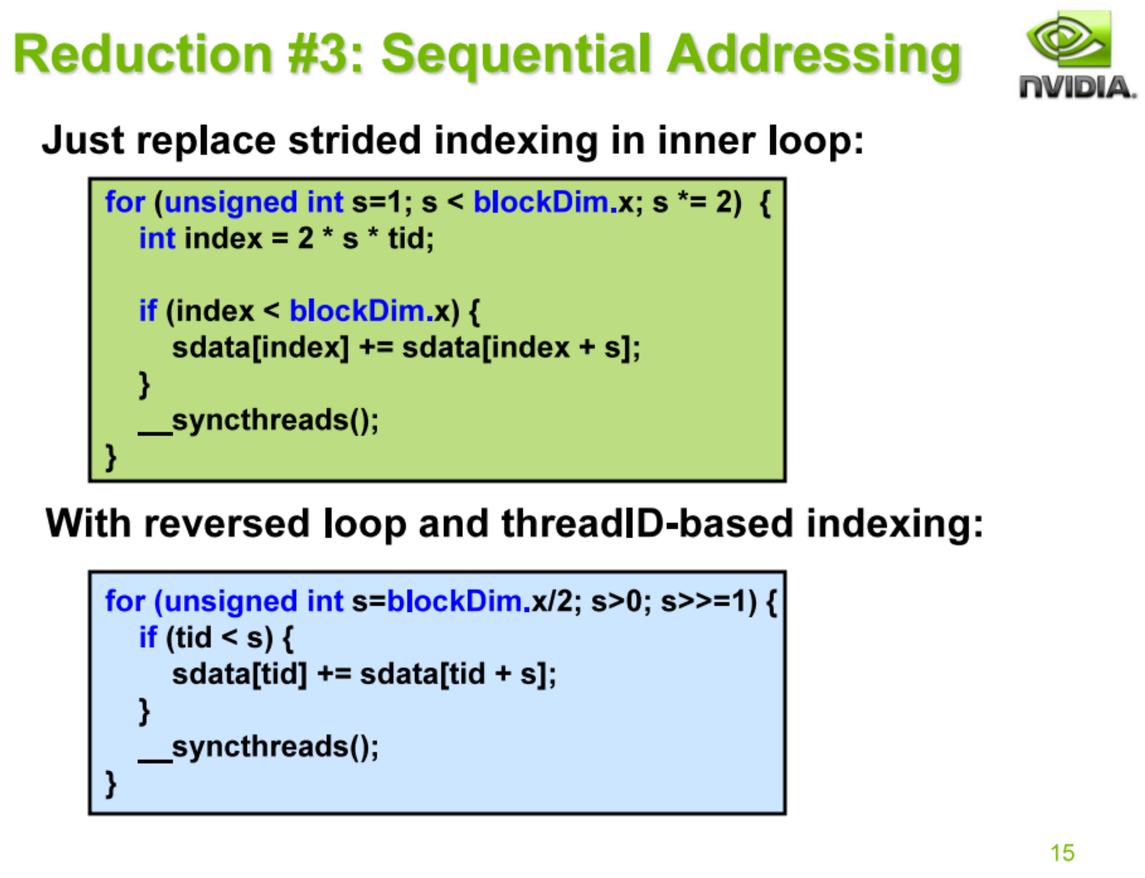
Parallel Reduction: Interleaved Addressing

Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		1		2		3		4		5		6		7	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				1				2				3			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								1							
	Values	24	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2
Step 4 Stride 8	Thread IDs	0															
	Values	41	1	7	-1	6	-2	8	5	17	-3	9	7	13	11	2	2

New Problem: Shared Memory Bank Conflicts

12

- Reduction 3 → sequential addressing:



Reduction #3: Sequential Addressing

Just replace strided indexing in inner loop:

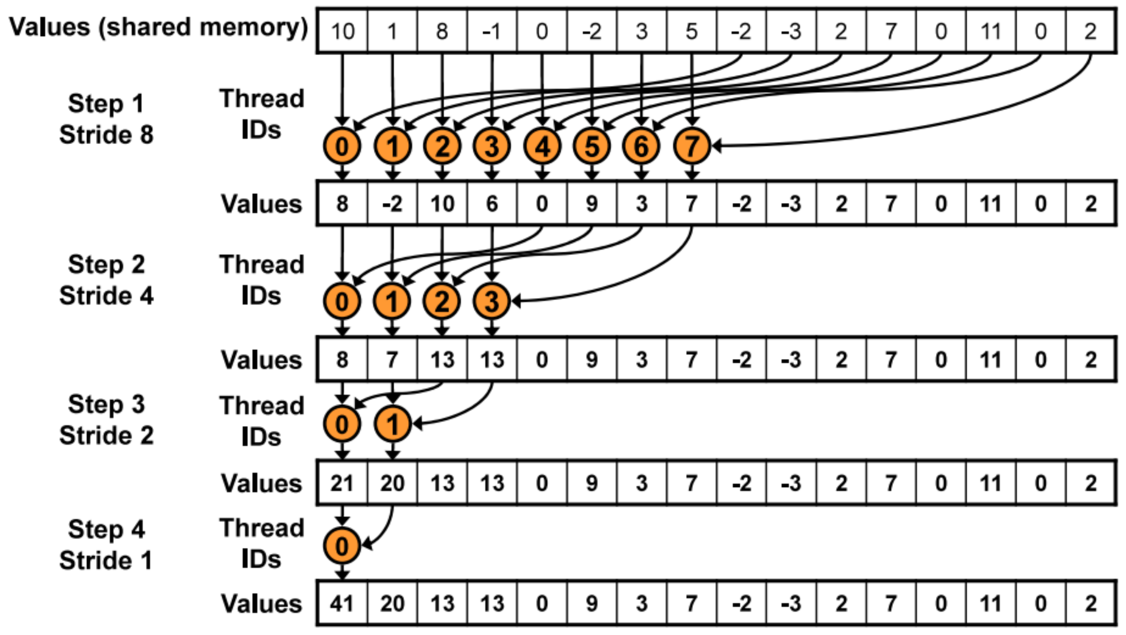
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
   int index = 2 * s * tid;

   if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
   }
   __syncthreads();
}
```

With reversed loop and threadID-based indexing:

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
   if (tid < s) {
      sdata[tid] += sdata[tid + s];
   }
   __syncthreads();
}
```

15

Parallel Reduction: Sequential Addressing

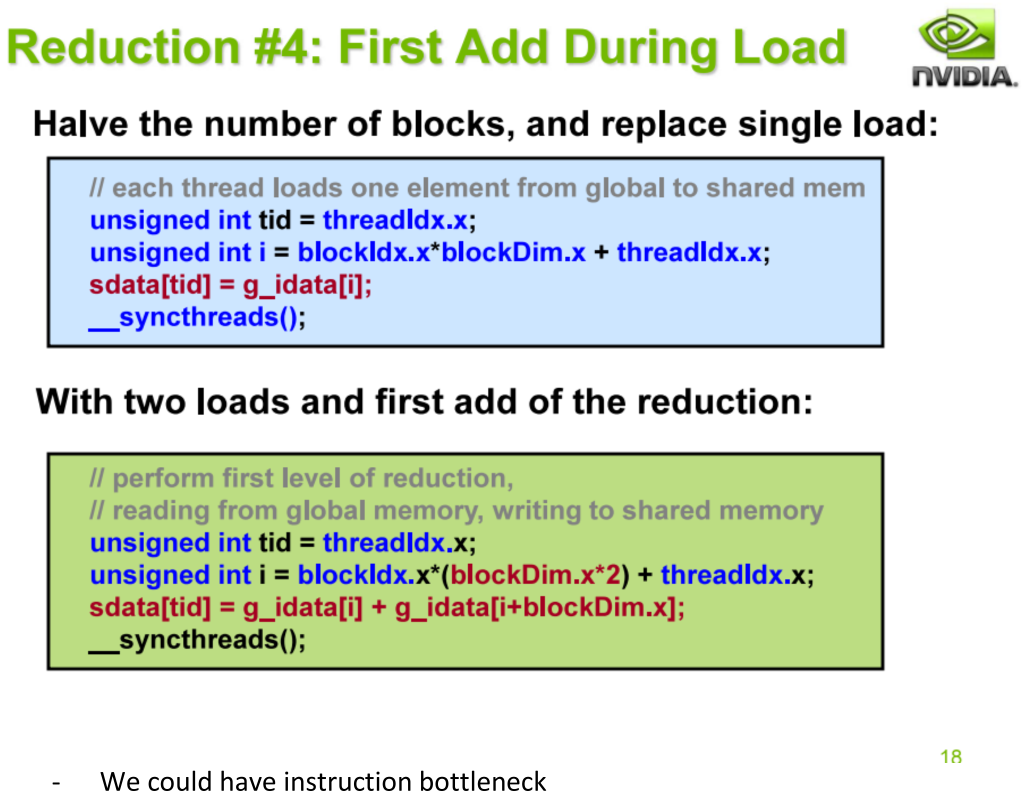


Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Sequential addressing is conflict free

14

- Problem with reduction 3:
    - Idle threads
        - Half of the threads are idle on the first loop iteration
- Reduction 4 → first add during load:



Reduction #4: First Add During Load

Halve the number of blocks, and replace single load:

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

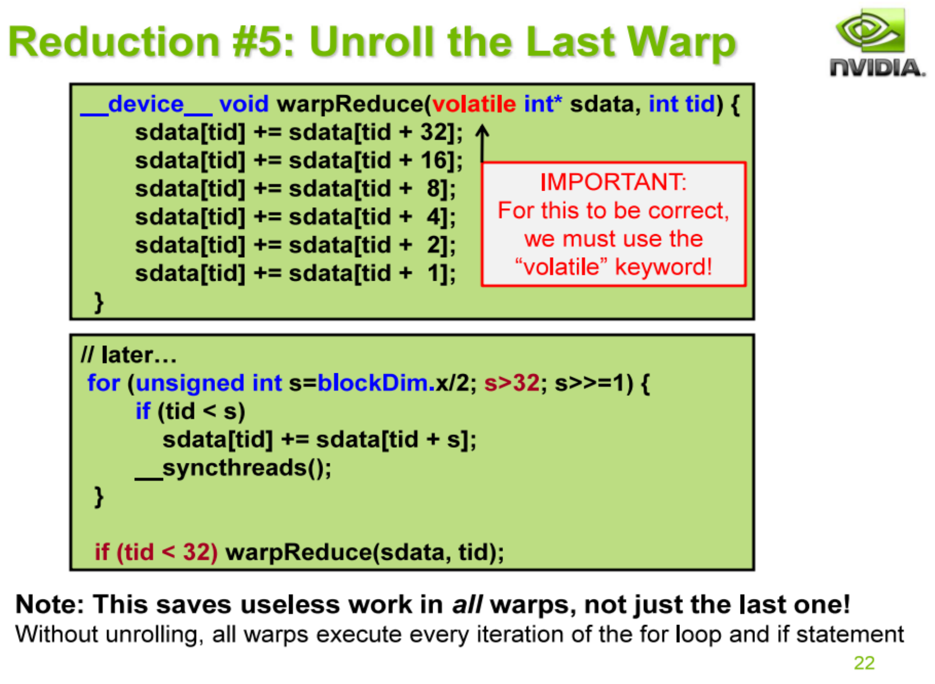
With two loads and first add of the reduction:

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

18

- We could have instruction bottleneck
- So, the next improvement (reduction 5) will be unrolled loops
- Unrolling the last warp:
    - As reduction proceeds, the number of active threads decrease
    - Instructions are SIMD synchronous within a warp

- Reduction 5 → unroll the last warp:



Reduction #5: Unroll the Last Warp

```
__device__ void warpReduce(volatile int* sdata, int tid) {
        sdata[tid] += sdata[tid + 32];
        sdata[tid] += sdata[tid + 16];
        sdata[tid] += sdata[tid +  8];
        sdata[tid] += sdata[tid +  4];
        sdata[tid] += sdata[tid +  2];
        sdata[tid] += sdata[tid +  1];
}
```

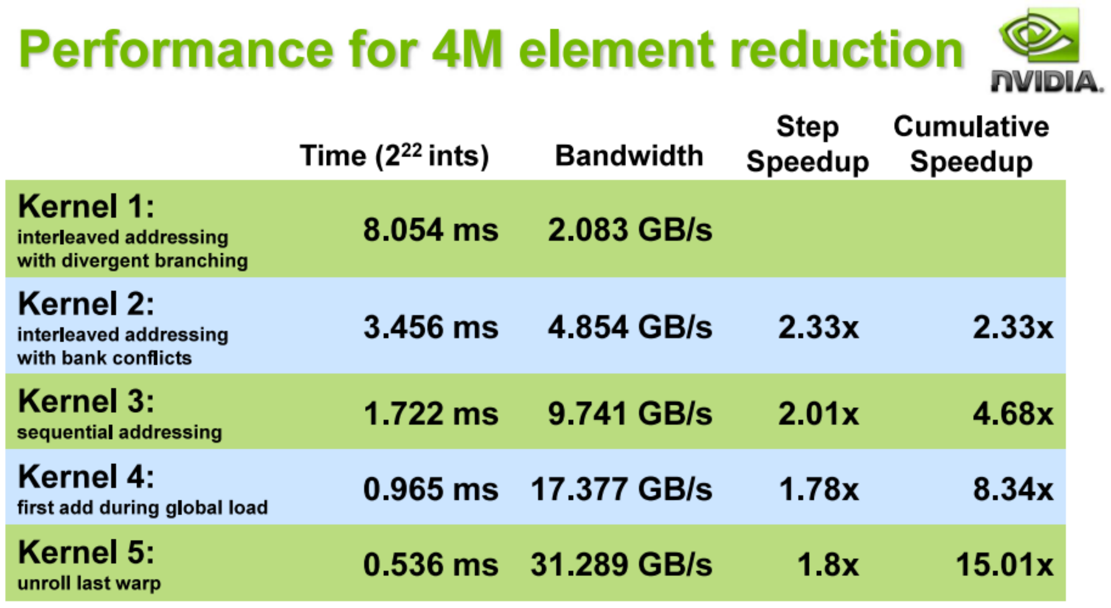
IMPORTANT:
For this to be correct, we must use the “volatile” keyword!

```
// later…
for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
}

if (tid < 32) warpReduce(sdata, tid);
```

Note: This saves useless work in *all* warps, not just the last one!
Without unrolling, all warps execute every iteration of the for loop and if statement

22

- Performance for all 5 reductions:

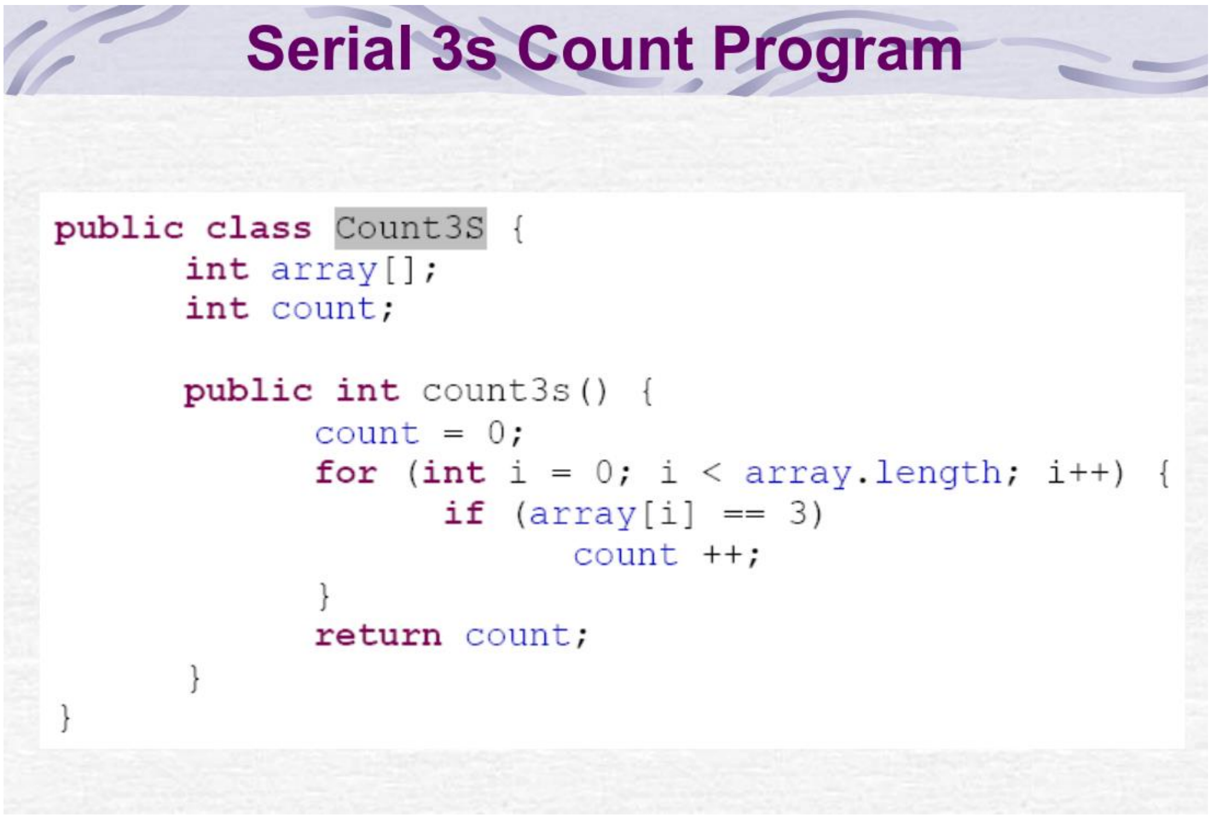


Performance for 4M element reduction

	Time (2^{22} ints)	Bandwidth	Step Speedup	Cumulative Speedup
Kernel 1: interleaved addressing with divergent branching	8.054 ms	2.083 GB/s		
Kernel 2: interleaved addressing with bank conflicts	3.456 ms	4.854 GB/s	2.33x	2.33x
Kernel 3: sequential addressing	1.722 ms	9.741 GB/s	2.01x	4.68x
Kernel 4: first add during global load	0.965 ms	17.377 GB/s	1.78x	8.34x
Kernel 5: unroll last warp	0.536 ms	31.289 GB/s	1.8x	15.01x

Reasoning about performance:

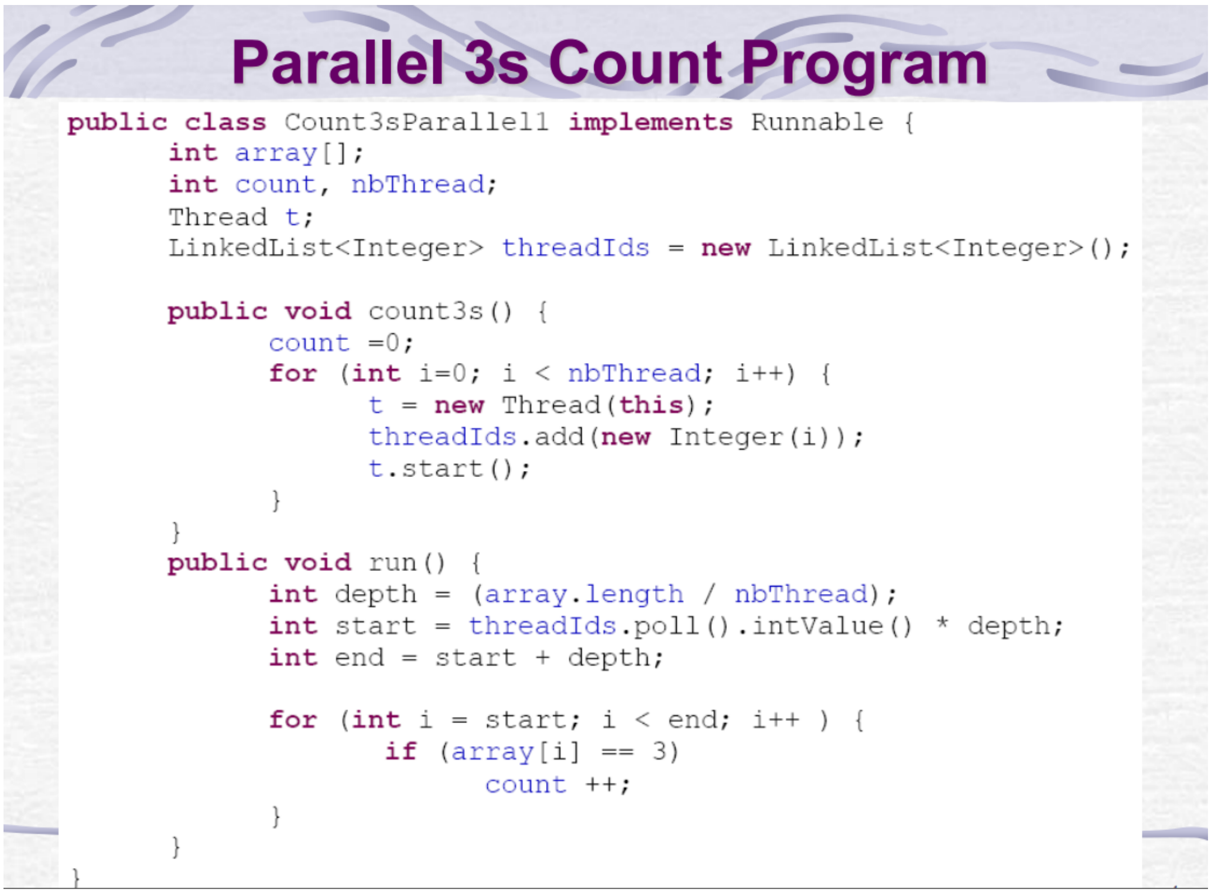
- Parallelism vs Performance:
  - Ideally a problem that takes T time on a single processor can execute in T/P time on P processors
  - The challenge of meeting T/P goal becomes more difficult as P increases
  - There are certain cases where the P processors produce shorter execution time than T/P
- First code, serial 3s count program:



Serial 3s Count Program

```
public class Count3S {
        int array[];
        int count;

        public int count3s() {
                count = 0;
                for (int i = 0; i < array.length; i++) {
                        if (array[i] == 3)
                                count ++;
                }
                return count;
        }
}
```

- Second code, parallel 3s count program (first one):



Parallel 3s Count Program

```
public class Count3sParallel1 implements Runnable {
        int array[];
        int count, nbThread;
        Thread t;
        LinkedList<Integer> threadIds = new LinkedList<Integer>();

        public void count3s() {
                count =0;
                for (int i=0; i < nbThread; i++) {
                        t = new Thread(this);
                        threadIds.add(new Integer(i));
                        t.start();
                }
        }
        public void run() {
                int depth = (array.length / nbThread);
                int start = threadIds.poll().intValue() * depth;
                int end = start + depth;

                for (int i = start; i < end; i++ ) {
                         if (array[i] == 3)
                                 count ++;
                }
        }
}
```

- The problem with this program is that the threads can all access the array and increment the count variable at the same time, which will result in inaccurate results.

- Third code, parallel 3s count program (second one):

```
public class Count3SParallel2 implements Runnable {
        int array[];
        int count, nbThread;
        Thread t;
        LinkedList<Integer> threadIds = new LinkedList<Integer>();
        Mutex m = new Mutex();

        public void count3s() {
                count =0;
                for (int i=0; i < nbThread; i++) {
                        t = new Thread(this);
                        threadIds.add(new Integer(i));
                        t.start();
                }
        }
        public void run() {
                int depth = (array.length / nbThread);
                int start = threadIds.poll().intValue() * depth;
                int end = start + depth;

                for (int i = start; i < end; i++ ) {
                           if (array[i] == 3) {
                                    try {
                                            m.acquire();
                                            count ++;
                                            m.release();
                                    } catch (InterruptedException e) {
                                            e.printStackTrace();
                                    }
                           }
                }
        }
}
```

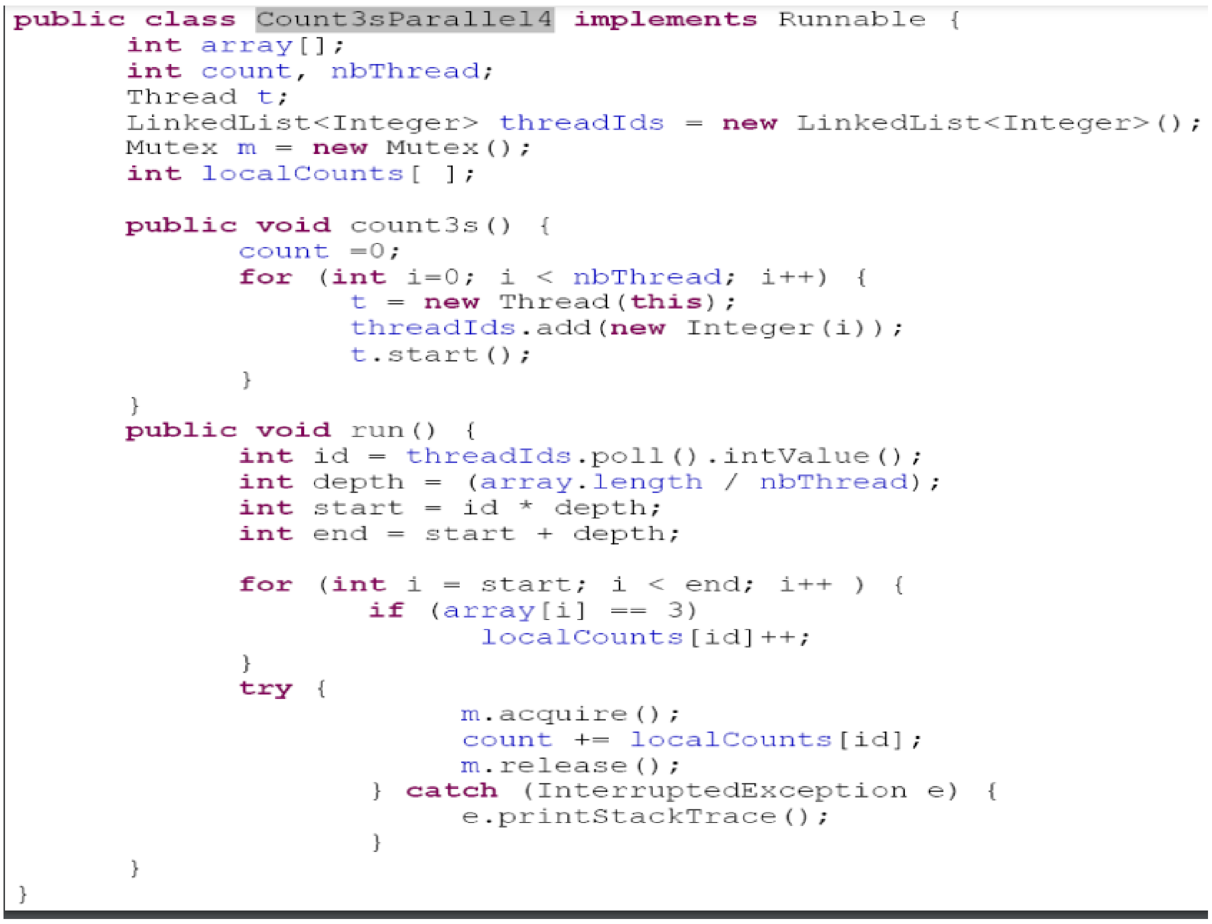
- Here, we fixed the issue of threads accessing at the same time by mutual exclusion (Mutex m) which will act as a key for threads to not access at the same time
- However, this will bring a lot of idle time on the table so further modifications are required

- Fourth code, parallel 3s count program (third one):

```
public class Count3sParallel3 implements Runnable {
        int array[];
        int count, nbThread;
        Thread t;
        LinkedList<Integer> threadIds = new LinkedList<Integer>();
        Mutex m = new Mutex();

        public void count3s() {
                count =0;
                for (int i=0; i < nbThread; i++) {
                        t = new Thread(this);
                        threadIds.add(new Integer(i));
                        t.start();
                }
        }
        public void run() {
                int depth = (array.length / nbThread);
                int start = threadIds.poll().intValue() * depth;
                int end = start + depth;
                int localCount = 0;

                for (int i = start; i < end; i++ ) {
                        if (array[i] == 3)
                                localCount++;
                }
                 try {
                                m.acquire();
                                count += localCount;
                                m.release();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
        }
}
```

- We fixed the issue of mutex by adding a local variable that will speed up the addition and threads don’t have to keep going in and out of the try block to add
- Another problem is posed here, which is copying data from local memory to global memory which is slow

- Fifth code, parallel 3s count (fourth one):



```java
public class Count3sParallel4 implements Runnable {
        int array[];
        int count, nbThread;
        Thread t;
        LinkedList<Integer> threadIds = new LinkedList<Integer>();
        Mutex m = new Mutex();
        int localCounts[ ];

        public void count3s() {
                count =0;
                for (int i=0; i < nbThread; i++) {
                        t = new Thread(this);
                        threadIds.add(new Integer(i));
                        t.start();
                }
        }
        public void run() {
                int id = threadIds.poll().intValue();
                int depth = (array.length / nbThread);
                int start = id * depth;
                int end = start + depth;

                for (int i = start; i < end; i++ ) {
                          if (array[i] == 3)
                                  localCounts[id]++;
                }
                try {
                                m.acquire();
                                count += localCounts[id];
                                m.release();
                          } catch (InterruptedException e) {
                                e.printStackTrace();
                          }
        }
}
```

  - Here, we fixed the issue of the previous run by making the local count a global array of 0 and each thread will increment the value of its slots in the array and then in the try block they only copy once.
- Threads:
  - A thread consists mainly of:
    - A program code
    - A call stack
    - And some modest amount of thread specific data
  - Threads share access to memory:
    - Communicate by reading from or writing to memory that is visible to all
    - Shared-memory parallel programming
- Processes:
  - A process is a thread that also has its own private address space
  - Processes communicate by passing messages
  - Processes have much more associated state than threads
- Latency and throughput:
  - Latency refers to the amount of time it takes to complete a given unit of work
  - Throughput refers to the amount of work that can be completed per unit of time
  - Parallelism can be exploited to improve throughput
  - Parallelist can be exploited to hide/reduce latency

- Source of performance loss:
  - Overhead
  - Non-parallelizable computation
  - Idle processors
  - Contention of resources
- Overhead:
  - Any cost that is incurred in the parallel solution but not in the serial solution
  - Sources of overhead:
    - Communication
    - Synchronization
    - Computation
    - Memory
- Non-parallelizable code:
  - Amandahl's Law:
    - If 1/S of a computation is inherently sequential
    - The maximum performance improvement is a factor of S
    - Tp = (1/S * Ts) + ((1-1/S) * TS/P)
- Idle time:
  - Idle time is a consequence of synchronization and communication
  - Ideally, all processors are working all the time
- Contention:
  - Contention is the degradation of system performance caused by computation
  - Contention can lead to slow down performance
  - Lock contention can reduce performance