CUDA Workshop

Nuno Subtil
nsubtil@nvidia.com

NVIDIA

Getting Started with CUDA

- Requirements:
  - CUDA-capable GPU
  - C compiler
    - Windows: Microsoft Visual Studio
      (Express Edition is fine --- but mind the license!)
    - Linux: gcc 4.x
    - Mac OS X: Xcode 3.2.x
  - CUDA drivers and toolkit
  - Your favorite text editor

IDEs for CUDA

- Pretty much whatever C IDE you are comfortable with
- Compile with `nvcc` instead of the native compiler
    - .c, .cpp, .cu files should be handled correctly without intervention

- Windows: Parallel Nsight
    - Integrated build and debug environment for Visual Studio
    - Not really a requirement

CUDA Basics: Driver vs. Runtime API

- Driver API
  - Pure low-level C API (much like OpenGL)
  - No syntax extensions
  - Verbose and not really very friendly
- Runtime API
  - Implemented on top of driver API
  - Extensions to C syntax (think OpenMP)
  - A lot more friendly with no real loss of flexibility

First Step: Hello World!

```
__global__ void empty_kernel(void) {
}

int main( void ) {
    empty_kernel<<<1, 1>>>();
    printf("Hello World!\n");
    return 0;
}

nvcc -o hello hello.cu
```

GPU Kernel

```
__global__ void empty_kernel(void) {
}
```

- `__global__` means "will run on GPU when called from CPU"
- Compiler automatically generates GPU code for the function
- Also available: `__device__` (function can be called on GPU but not on CPU)

Kernel Launch Syntax

```
empty_kernel<<<1, 1>>>();
```

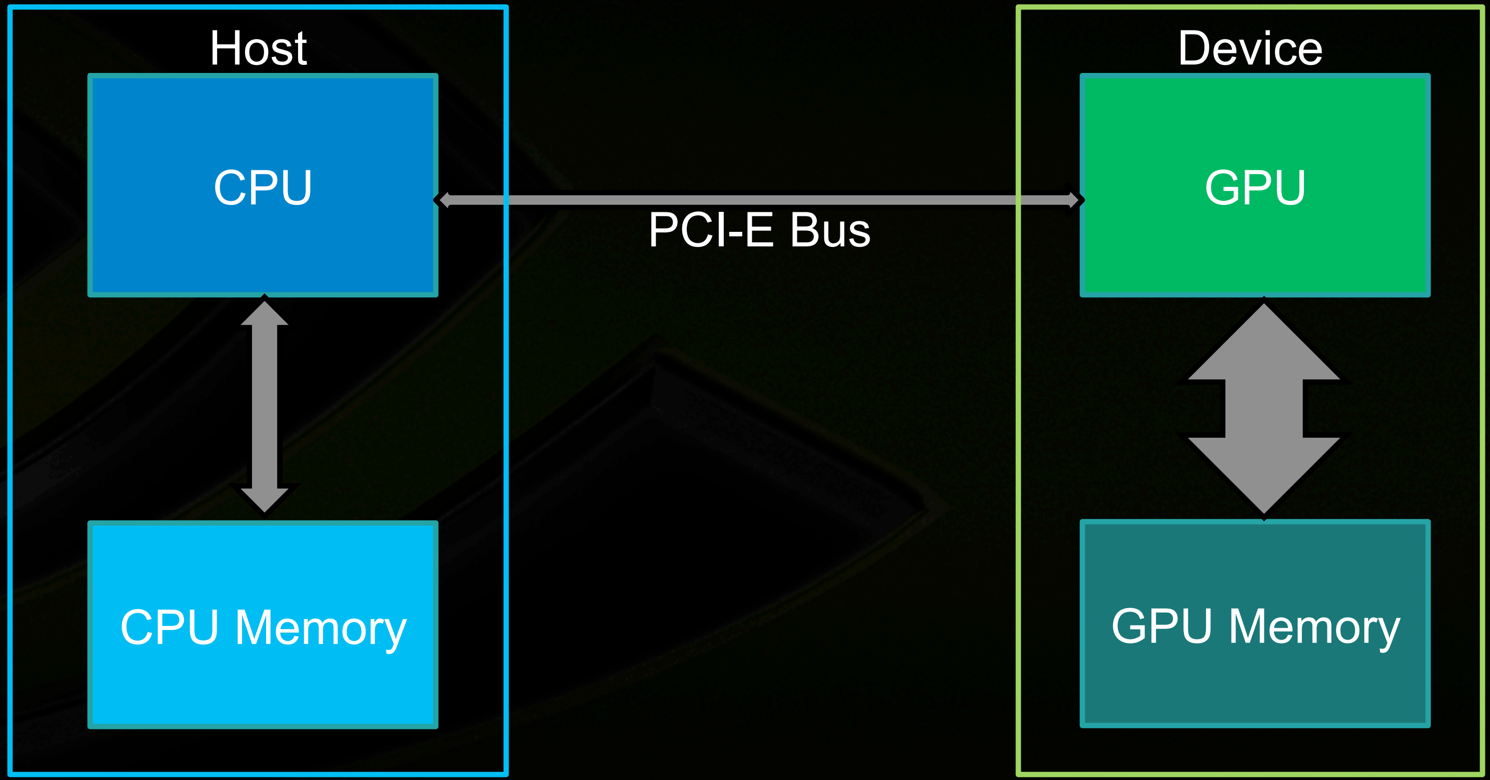
- Automatically invokes GPU code
- Kernels run on a “grid”
- Launch syntax specifies the grid
- Function parameters work just like in C

Second Step: 2 + 7

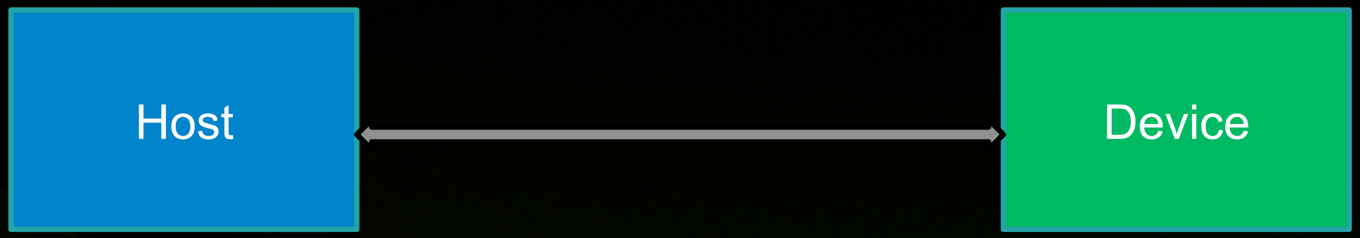
```
__global__ void add(int a, int b, int *c) {
    *c = a + b;
}

int main( void ) {
    int c, *dev_c;
    HANDLE_ERROR( cudaMalloc(&dev_c, sizeof(int)) );
    add<<<1, 1>>>(2, 7, dev_c);
    HANDLE_ERROR( cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost) );
    printf("2 + 7 = %d\n", c);
}
```

Host vs Device Memory

- Device: your graphics/compute card
- Host: the rest of your computer

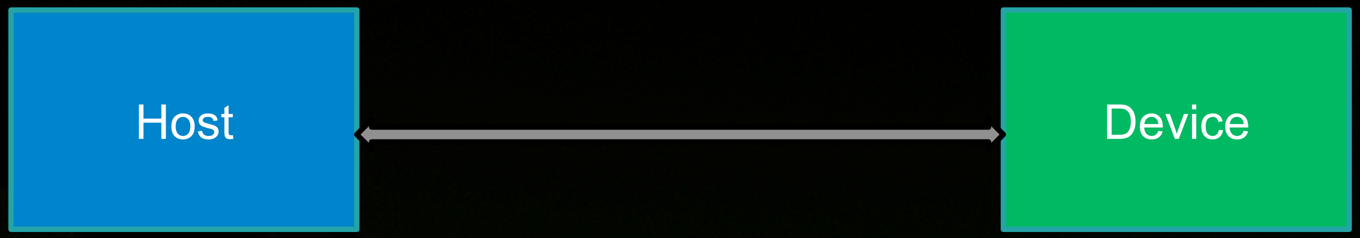


Host vs Device Memory



- NUMA: separate host and device address spaces
  - Device can not read/write host memory (well… sort of)
  - Host can not read/write device memory (this one is mostly true)
- Full 32-bit device pointers
- Device-specific memory management functions

Device Pointers



- NOT valid on host!
  - … and host pointers not valid on device either!

- Pointer arithmetic is allowed
- No bounds checking
  - You can segfault on the GPU and your process can get killed for that!)

Become a CUDA Memory Management Expert

- cudaMalloc: allocate on-device memory
  - Runs on host, returns device pointer to host
- cudaMemcpy:
  - Copy between host and device
  - Copy between two device buffers
  - Copy between different devices (on a multi-GPU system)
  - Called on host, executes on host and on device
  - (Also available: cudaMemset)
- cudaFree: free device memory

2 + 7 = ?...

```
int main( void ) {
    int c, *dev_c;
    HANDLE_ERROR( cudaMalloc(&dev_c, sizeof(int)) );
    add<<<1, 1>>>(2, 7, dev_c);
    HANDLE_ERROR( cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost) );
    printf("2 + 7 = %d\n", c);
}
```

2 + 7 = 27...

Device Pointer

Host Memory

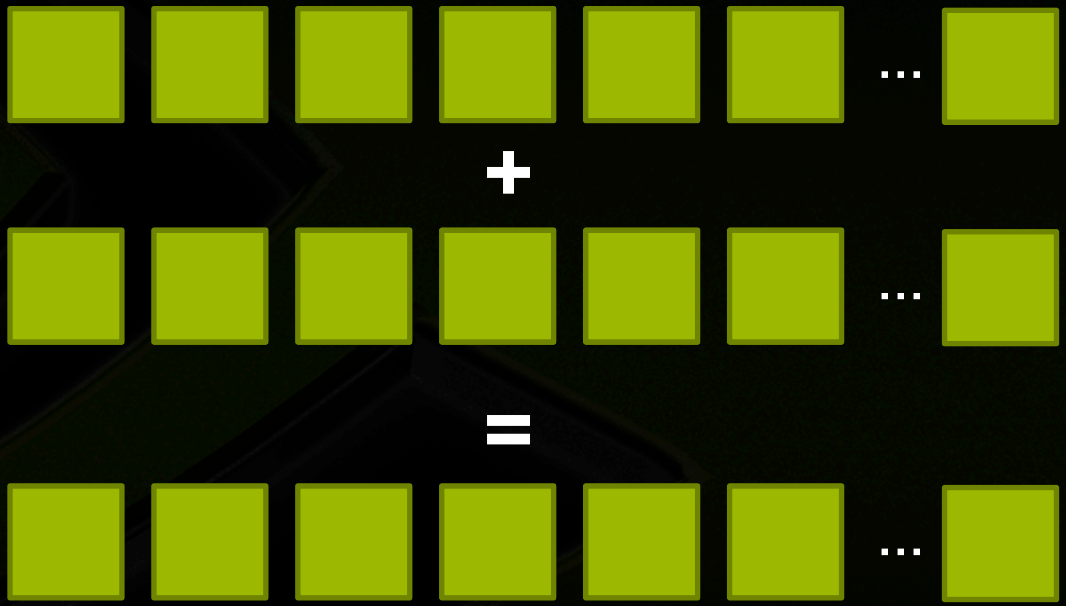
```
int main( void ) {
    int c, *dev_c;
    HANDLE_ERROR( cudaMalloc(&dev_c, sizeof(int)) );
    add<<<1, 1>>>(2, 7, dev_c);
    HANDLE_ERROR( cudaMemcpy(&c, dev_c, sizeof(int),
                             cudaMemcpyDeviceToHost) );
    printf("2 + 7 = %d\n", c);
}
```

2 + 7 = 9!

```
int main( void ) {
    int c, *dev_c;
    HANDLE_ERROR( cudaMalloc(&dev_c, sizeof(int)) );
    add<<<1, 1>>>(2, 7, dev_c);
    HANDLE_ERROR( cudaMemcpy(&c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost) );
    cudaFree(dev_c);
    printf("2 + 7 = %d\n", c);
}
```

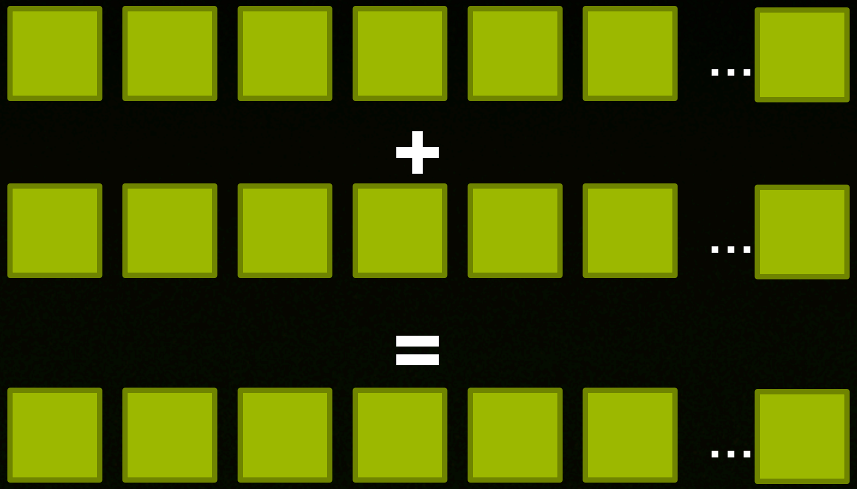
Vector Sum

Problem: compute the sum of two integer vectors



Serial CPU Version

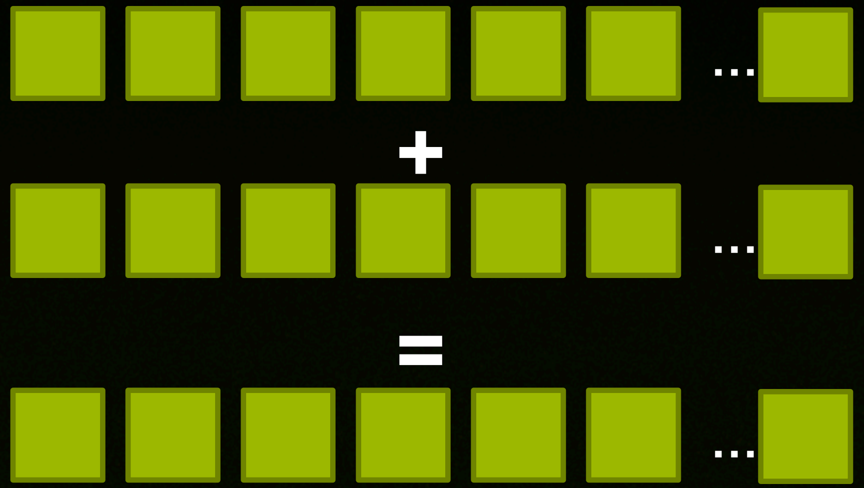
```
void sum(int *a,
         int *b, int *c)
{
  for(int i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
  }
}
```



Serial Approach on the GPU

Exercise: run our vector sum on the GPU using a single thread

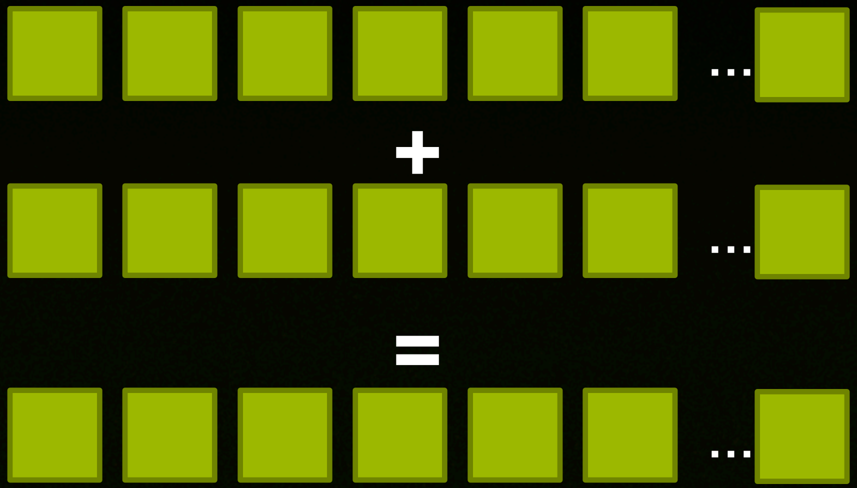
- **Implement single-threaded GPU kernel**
  - **Hint: *very* similar to CPU version**
- **Allocate and fill GPU memory**
- **Launch kernel on the GPU**
- **Read results back**



Serial Approach on the GPU

To run on GPU, just add __global__...

```
__global__ void sum(int *a,
              int *b, int *c)
{
   for(i = 0; i < N; i++) {
      c[i] = a[i] + b[i];
   }
}
```



Serial Approach on the GPU

… do the memory allocation on the GPU and copy the data …

```
int *dev_a, *dev_b, *dev_c;
HANDLE_ERROR( cudaMalloc(&dev_a, sizeof(int) * N));
HANDLE_ERROR( cudaMalloc(&dev_b, sizeof(int) * N));
HANDLE_ERROR( cudaMalloc(&dev_c, sizeof(int) * N));
cudaMemcpy(dev_a, host_a, sizeof(int) * N,
           cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, sizeof(int) * N,
           cudaMemcpyHostToDevice);
```

Serial Approach on the GPU

… then launch the kernel…

```
sum<<<1, 1>>>(dev_a, dev_b, dev_c);
```

… and finally read the results back.

```
cudaMemcpy(host_c, dev_c, sizeof(int) * N,
           cudaMemcpyDeviceToHost);
```

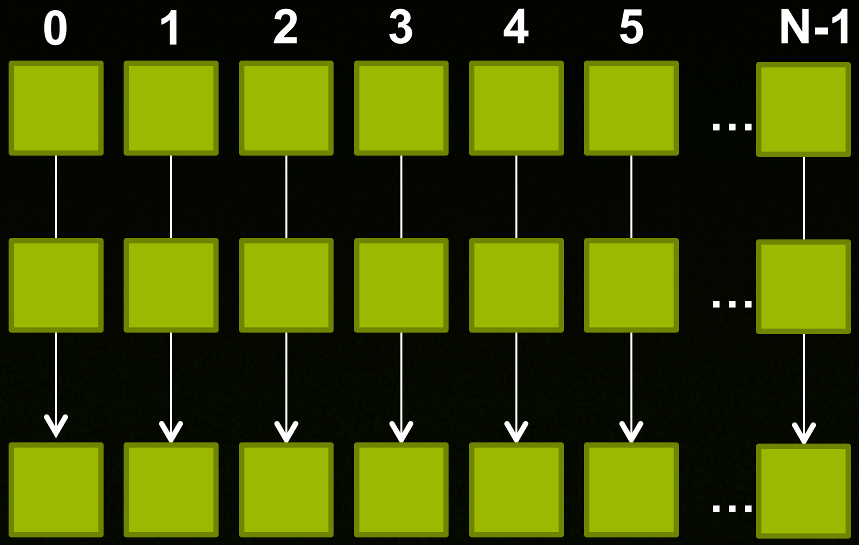
Running in Parallel

Going Parallel

- CPUs: single-threaded, complex code
  - Generally one, sometimes two threads per CPU core
  - Hardware copes very well with “bad” code
- GPUs: massively multi-threaded, simple code
  - Not that great at branching (but getting better!)
  - Run thousands of threads per GPU core very efficiently
  - Brute force makes up for the limitations ☺

Parallel Approach on the GPU

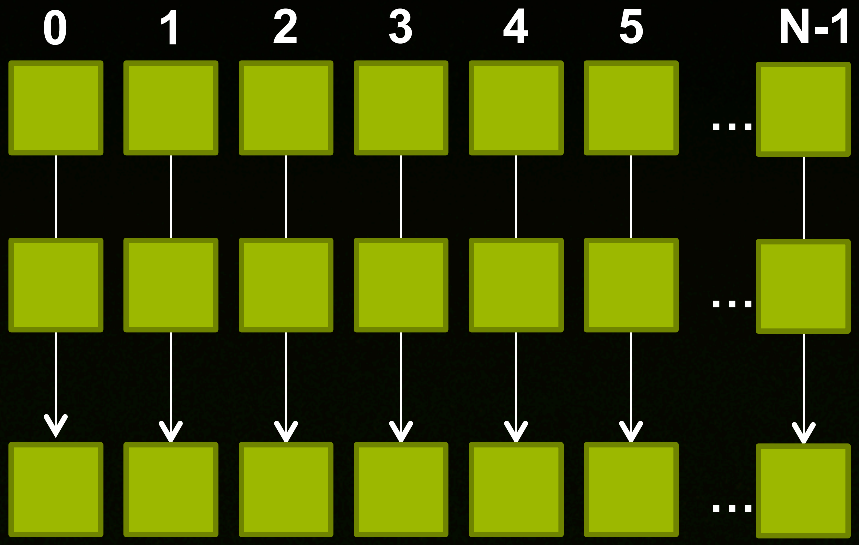
- Run one thread per index
- Each thread computes
  a single element of the output



Parallel Approach on the GPU

Kernel launch syntax:

```
sum<<<N, 1>>>(dev_a, dev_b,
              dev_c);
```



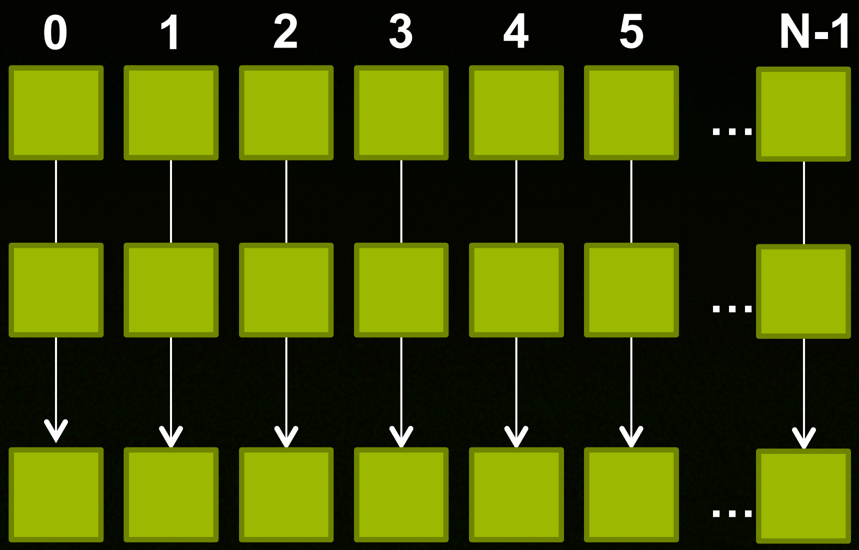
Parallel Approach on the GPU

Kernel launch syntax:

```
sum<<<N, 1>>>(dev_a, dev_b,
              dev_c);
```

Blocks

Threads per block



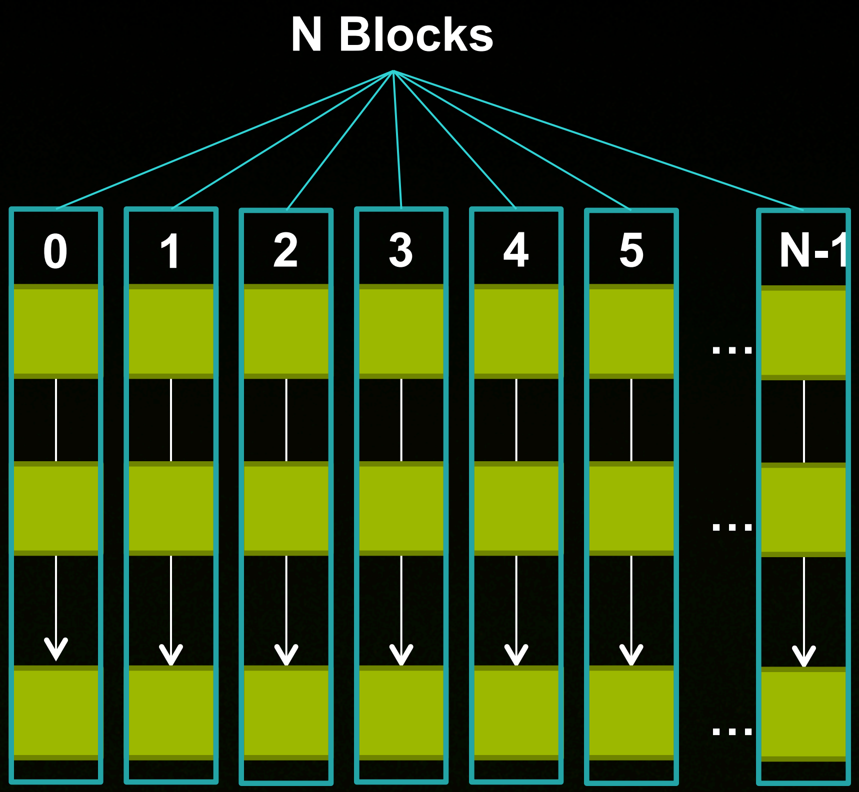
Parallel Approach on the GPU

Kernel launch syntax:

```
sum<<<N, 1>>>(dev_a, dev_b,
              dev_c);
```

Blocks

Threads per block



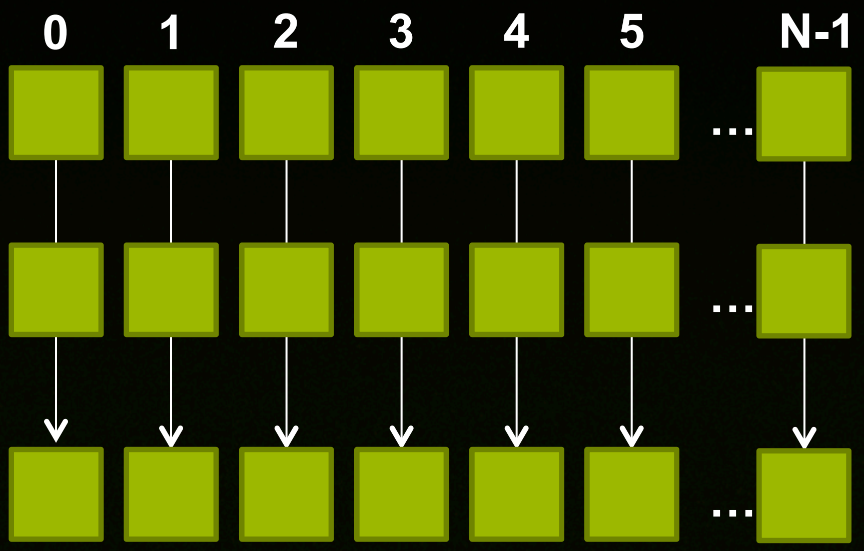
One thread per block

Parallel Approach on the GPU

Kernel function runs once per element

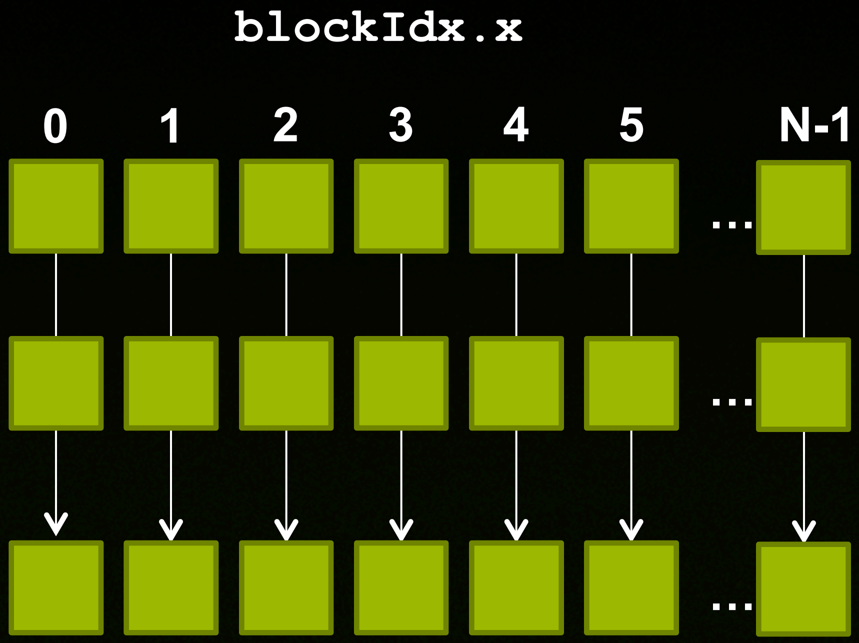
- How do we know which element to process?

A: CUDA tells us through `blockIdx`



Parallel Approach on the GPU

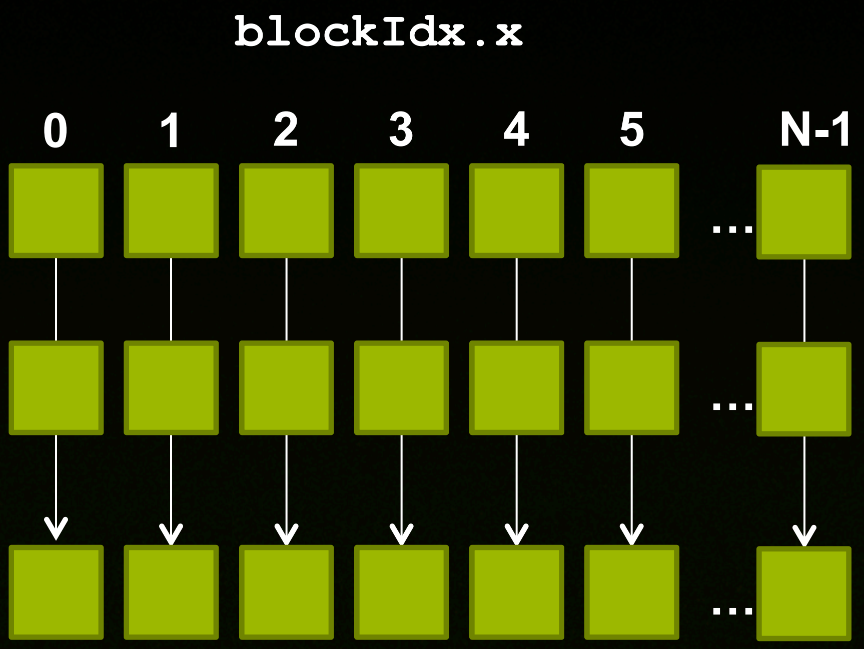
```
__global__ void sum(int *a,
            int *b, int *c)
{
      /* magic here */
}
```



Exercise: implement the parallel version of the kernel

Parallel Approach on the GPU

```
__global__ void sum(int *a,
             int *b, int *c)
{
  int i = blockIdx.x;
  c[i] = a[i] + b[i];
}
```



Q: How big can the vector be?

GPU Work Partitioning

GPU Work Partitioning

- Each GPU kernel runs in thousands of threads in parallel
- Sets of threads (“thread blocks”) will share certain hardware resources
- CUDA runtime mimics the hardware architecture
  - Kernels run by blocks of threads
  - Launch syntax specifies both

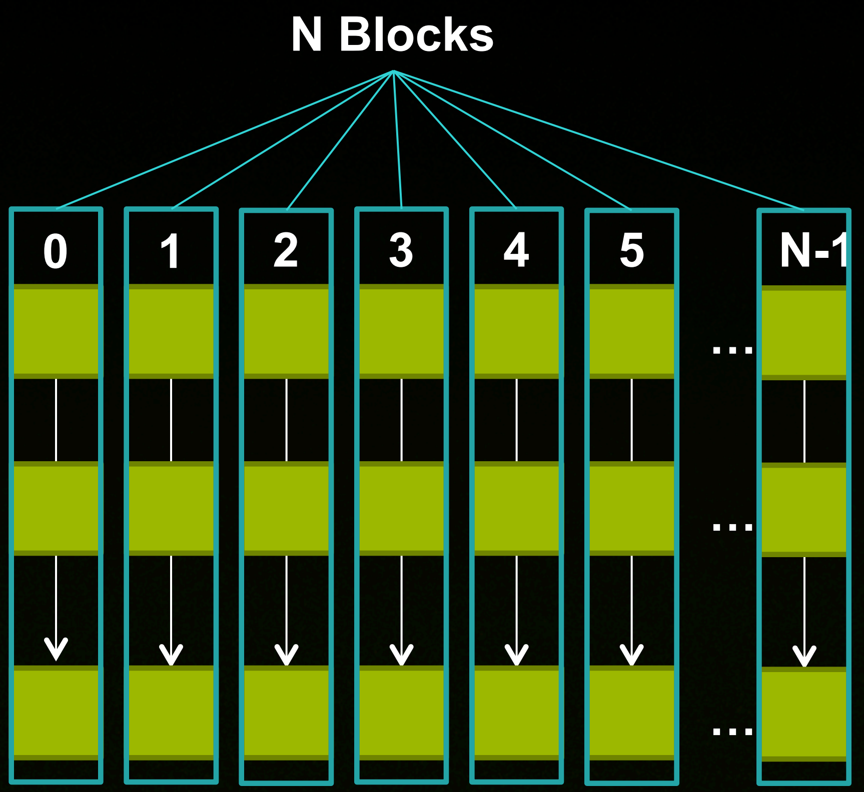
GPU Work Partitioning

Kernel launch syntax:

```
sum<<<N, 1>>>(dev_a, dev_b,
              dev_c);
```

Blocks

Threads per block



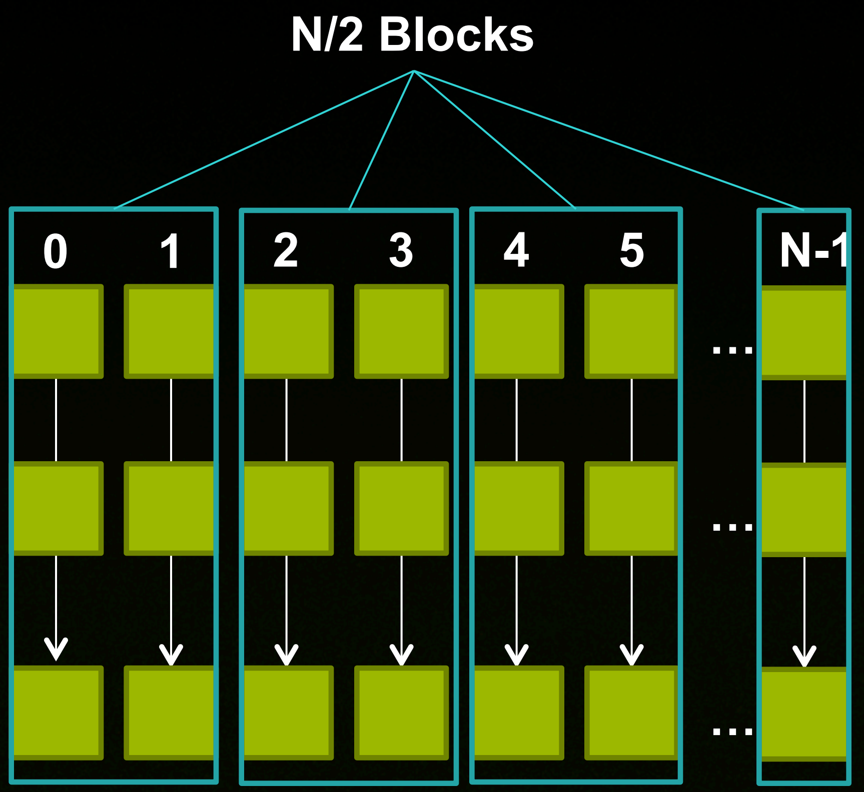
One thread per block

GPU Work Partitioning

Kernel launch syntax:

```
sum<<<N/2, 2>>>(dev_a, dev_b,
                dev_c);
```

2 threads/block * N/2 blocks
= N threads



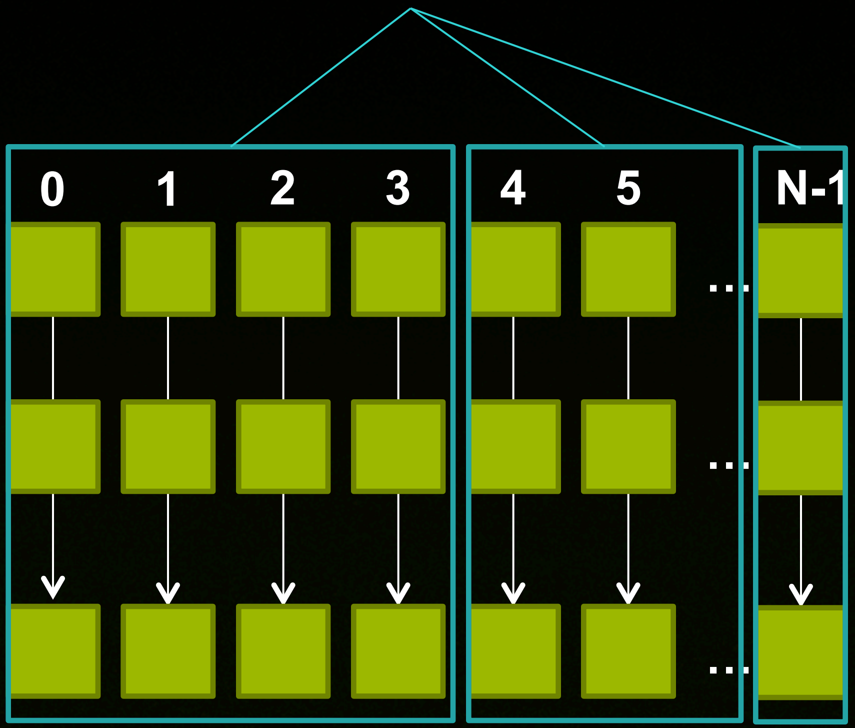
2 threads per block

GPU Work Partitioning

Kernel launch syntax:

```
sum<<<N/4, 4>>>(dev_a, dev_b,
                dev_c);
```

4 threads/block * N/4 blocks
= N threads



4 threads per block

GPU Work Partitioning: Why?

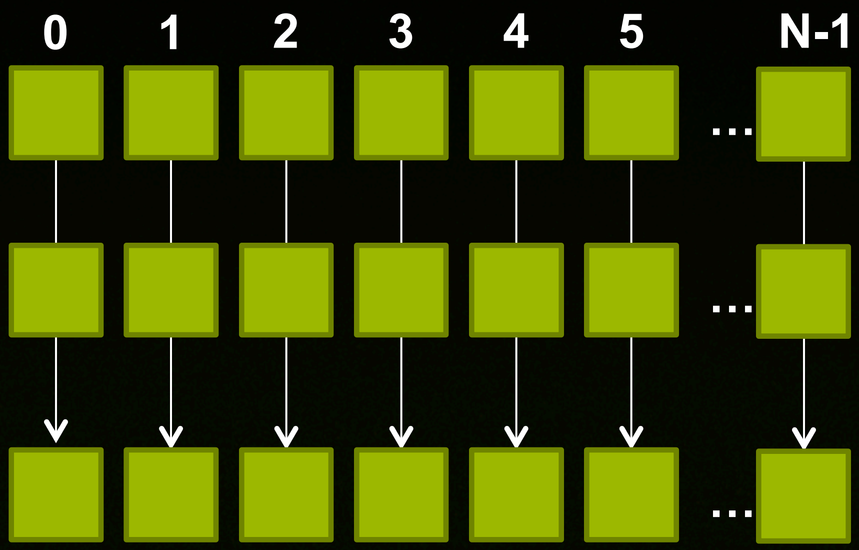
- Immediate concern: limited number of blocks per launch
  - Vector sum example breaks!
- Efficiency
  - Maximize hardware occupancy
  - Make use of shared memory per block
  - Benefit from memory access coalescing
  - Lower instruction fetch pressure

GPU Work Partitioning

Kernel function still runs once per element

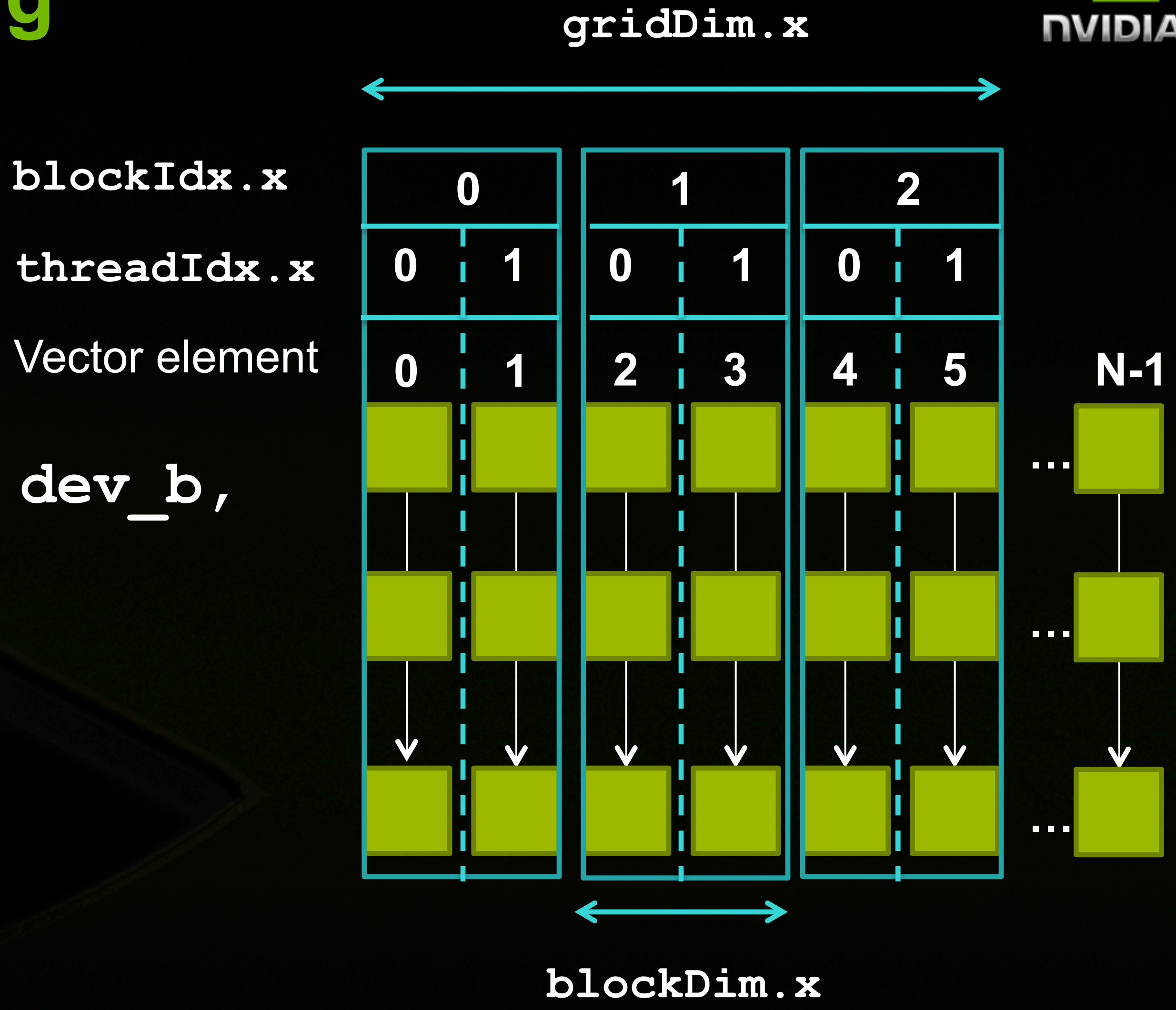
- How do we know which element to process?

A: `blockIdx`, `blockDim`, `threadIdx`.

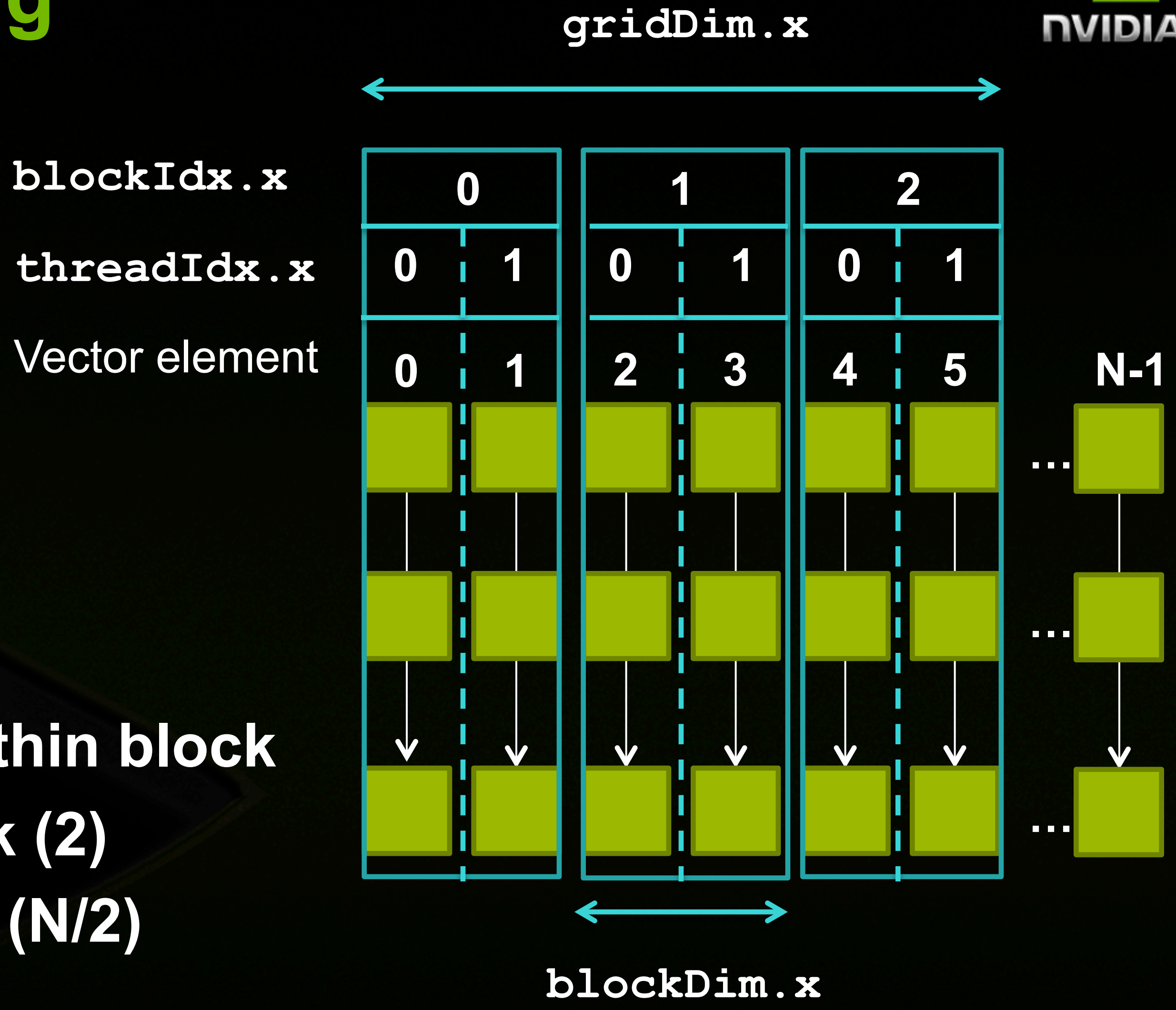


GPU Work Partitioning

```
sum<<<N/2, 2>>>(dev_a, dev_b,
                dev_c);
```



GPU Work Partitioning



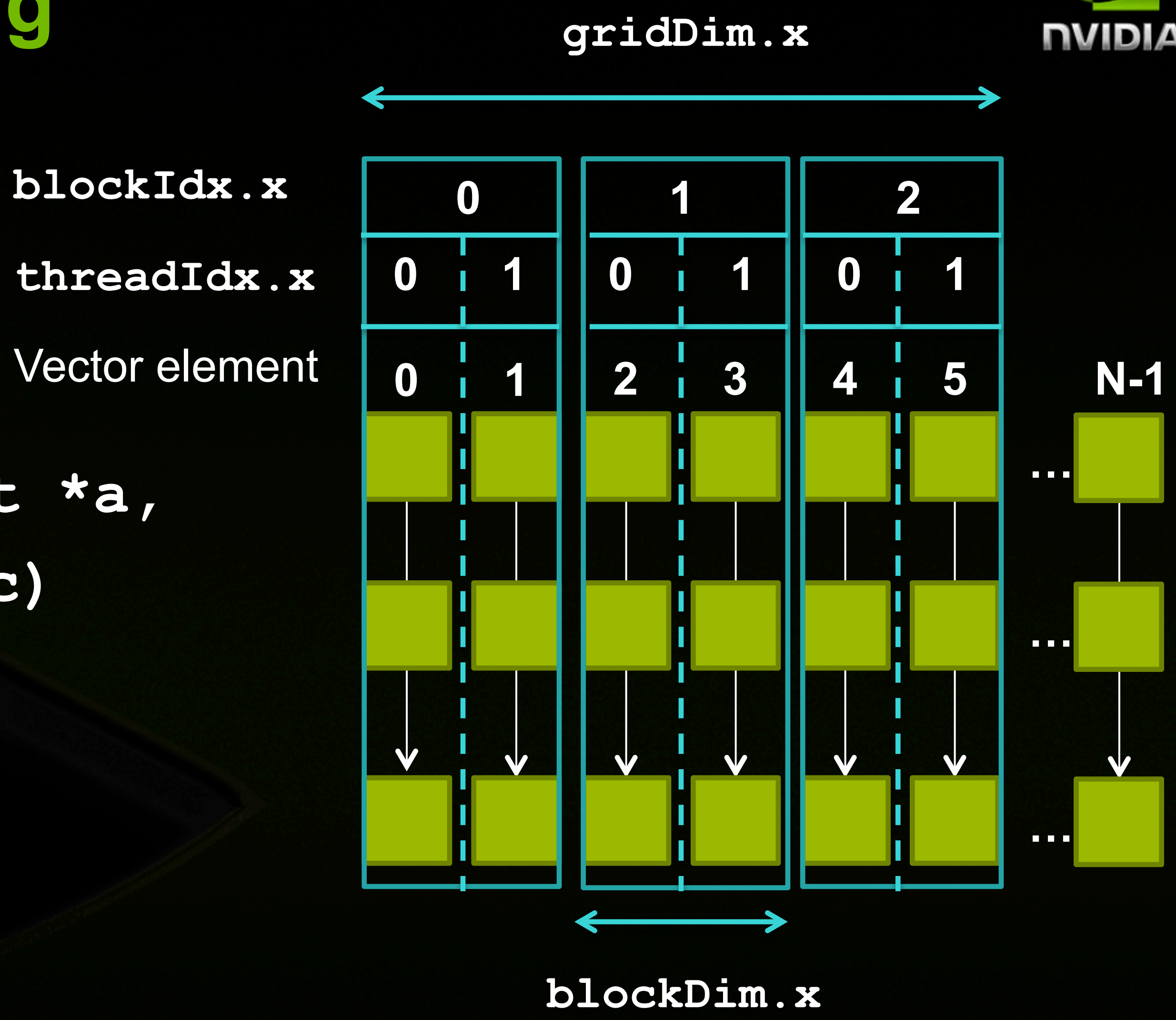
`blockIdx`: block index

`threadIdx`: thread index within block

`blockDim`: threads per block (2)

`gridDim`: blocks per launch (N/2)

GPU Work Partitioning

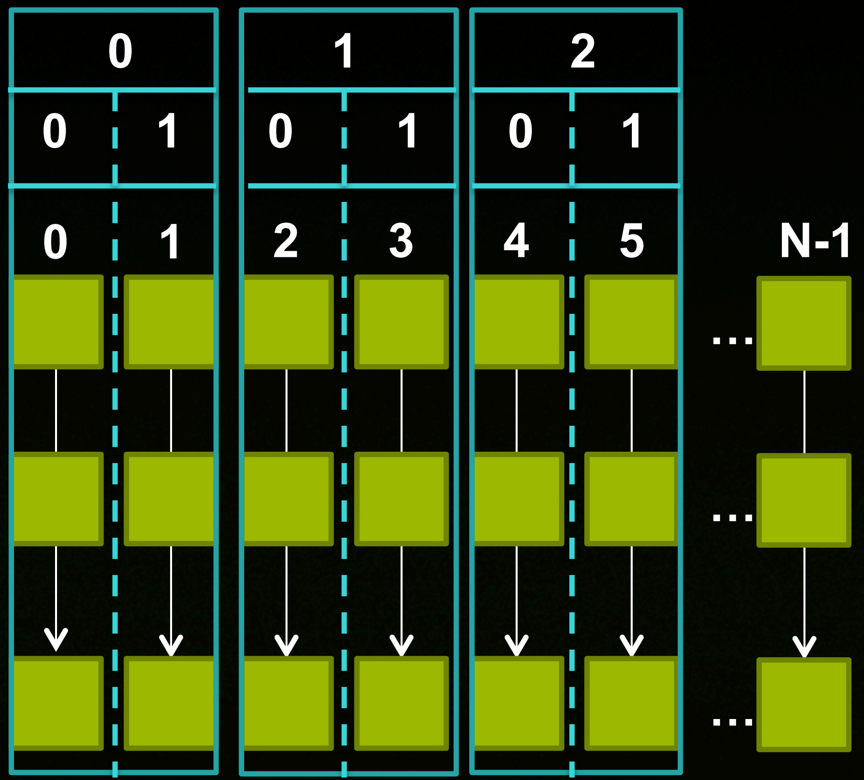



```
__global__ void sum(int *a,
                    int *b, int *c)
{
  /* ??? */
}
```

Exercise: make it work!

GPU Work Partitioning

```
__global__ void sum(int *a,
           int *b, int *c)
{
   int i;
   i = blockIdx.x * blockDim.x +
       threadIdx.x;
   c[i] = a[i] + b[i];
}
```



Kernel Launch Limits

How many threads can you launch at once now?

A: Many, but not infinitely many!

Kernel Launch Limits

- Limited number of threads per block
  - GF100: maximum is 1024 threads
  - Depends on shared resource usage
- Limited number of blocks per launch
  - GF100: maximum is 65536 blocks
- Total number of threads is still limited
- CUDA provides an API to query these limits

Querying Device Limits

```
cudaDeviceProp prop;
HANDLE_ERROR( cudaDeviceGetProperties(&prop, 0) );
```

Interesting fields:

- prop.name (device name string)
- prop.totalGlobalMem (size of device memory)
- prop.maxThreadsPerBlock
- prop.gridDim[0] (max blocks per launch)

See the deviceQuery example from the CUDA SDK

Arbitrarily long vector sums

Modify the vector sum example to handle arbitrary size vectors while adhering to your particular device's limits.

(Hint: you can do more work per thread. But you don't have to.)

Arbitrarily long vector sums

Run the kernel multiple times…

```
__global__ void sum(int *a, int *b, int *c, int o) {
  int i = blockIdx.x * blockDim.x + threadIdx.x + o;
  if (i < N)
    c[i] = a[i] + b[i];
}
…
for(j = 0; j < N; j += 256 * 256)
  sum<<<256, 256>>>(dev_a, dev_b, dev_c, j);
```

Arbitrarily long vector sums

… or process more elements per thread (how does this work?)

```
__global__ void sum(int *a, int *b, int *c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   while (i < N) {
      c[i] = a[i] + b[i];
      i += gridDim.x * blockDim.x;
   }
}
…
sum<<<B, T>>>(dev_a, dev_b, dev_c);
```

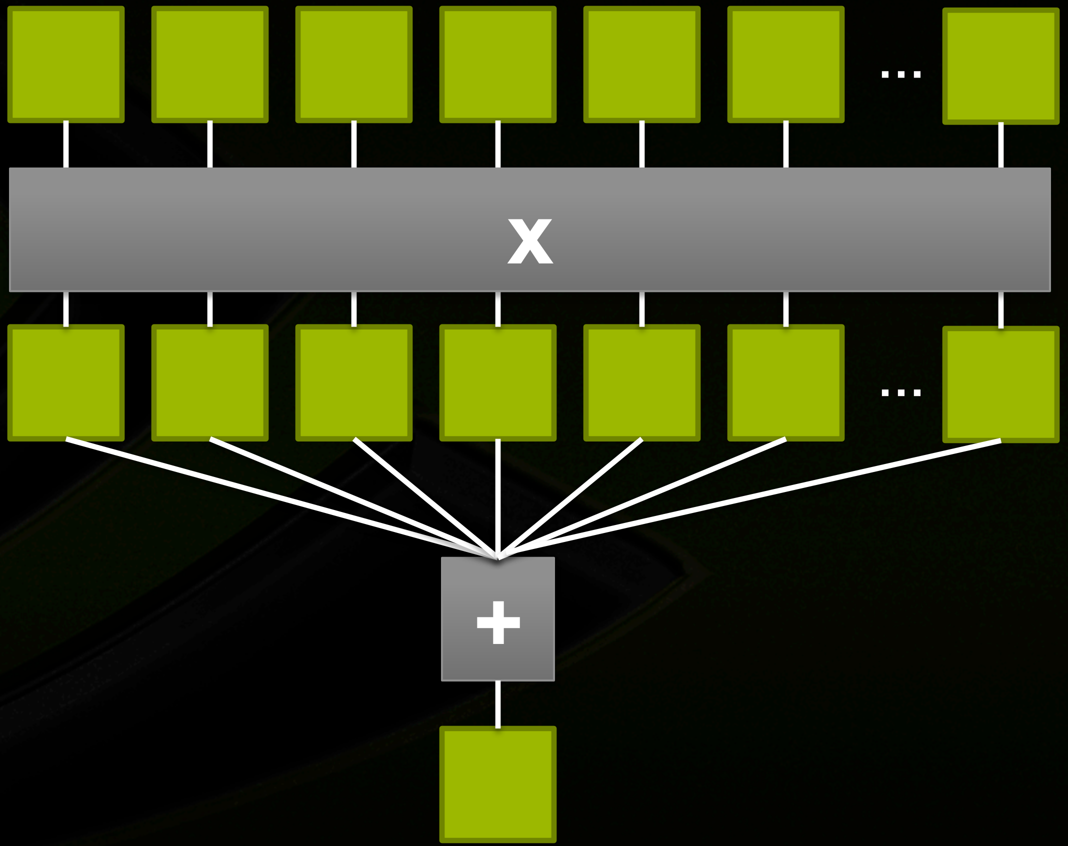
GPU Reductions

Vector Dot Product

Given two arbitrarily long integer vectors, compute their dot product.

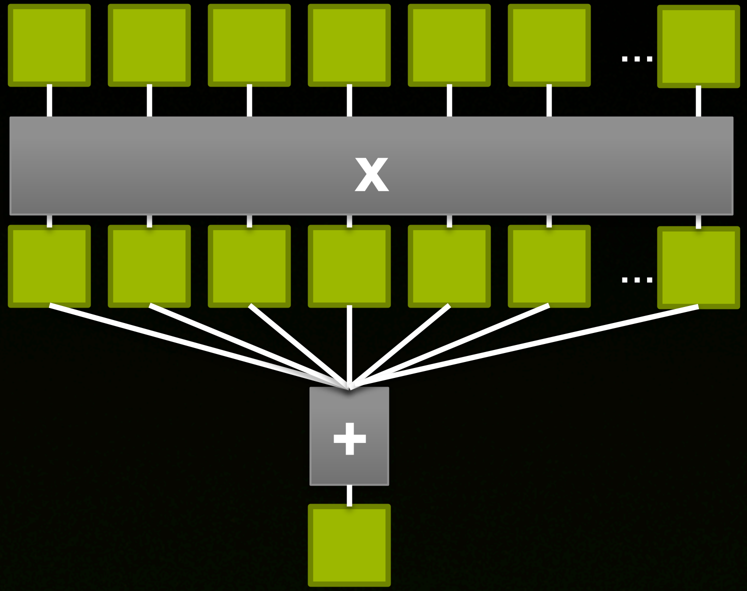
Vector Dot Product

A[0] * B[0] + A[1] * B[1] + .. + A[N-1] * B[N-1]



Parallel Vector Dot Product

- Two steps
  - Multiplication
  - Summation
- Summation is a “reduction” operation
  - Many input elements for one output element



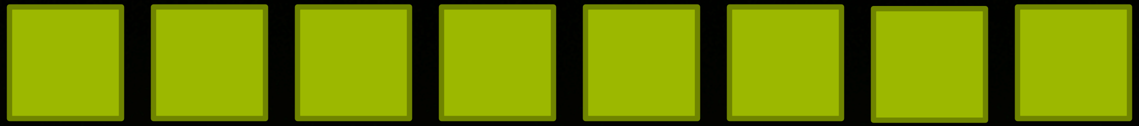
Exercise: implement dot product on the GPU.
(Do the reduction step on the CPU for now.)

Parallel Vector Dot Product: Reduction

- Obtain one output element from many input elements
  - Input: a vector
  - Output: a scalar (sum of elements of the vector)
- Solution: use many threads and multiple steps

Reduction

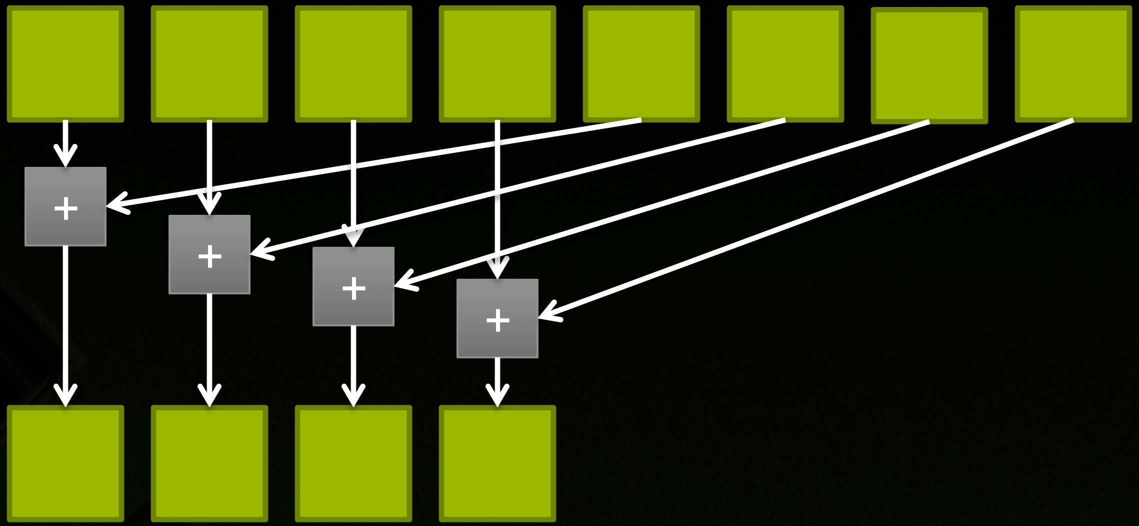
Input:



Reduction

Input:

Reduction step 1:

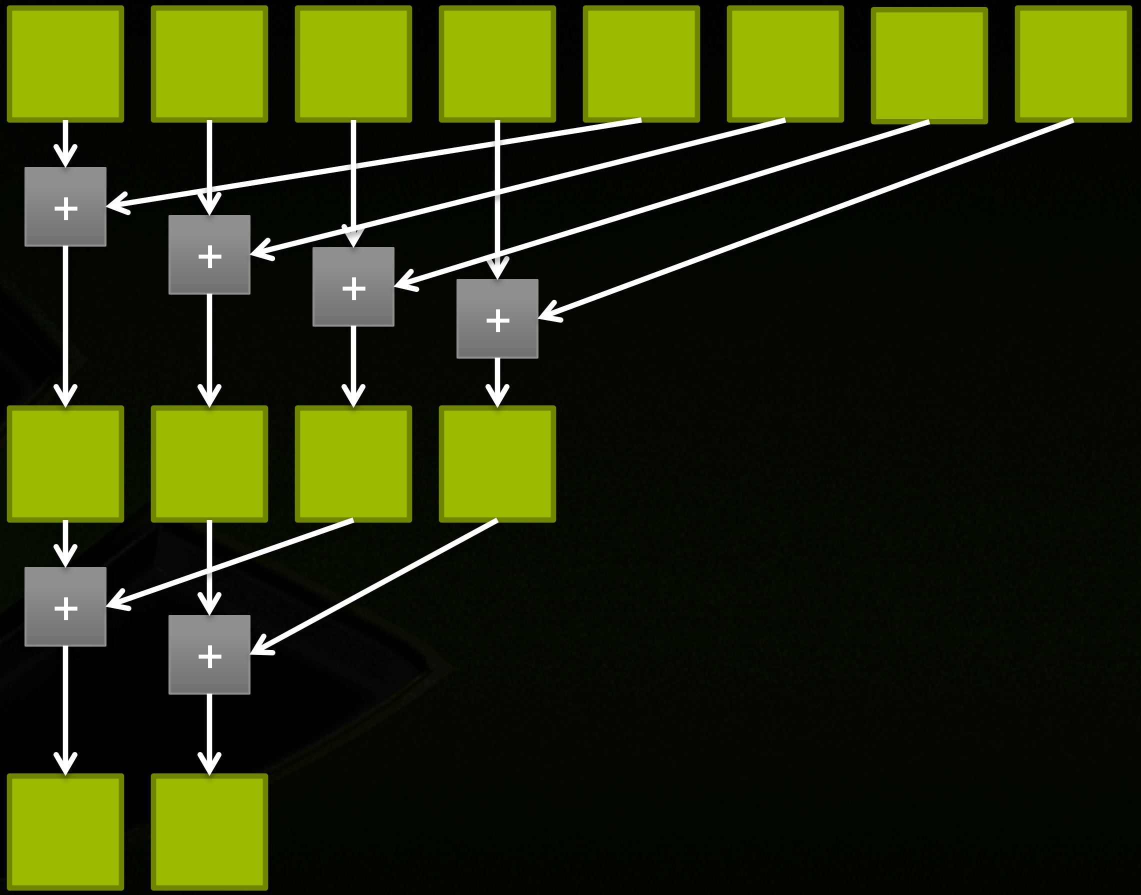


Reduction

Input:

Reduction step 1:

Reduction step 2:

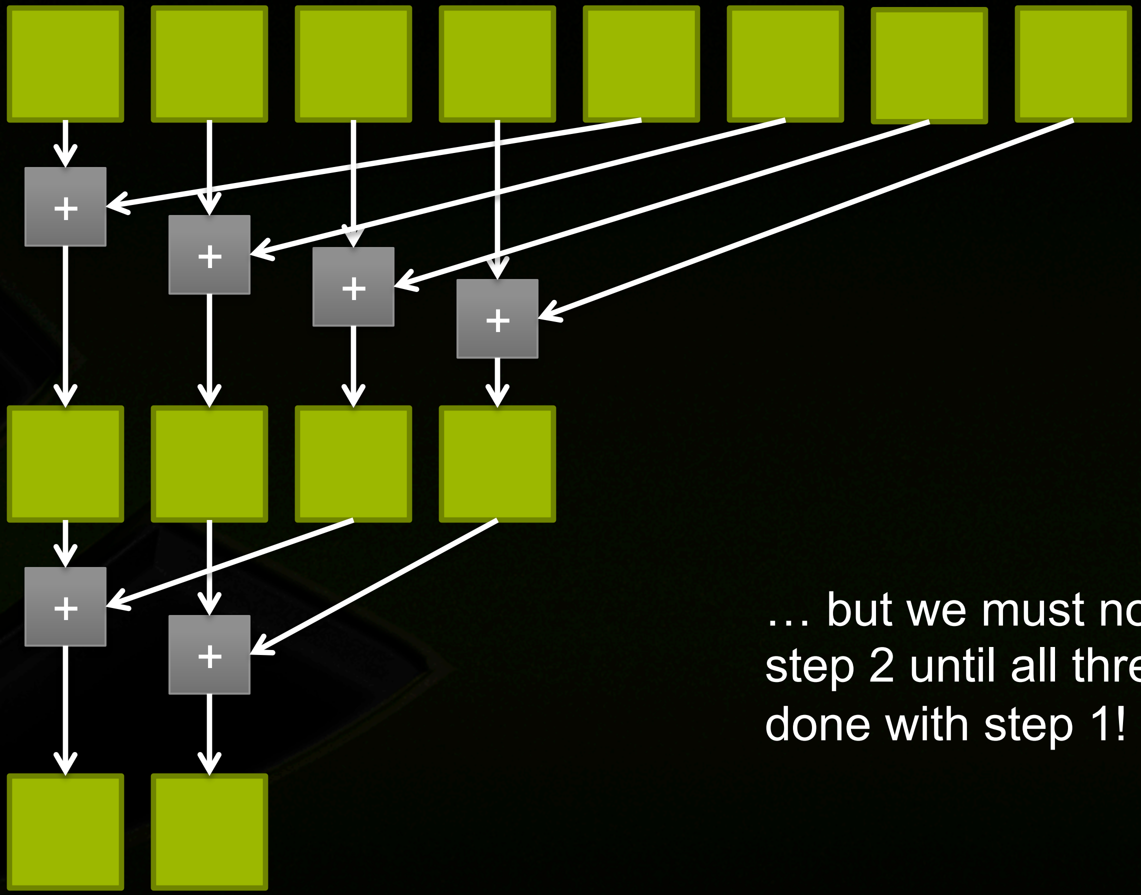


Reduction

Input:

Reduction step 1:

Reduction step 2:



… but we must not start step 2 until all threads are done with step 1!

Thread Cooperation: Barrier

CUDA implements a thread barrier: `__syncthreads()`

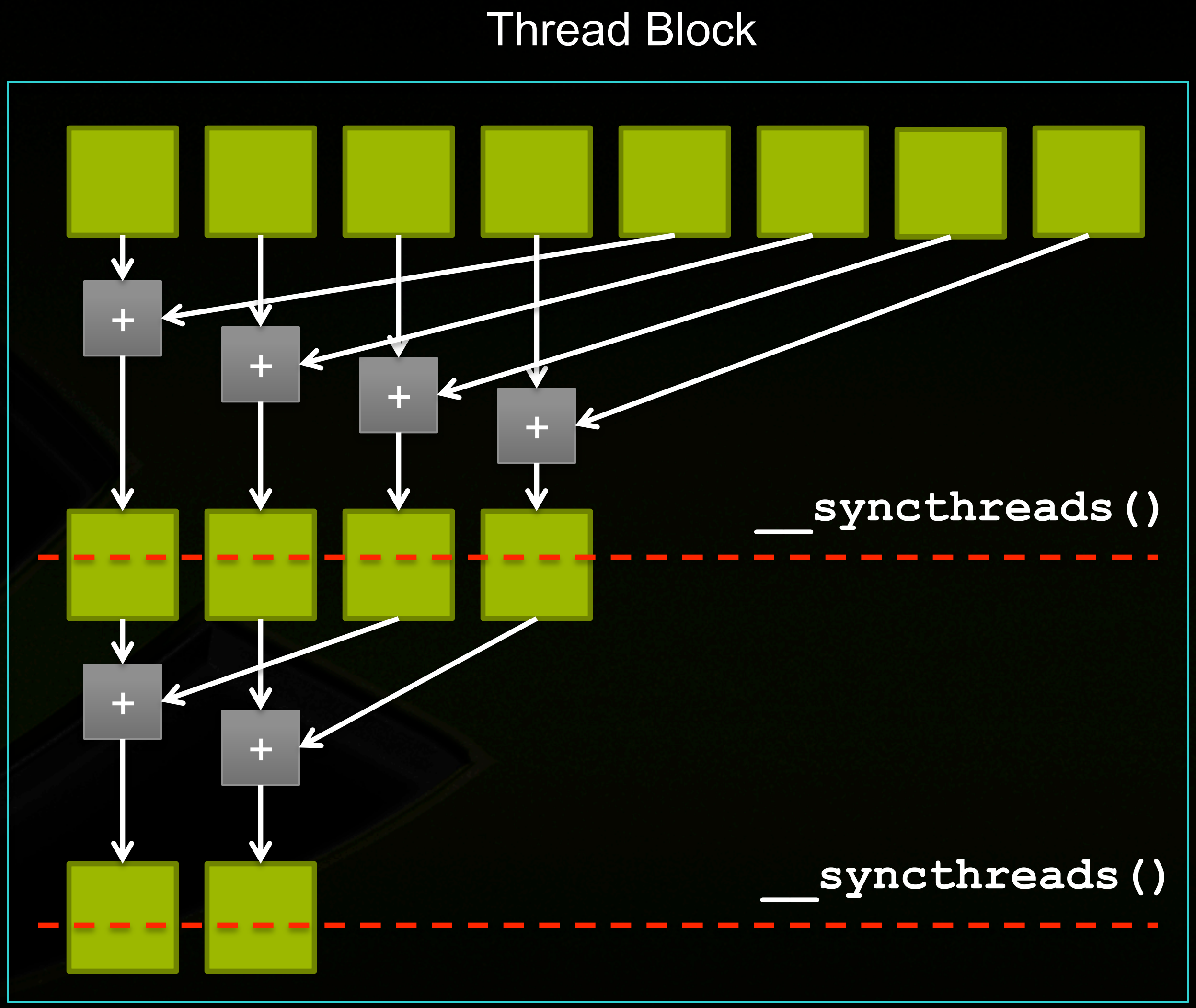
- Threads wait inside `__syncthreads()` until *all* threads *within the same block* reach `__syncthreads()`
  - If one thread in the block calls `__syncthreads()`, then ALL threads on the same block must call `__syncthreads()`
- No synchronization across blocks
  - CUDA requires that blocks be completely independent
  - Order of execution of blocks is arbitrary --- not guaranteed to be scheduled in order

Reduction

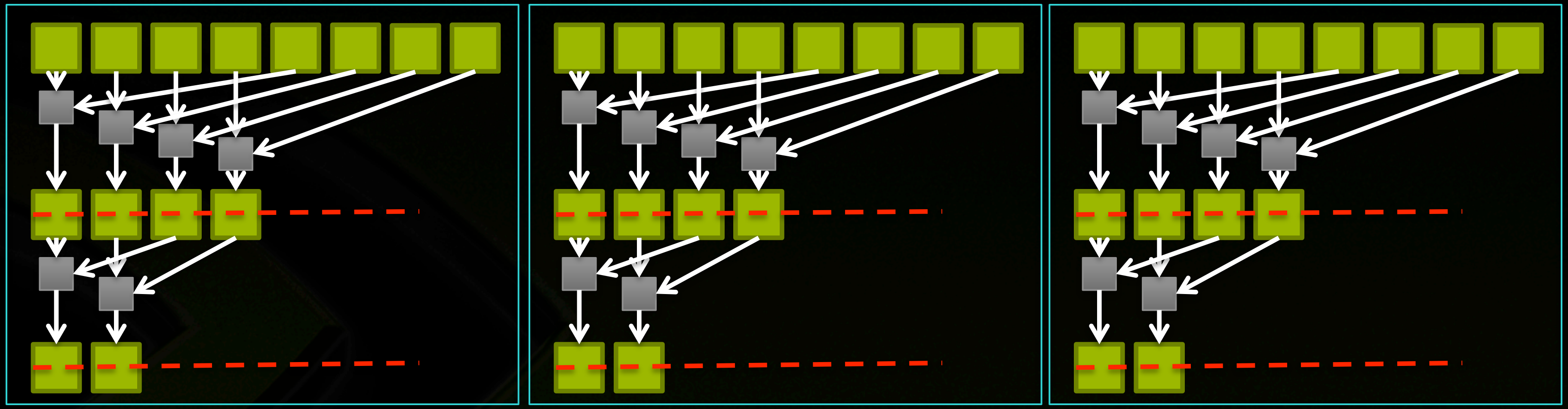
Input:

Reduction step 1:

Reduction step 2:



Reduction

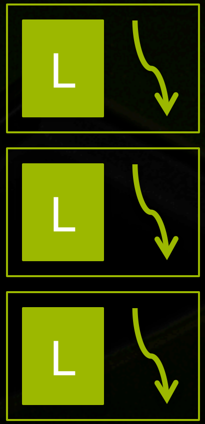


Exercise: implement GPU reduction for the dot product.

GPU Memory Hierarchy and Thread Cooperation

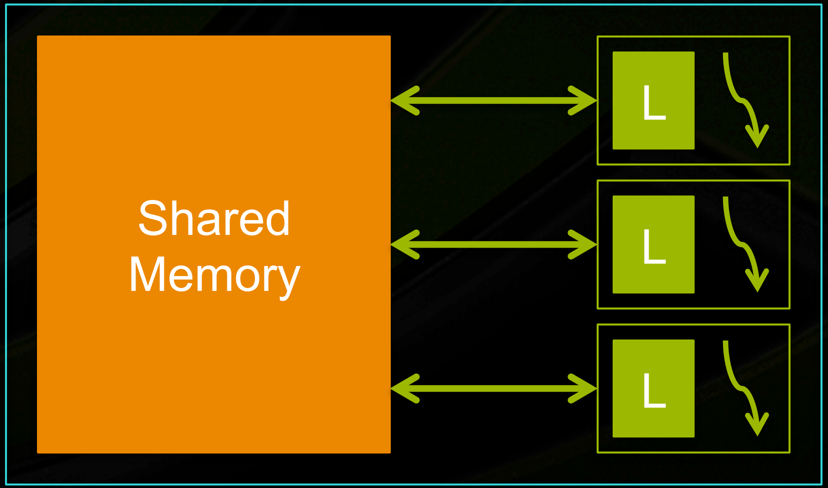
GPU Memory Hierarchy

- Each thread has local storage
  - On-chip, low-latency, very fast
  - Implemented as registers with spill over into L1 cache
  - GPU function variables live here



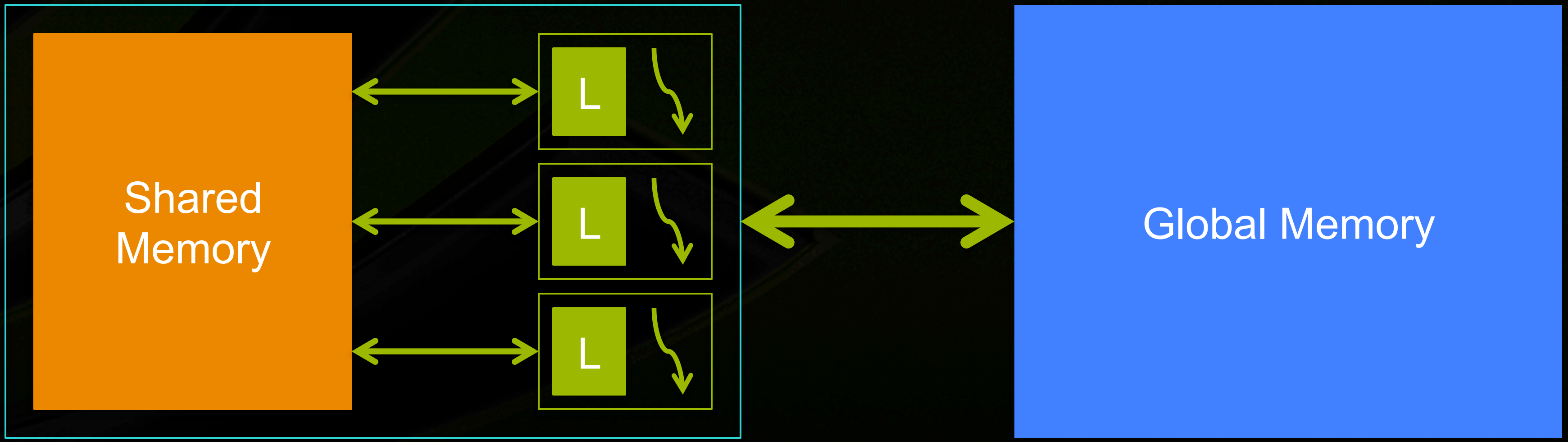
GPU Memory Hierarchy

- **Thread blocks have shared memory**
  - **On-chip, fast**
  - **Implemented as L2 cache**
  - **GPU variables declared as `__shared__` live here**



GPU Memory Hierarchy

- GPU has access to global memory
  - RAM on the graphics card
  - Off-chip, much slower
  - cudaMalloc allocates memory here



Thread Cooperation via Shared Memory

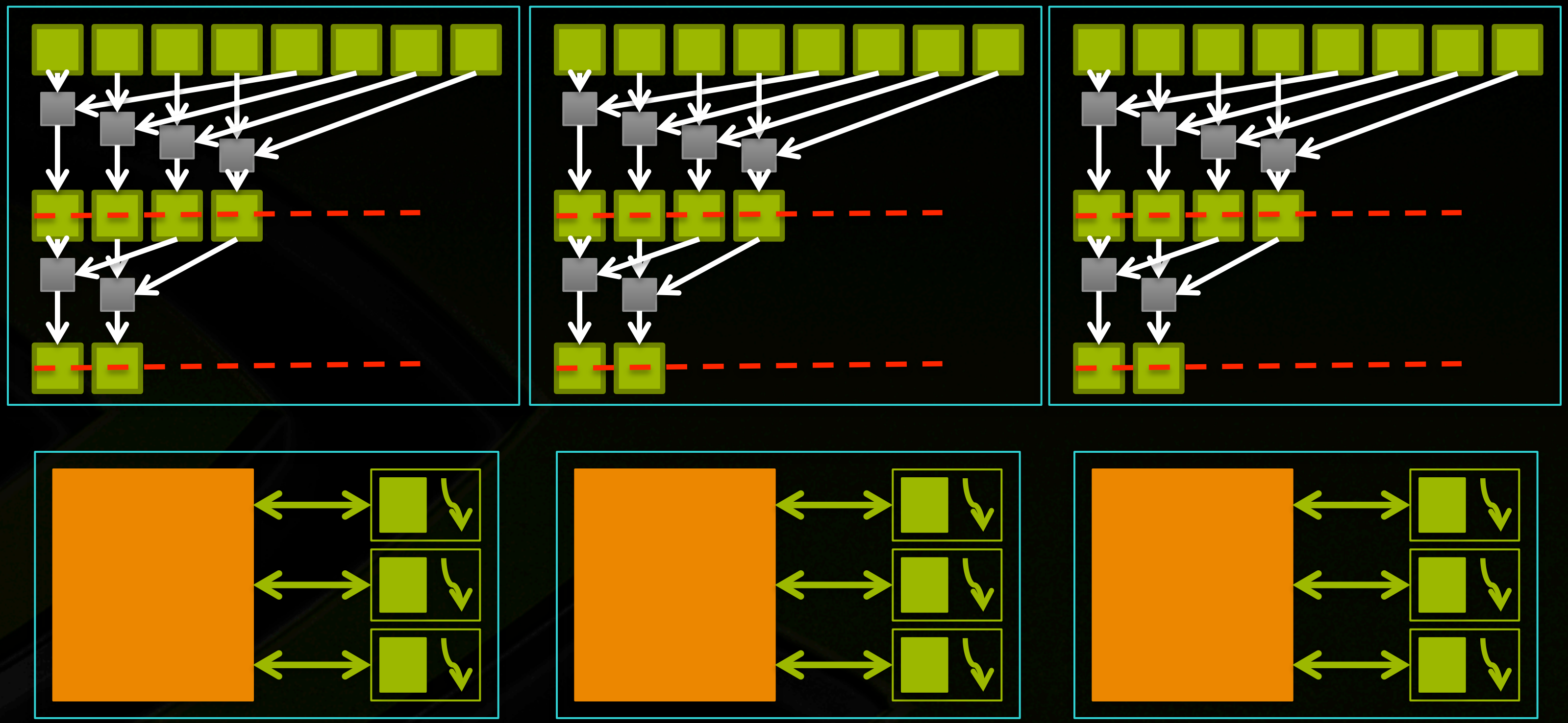
Shared memory: common storage area visible to all threads within a block

- Just declare a variable or array with the __shared__ attribute…

```
__shared__ int localStorage[THREADS_PER_BLOCK];
```

- … and then make sure to use it wisely!
  - Limited amount of shared memory available

Dot Product with Shared Memory



Exercise: make use of shared memory instead of global memory for the reduction in the dot product.

Atomic Operations

Thread Cooperation: Atomics

- __syncthreads() is not useful for synchronizing across thread blocks

- CUDA provides atomic memory access functions
  - Read-modify-write operations on memory addresses
  - Can operate on global or shared memory

Thread Cooperation: Atomics

```
int atomicAdd(int *address, int val)
```

Reads the old value at address

Computes `val + old` and writes result back to address

Returns the old value

```
int atomicSub(int *address, int val)
```

Same as atomicAdd, but performs a subtraction

Thread Cooperation: Atomics

```
int atomicExch(int *address, int val)
```

Writes val at address, returns old value

```
int atomicCAS(int *address, int cmp, int val)
```

Reads old value at address

```
result = (old == cmp ? val : old)
```

Writes result to address, returns old value

Also available: atomicInc, atomicDec, atomicAnd/Or/Xor

Thread Cooperation: Atomics

- When do we use atomics?
  - Synchronizing access to a memory area
  - Allows multiple threads to communicate safely
- When not to use atomics? When you need performance!
  - Atomics serialize threads
  - Reduced parallelism
  - Performance penalty is especially high when using atomics on global memory

Simple Atomics Example: Vector Sum

```
__global__ void vector_sum(int *vector, int *out) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int sum = 0;
  while(i < N) {
    sum += vector[i];
    i += blockDim.x * gridSize.x;
  }
  atomicAdd(out, sum);
}
```

Histogram Computation

Problem: compute the histogram of an array of bytes.

(In other words, count how many bytes in the given data set have each of the 256 possible values.)

Histogram: CPU Solution

```
void compute_histogram(char *data, int *histogram) {
    int c;
    memset(histogram, 0, sizeof(int) * 256);
    for(c = 0; c < N; c++)
        histogram[data[c]]++;
}
```

Now implement the GPU version. And make it fast!

(Hint: shared memory is your friend)