CUDA Part II

Chris Rossbach and Calvin Lin

cs380p

Outline

Over the last few and upcoming classes:

Background from many areas

- Architecture
  - Vector processors
  - Hardware multi-threading
- Graphics
  - Graphics pipeline
  - Graphics programming models
- Algorithms
  - parallel architectures → parallel algorithms

Programming GPUs

- CUDA
- Basics: getting something working
- Advanced: making it perform

Acknowledgements:

http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

Outline

Over the last few and upcoming classes:

Background from many areas

- Architecture
  - Vector processors
  - Hardware multi-threading
- Graphics
  - Graphics pipeline
  - Graphics programming models
- Algorithms
  - parallel architectures → parallel algorithms

Programming GPUs

- CUDA

This lecture

Acknowledgements:

http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

Review

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Memory Controller
HBM2
L2 Cache
GAMERS NEXUS
High-Speed Hub
NVLink

Review

Each SM has multiple vector units (4)

32 lanes wide → warp size

Review

Each SM has multiple vector units (4)

32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Review

Each SM has multiple vector units (4)

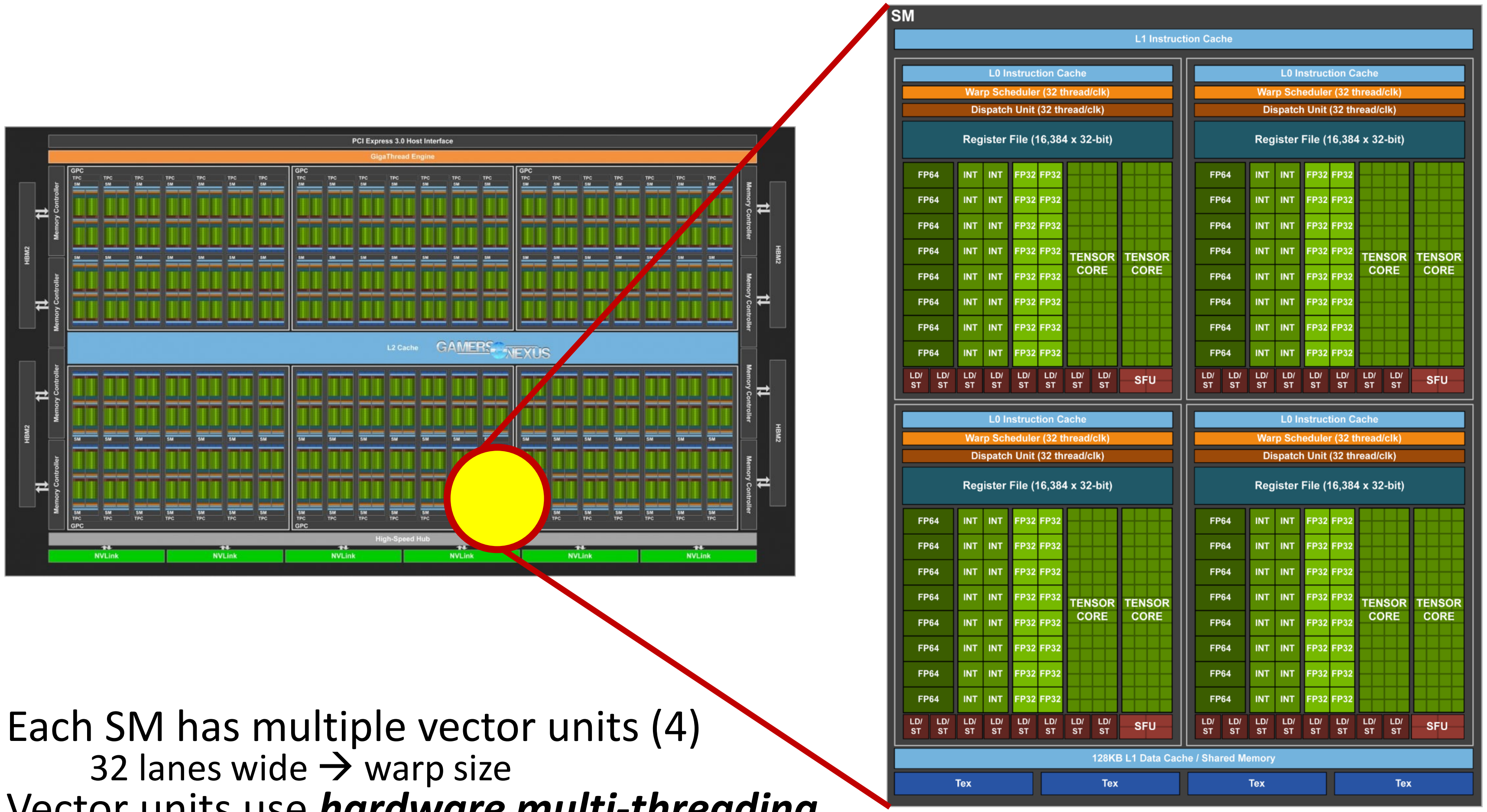
32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

Each TB has some number of threads

Review



Each SM has multiple vector units (4)

32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

Each TB has some number of threads

Review

Thread block scheduler

Each SM has multiple vector units (4)

- 32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

- Each TB has some number of threads

Review

Thread block scheduler

warp (thread) scheduler

Each SM has multiple vector units (4)

32 lanes wide → warp size

Vector units use ***hardware multi-threading***

Execution → a grid of thread blocks (TBs)

Each TB has some number of threads

Review

Thread block scheduler

warp (thread) scheduler

Each SM has multipl

32 lanes wide → w

Vector units use ***har***

Execution → a grid o

Each TB has some num

1000s of HW-scheduled threads per kernel

Threads grouped into independent blocks.

- Threads in a block can synchronize (barrier)
- This is the *only* synchronization

“Grid” == “launch” == “invocation” of a kernel

- a group of blocks (or warps)

Review

Review

Heterogeneous Computing → Host (CPU) offloads to GPU

Review

Heterogeneous Computing → Host (CPU) offloads to GPU
BSP-like Programming Model → Host Serial, GPU parallel

Review

Heterogeneous Computing → Host (CPU) offloads to GPU
BSP-like Programming Model → Host Serial, GPU parallel
Host and Device: 1 program → separate binaries

Review

Heterogeneous Computing → Host (CPU) offloads to GPU
BSP-like Programming Model → Host Serial, GPU parallel
Host and Device: 1 program → separate binaries

Launching parallel kernels

- "Grid" == "launch" == "invocation" of a kernel == a group of blocks (or warps)
- Launch **N** copies of **add()** with **add<<<N/M,M>>>(…);**
- Use **blockIdx.x** to access block index
- Use **threadIdx.x** to access thread index within block

Review

Heterogeneous Computing → Host (CPU) offloads to GPU
BSP-like Programming Model → Host Serial, GPU parallel
Host and Device: 1 program → separate binaries

Launching parallel kernels

- "Grid" == "launch" == "invocation" of a kernel == a group of blocks (or warps)
- Launch **`N`** copies of **`add()`** with **`add<<<N/M,M>>>(…);`**
- Use **`blockIdx.x`** to access block index
- Use **`threadIdx.x`** to access thread index within block

Allocate elements to threads:

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

Review: Why Bother with Threads?

Review: Why Bother with Threads?

```
add<<< 1, N >>>();
add<<< N, 1 >>>();
```

Review: Why Bother with Threads?

```
add<<< 1, N >>>();
add<<< N, 1 >>>();
```

Threads may seem unnecessary

- They add a level of complexity
- Why are there both blocks and threads in the model?

Review: Why Bother with Threads?

```
add<<< 1, N >>>();
add<<< N, 1 >>>();
```

Threads may seem unnecessary

- They add a level of complexity
- Why are there both blocks and threads in the model?

Unlike parallel blocks, threads have mechanisms to:

- Communicate
- Synchronize

Review: Why Bother with Threads?

```
add<<< 1, N >>>();
add<<< N, 1 >>>();
```

Threads may seem unnecessary
- They add a level of complexity
- Why are there both blocks and threads in the model?

Unlike parallel blocks, threads have mechanisms to:
- Communicate
- Synchronize

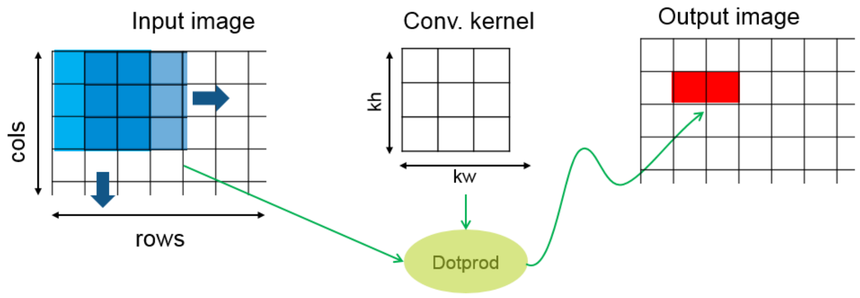
To understand how/why, we need a new example…

Stencils

Stencils

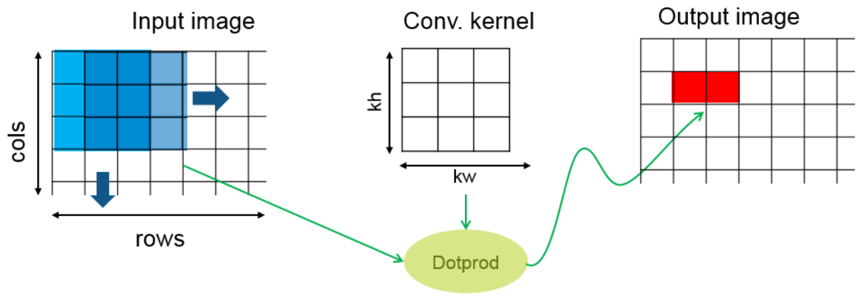
Each pixel → function of neighbors

Stencils



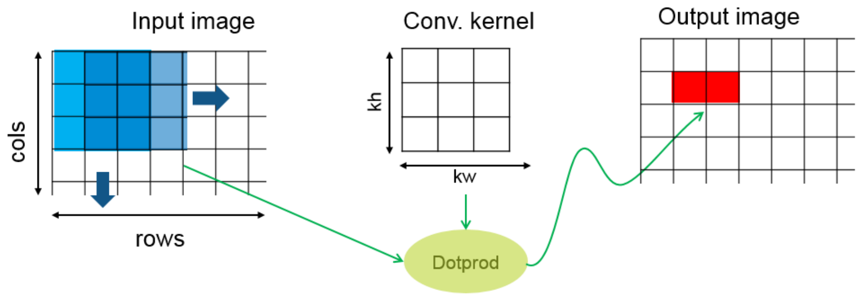
Each pixel → function of neighbors

Stencils



Each pixel → function of neighbors
Edge detection:

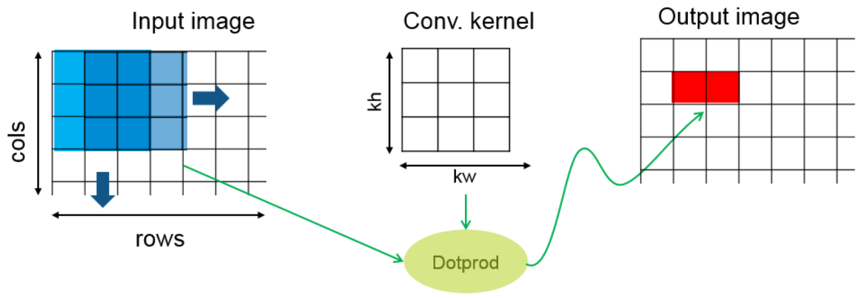
Stencils



Each pixel → function of neighbors
Edge detection:

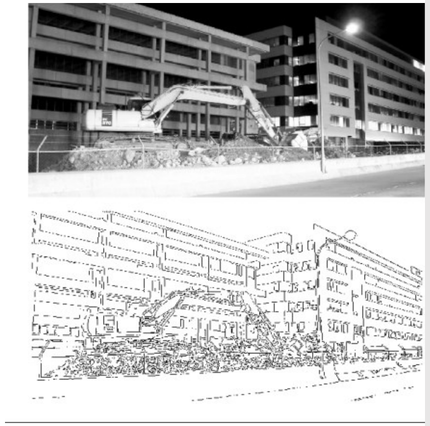
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Stencils

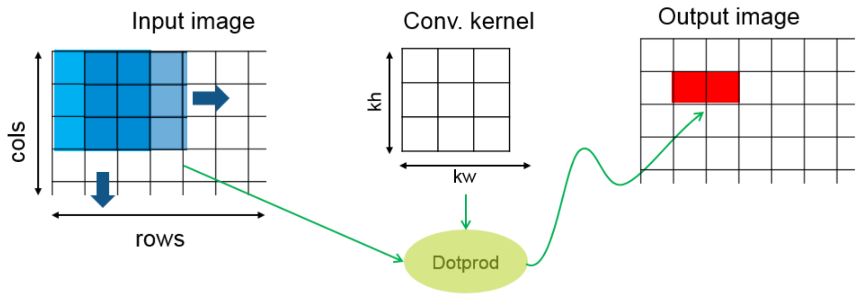


Each pixel → function of neighbors
Edge detection:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$



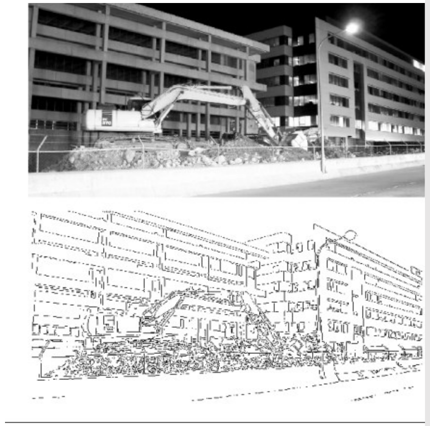
Stencils



Each pixel → function of neighbors

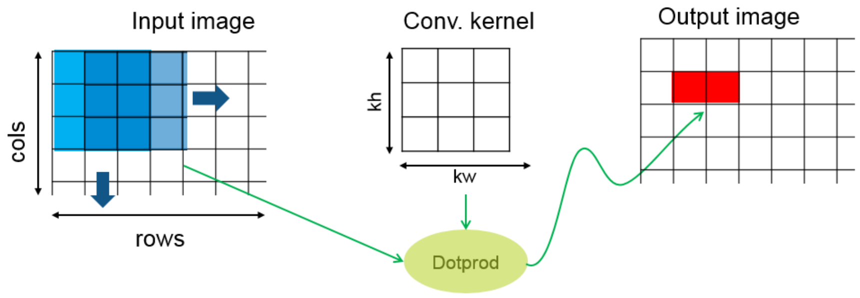
Edge detection:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$



Blur:

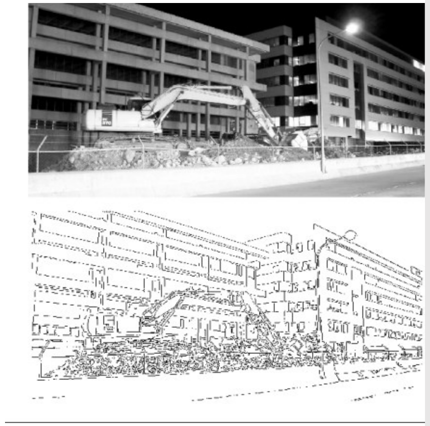
Stencils



Each pixel → function of neighbors

Edge detection:

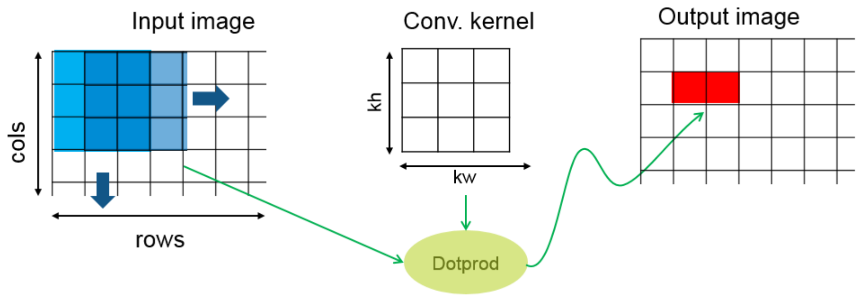
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$



Blur:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

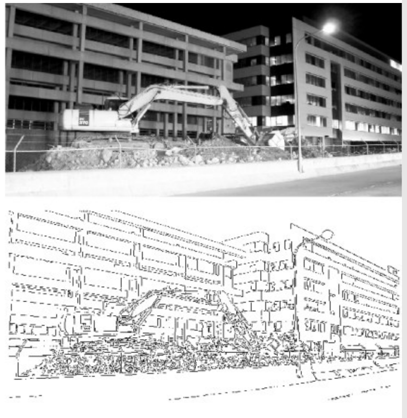
Stencils



Each pixel → function of neighbors

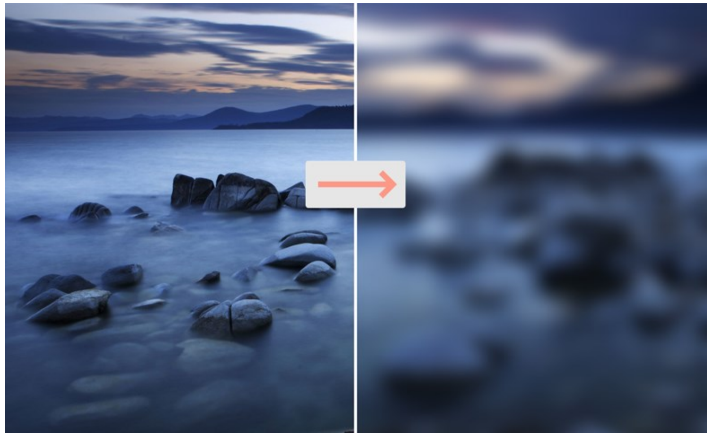
Edge detection:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$



Blur:

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16



1D Stencil

Consider 1D stencil over 1D array of elements

- Each output element is the sum of input elements within a radius

1D Stencil

Consider 1D stencil over 1D array of elements

- Each output element is the sum of input elements within a radius

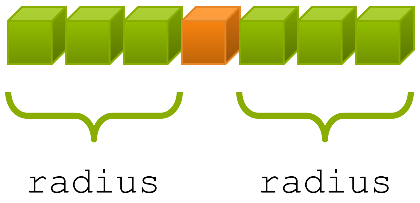
Radius == 3 → each output element is sum of 7 input elements:

1D Stencil

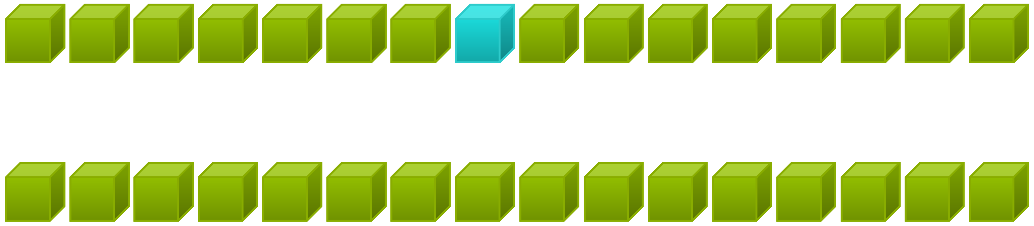
Consider 1D stencil over 1D array of elements

- Each output element is the sum of input elements within a radius

Radius == 3 → each output element is sum of 7 input elements:



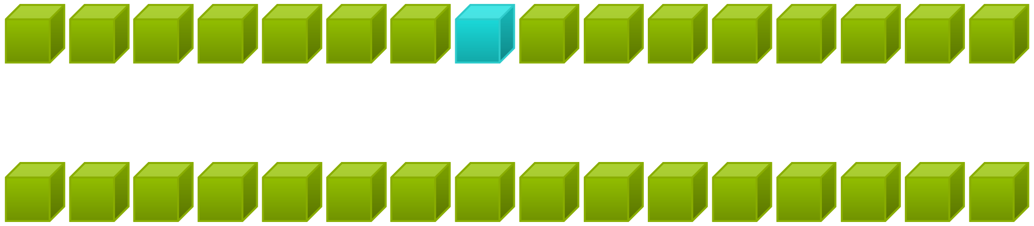
Implementation within a block



Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

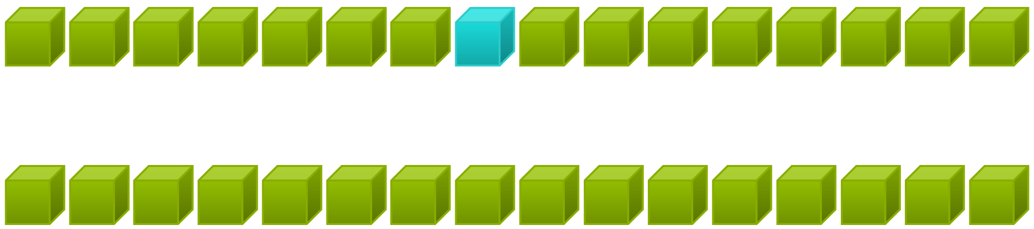


Implementation within a block

Each thread: 1 output element

- blockDim.x elements per block

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



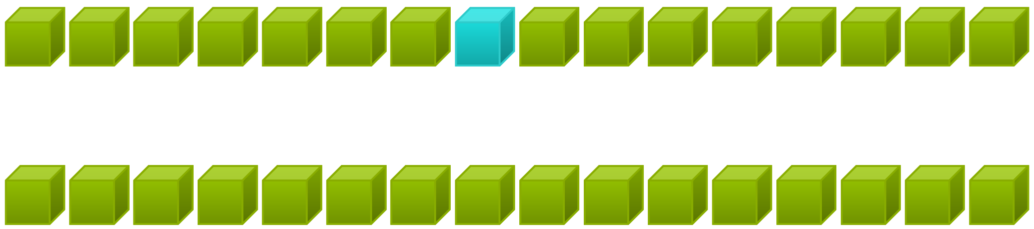
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



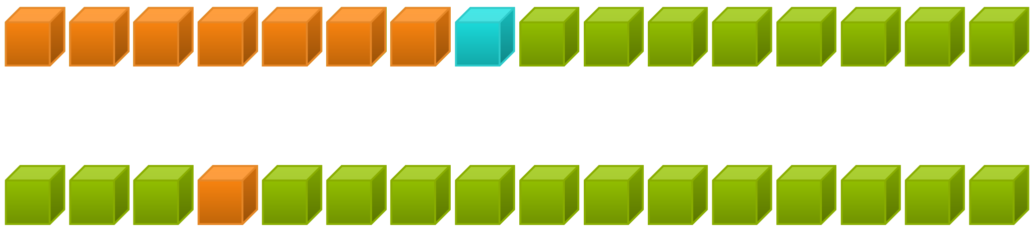
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



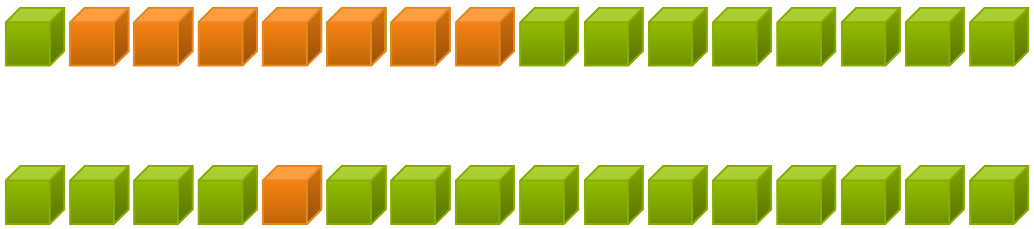
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



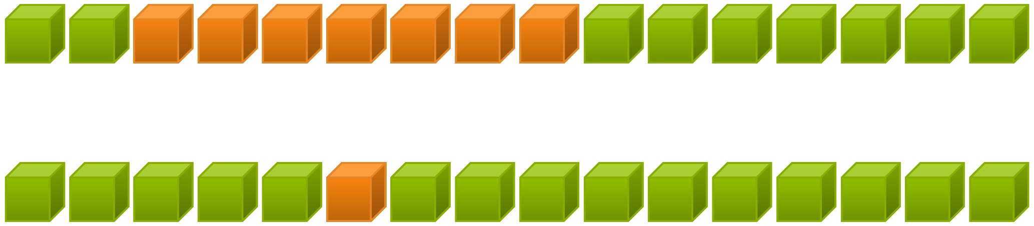
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



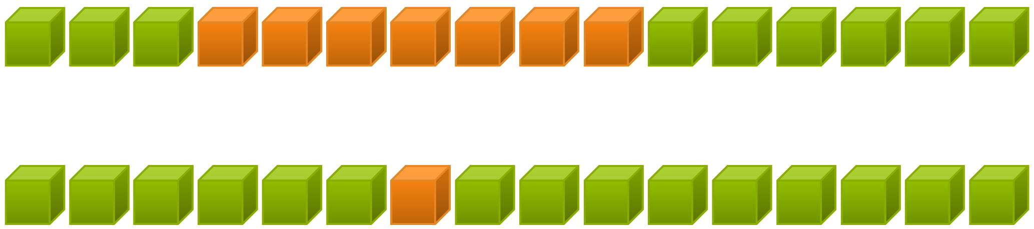
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



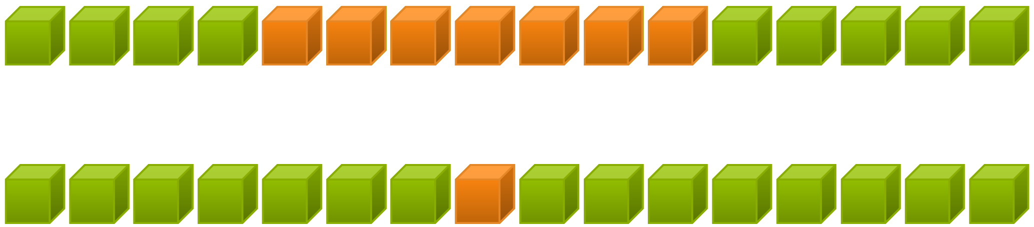
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



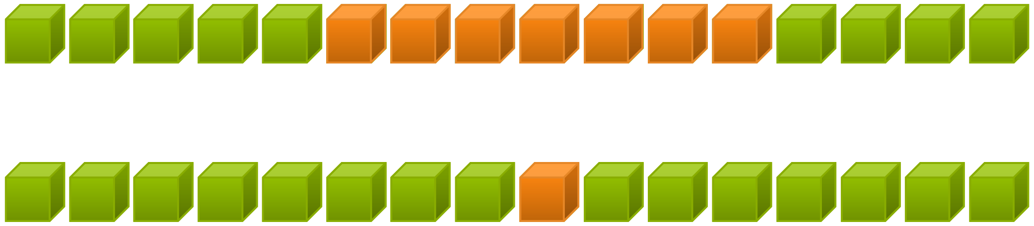
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



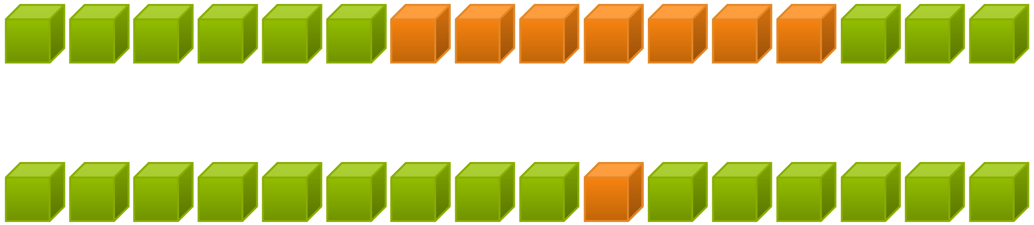
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



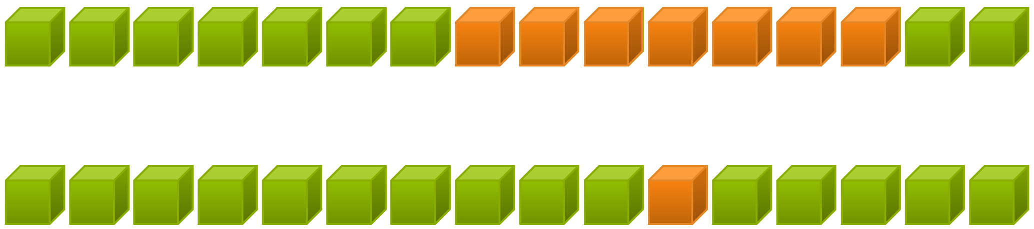
Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



Implementation within a block

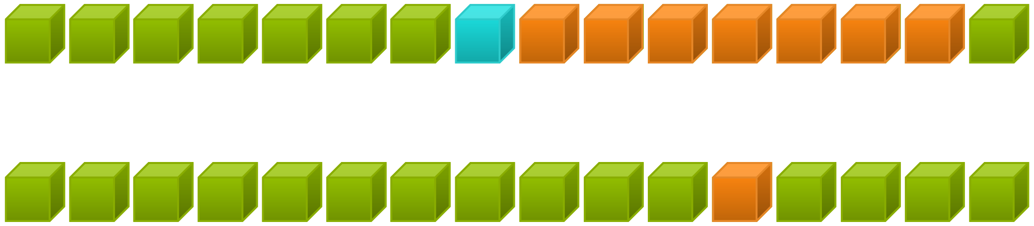
Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

Radius 3 → each elem read 7X!

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```



Implementation within a block

Implementation within a block

Each thread: 1 output element

- blockDim.x elements per block

Implementation within a block

Each thread: 1 output element

- blockDim.x elements per block

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```

Implementation within a block

Each thread: 1 output element

blockDim.x elements per block

Input elems read many times

Radius 3 → each elem is read 7X!

```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

  // Store the result
  out[idx] = result;
}
```

Sharing Data Between Threads

Sharing Data Between Threads

Terminology: within a block, threads share via ***shared memory***

Sharing Data Between Threads

Terminology: within a block, threads share via ***shared memory***

Extremely fast on-chip memory, user-managed

Sharing Data Between Threads

Terminology: within a block, threads share via ***shared memory***

Extremely fast on-chip memory, user-managed

Declare using `__shared__`, allocated per block

Sharing Data Between Threads

Terminology: within a block, threads share via ***shared memory***

Extremely fast on-chip memory, user-managed

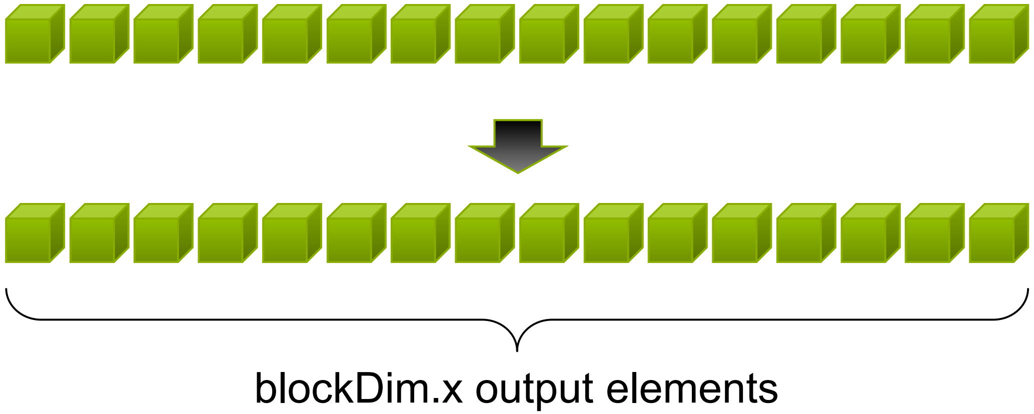
Declare using `__shared__`, allocated per block

Data is *not visible* to threads in other blocks

Stencil with Shared Memory

Cache data in shared memory

- –Read (blockDim.x + 2 * radius) elements from memory to shared
- –Compute blockDim.x output elements
- –Write blockDim.x output elements to global memory

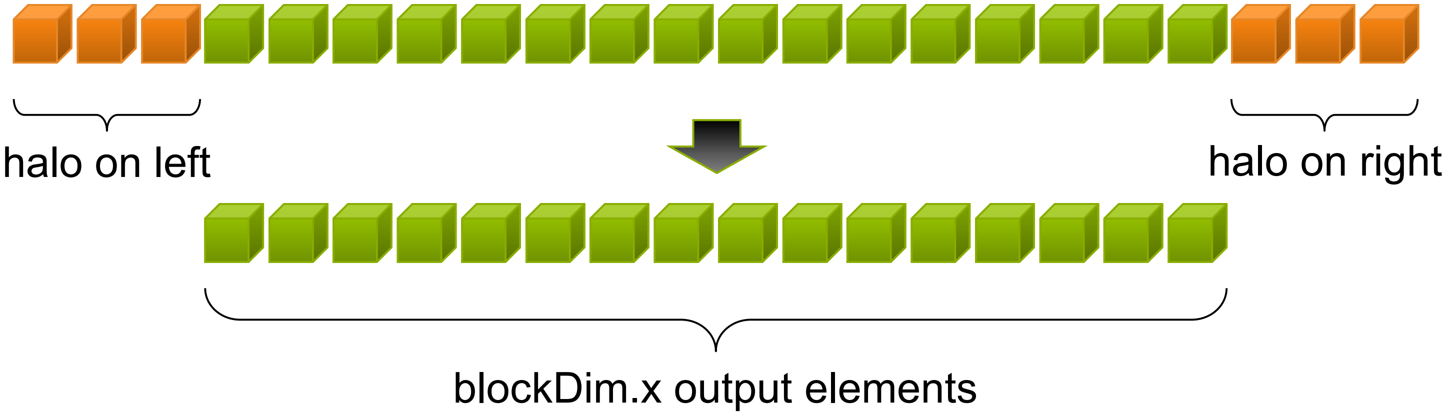


Stencil with Shared Memory

Cache data in shared memory

- –Read (blockDim.x + 2 * radius) elements from memory to shared
- –Compute blockDim.x output elements
- –Write blockDim.x output elements to global memory

- –Each block needs a halo of radius elements at each boundary

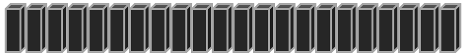


Stencil Kernel

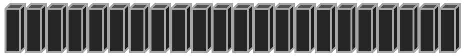
```
__global__ void stencil_1d(int *in, int *out) {
```

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```



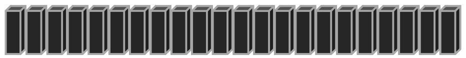
Stencil Kernel



```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
```

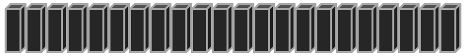
Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
```



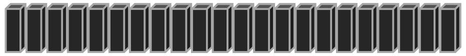
Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
```



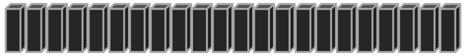
Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
```



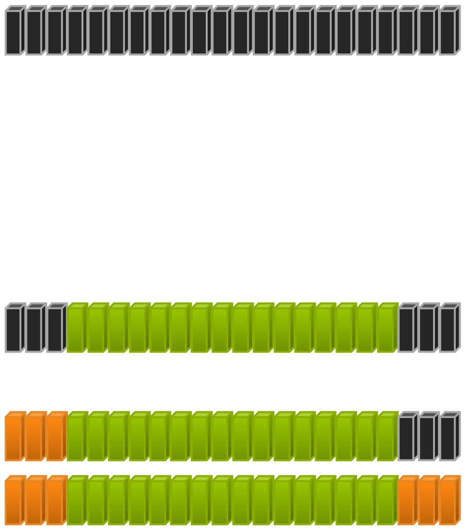
Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
```



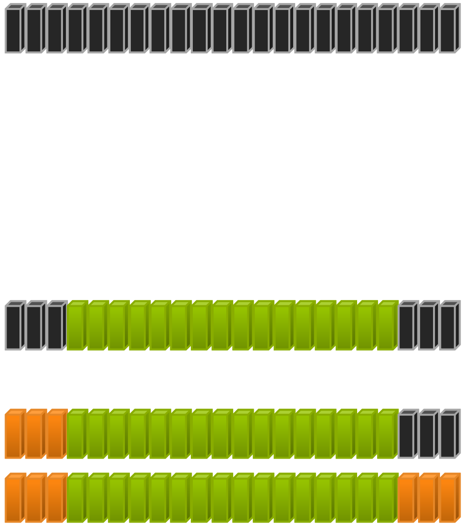
Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



Are we done?

Data Race!

- The stencil example will not work…

Data Race!

- The stencil example will not work…

- Suppose thread 15 reads halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Data Race!

- The stencil example will not work…

- Suppose thread 15 reads halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18] 

Skipped, threadIdx > RADIUS

Load from temp[19] 

__syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

- Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier

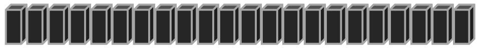
- In conditional code, the condition must be uniform across the block

Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
```

Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
```

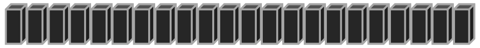


Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];        [redacted]
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
```

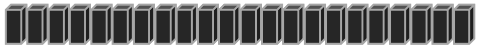
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
```



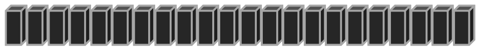
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
```



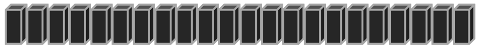
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
```



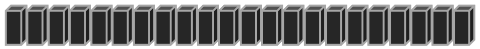
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
```



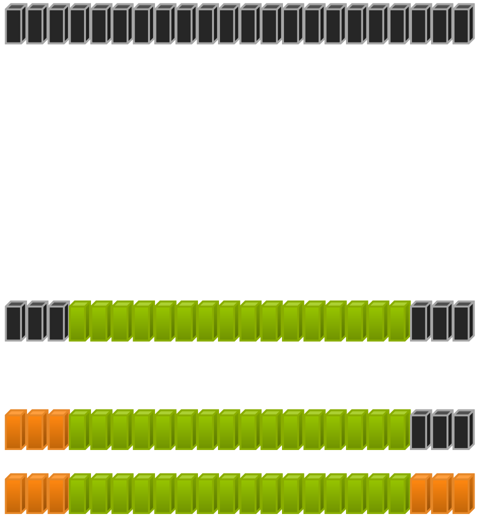
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



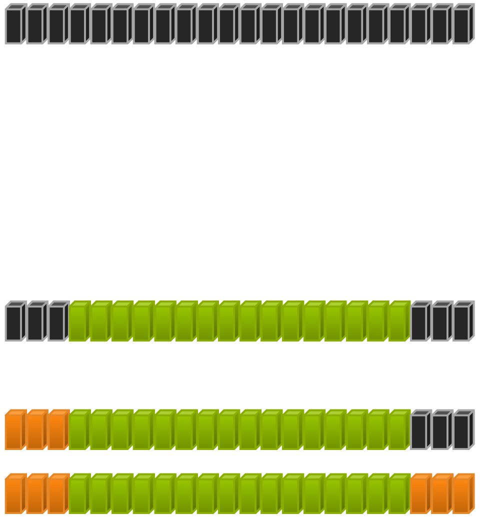
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
```



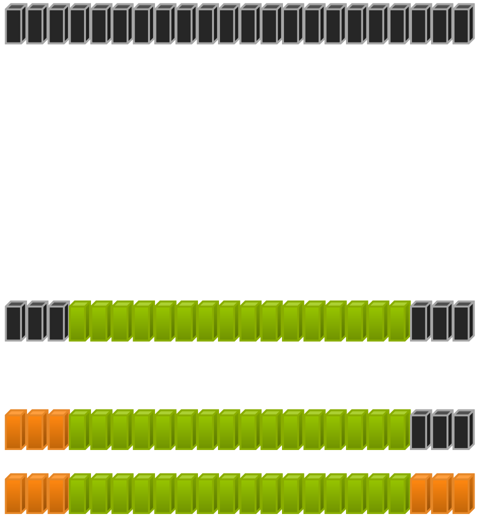
Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
 // Apply the stencil
 int result = 0;
 for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
   result += temp[lindex + offset];

 // Store the result
 out[gindex] = result; }
```



Correct Stencil Kernel

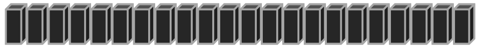
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RA
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result; }
```



Why doesn't L1 provide same benefit?

Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RA
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result; }
```

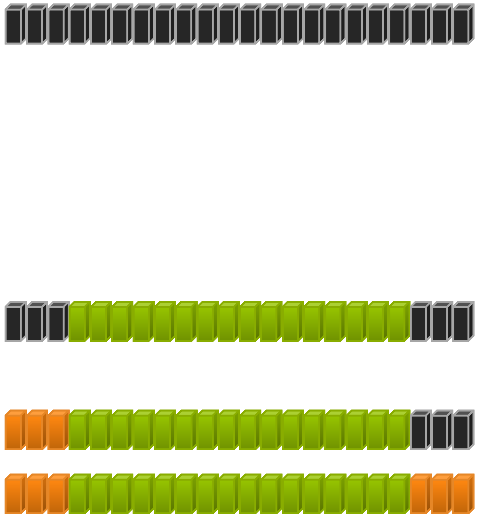


Why doesn't L1 provide same benefit?

- manual control avoids eviction

Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RA
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result; }
```

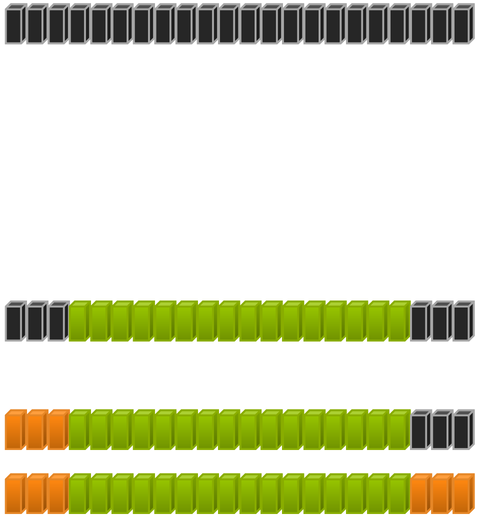


Why doesn't L1 provide same benefit?

- manual control avoids eviction
- write-through overheads avoided

Correct Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RA
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result; }
```

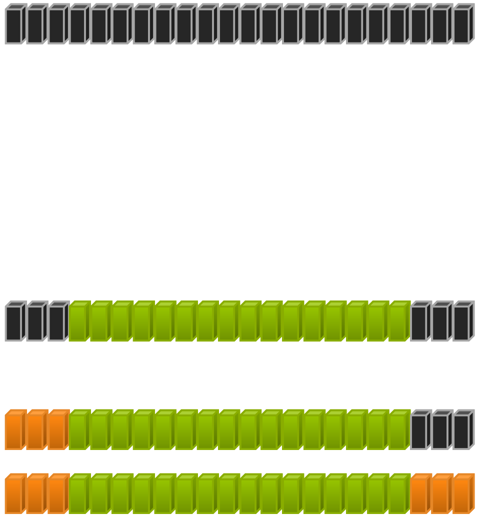


Why doesn't L1 provide same benefit?

- manual control avoids eviction
- write-through overheads avoided
- no write-back of dead values

Correct Stencil Kernel

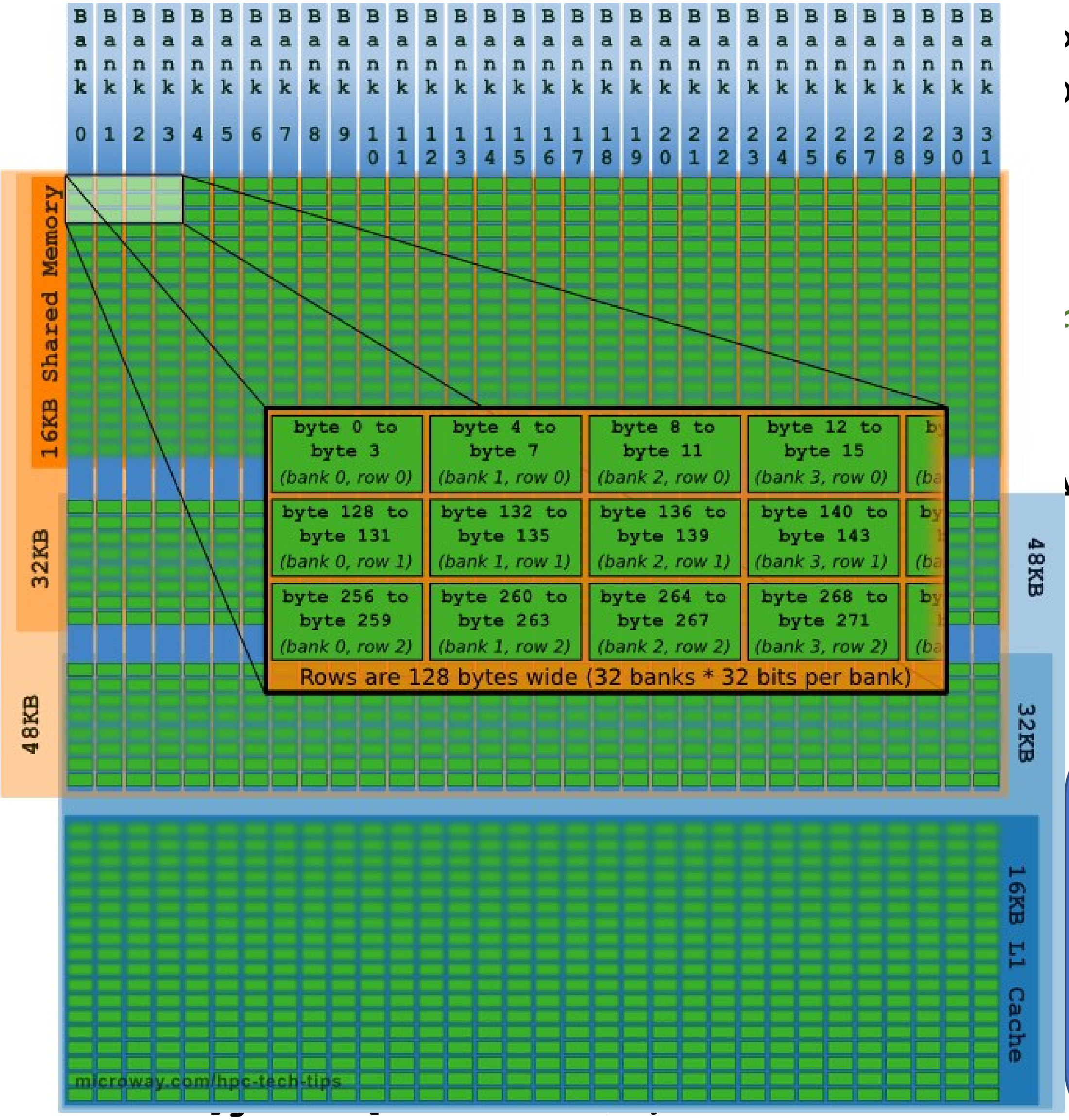
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
  __syncthreads();
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RA
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result; }
```



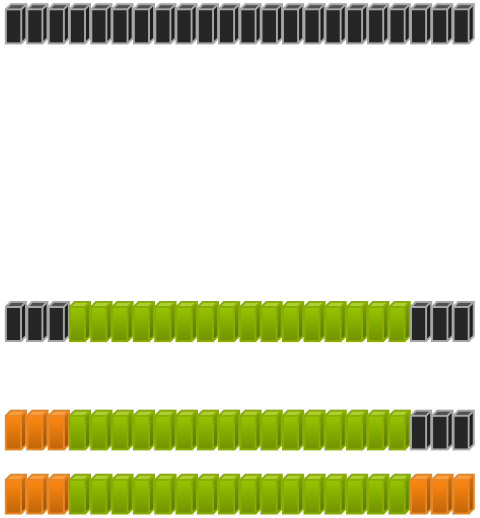
Why doesn't L1 provide same benefit?

- manual control avoids eviction
- write-through overheads avoided
- no write-back of dead values
- provides an opportunity to make mis-aligned / bank conflicting accesses aligned/non-conflicting

Correct Stencil Kernel




```
ut) {
IUS];
 blockDim.x;

y

ADIUS];
```



Why doesn't L1 provide same benefit?

- manual control avoids eviction
- write-through overheads avoided
- no write-back of dead values
- provides an opportunity to make mis-aligned / bank conflicting accesses aligned/non-conflicting

Notes on __syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

- Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier

- In conditional code, the condition must be uniform across the block

Notes on __syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

–Used to prevent RAW / WAR / WAW hazards

```
__global__ void some_kernel(int *in, int *out) {
  // good idea?
  if(threadIdx.x == SOME_VALUE)
     __syncthreads();
}
```

All threads must reach the barrier

–In conditional code, the condition must be uniform across the block

Notes on __syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

- Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier

- In conditional code, the condition must be uniform across the block

Notes on __syncthreads()

```
void __syncthreads();
```

Synchronizes all threads within a block

- Used to prevent RAW / WAR / WAW hazards

All threads must reach the barrier

- In conditional code, the condition must be uniform across the block

```
__device__ void lock_trick(int *in, int *out) {
  __syncthreads();
  if(myIndex == 0)
    critical_section();
  __syncthreads();
}
```

GPU Atomics

GPU Atomics

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
atomicAdd()        atomicInc()
atomicSub()        atomicDec()
atomicMin()        atomicExch()
atomicMax()        atomicCAS()
```

Implemented as write-through to L2

- “Fire-and-forget”

GPU Atomics

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
atomicAdd()        atomicInc()
atomicSub()        atomicDec()
atomicMin()        atomicExch()
atomicMax()        atomicCAS()
```

Implemented as write-through to L2

- "Fire-and-forget"

GPU Atomics

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;   // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Is this a good idea?

Read-Modify-Write – atomic

```
atomicAdd()          atomicInc()
atomicSub()          atomicDec()
atomicMin()          atomicExch()
atomicMax()          atomicCAS()
```

Implemented as write-through to L2

- "Fire-and-forget"

GPU Atomics

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

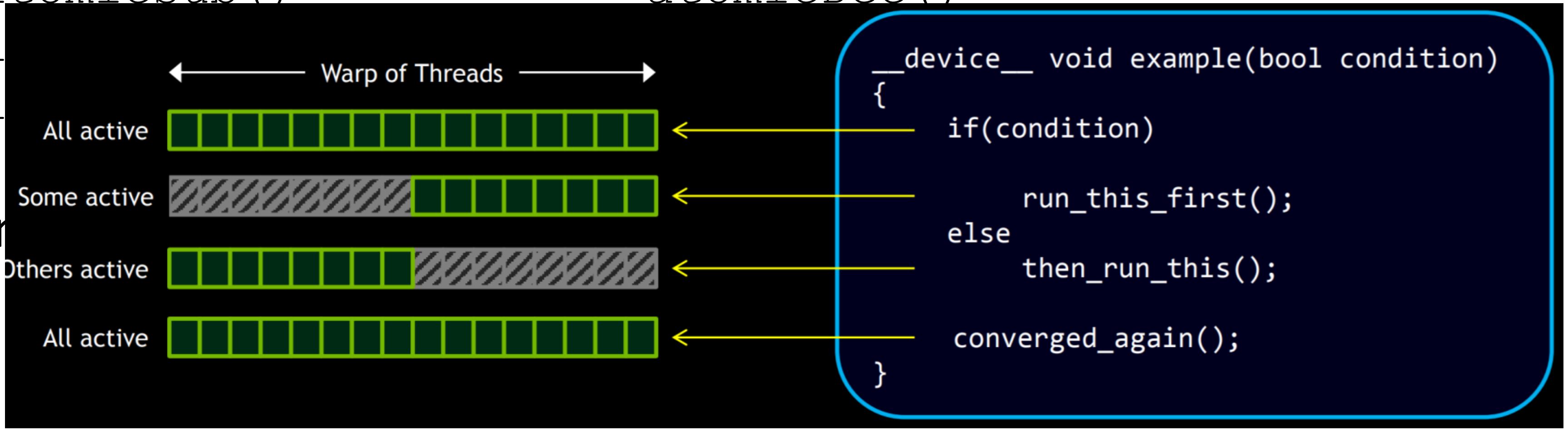
Is this a good idea?

Read-Modify-Write – atomic

```
atomicAdd()        atomicInc()
atomicSub()        atomicDec()
a
a
```

Ir

•



GPU Atomics

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Is this a good idea?

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

```
atomicAdd()          atomicInc()
atomicSub()          atomicDec()
atomicMin()          atomicExch()
atomicMax()          atomicCAS()
```

Implemented as write-through to L2

- "Fire-and-forget"

GPU Atomics

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

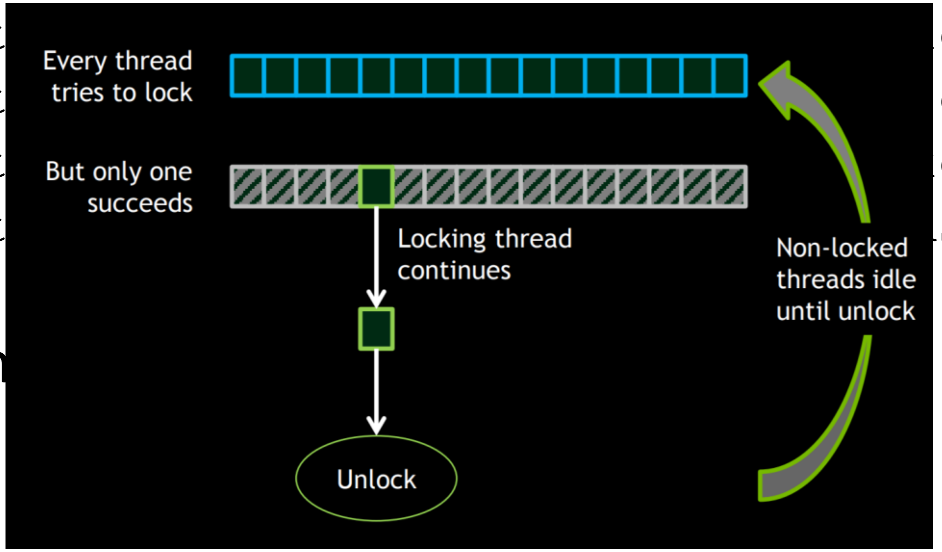
Is this a good idea?

Race conditions –

- Traditional locks: avoid!
- How do we synchronize?

Read-Modify-Write – atomic

at c()
at c()
at ch()
at S()



Im

•

GPU Atomics

Race conditions –

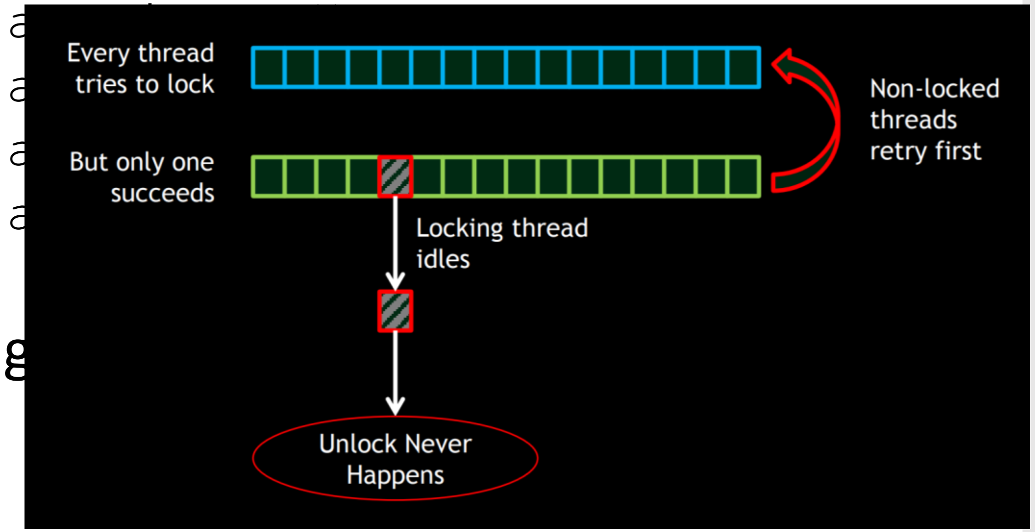
- Traditional locks: avoid!
- How do we synchronize?

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;   // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Is this a good idea?

Read-Modify-Write – atomic

```
atomicAdd()          a
atomicSub()          a
atomicMin()          a
atomicMax()          a
```



Implemented as write-throug

- "Fire-and-forget"

Coordinating Host & Device

Kernel launches are asynchronous

- Control returns to the CPU immediately

CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

Reporting Errors

All CUDA API calls return an error code (**`cudaError_t`**)

- Error in the API call itself
  - OR
- Error in an earlier asynchronous operation (e.g. kernel)

Reporting Errors

All CUDA API calls return an error code (`cudaError_t`)

- Error in the API call itself
  - OR
- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
```

Reporting Errors

All CUDA API calls return an error code (`cudaError_t`)

- Error in the API call itself
  - OR
- Error in an earlier asynchronous operation (e.g. kernel)

Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

Get a string to describe the error:

```
char *cudaGetErrorString(cudaError_t)
printf("%s\n", cudaGetErrorString(cudaGetLastError()));
```

Device Management

Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop,
                        int device)
```

Multiple threads can share a device

A single thread can manage multiple devices

`cudaSetDevice(i)` to select current device

`cudaMemcpy(…)` for peer-to-peer copies †

CUDA Events: Measuring Performance

```
float memsettime;
cudaEvent_t start, stop;

// initialize CUDA timer
cudaEventCreate(&start);  cudaEventCreate(&stop);
cudaEventRecord(start,0);

// CUDA Kernel
 . . .

// stop CUDA timer
cudaEventRecord(stop,0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&memsettime,start,stop);
printf(" *** CUDA execution time: %f *** \n", memsettime);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Summary

- Launching parallel threads
  - Launch `N` blocks with `M` threads per block with `kernel<<<N,M>>>(…);`
  - Use `blockIdx.x` to access block index within grid
  - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

- Use `__shared__` to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use `__syncthreads()` as a barrier
  - Use to prevent data hazards
- Device Management APIs
- Instrumentation APIs