CUDA

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
  - Exam grading in progress
- Agenda
  - CUDA p2 + GPU optimization

Acknowledgements:

- http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx
- http://www.seas.upenn.edu/~cis565/LECTURES/CUDA%20Tricks.pptx
- http://ece757.ece.wisc.edu/lect13-gpgpu.pptx
- http://www.cs.utexas.edu/~pingali/CS378/2015sp/lectures/GPU%20Programming.pptx

COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

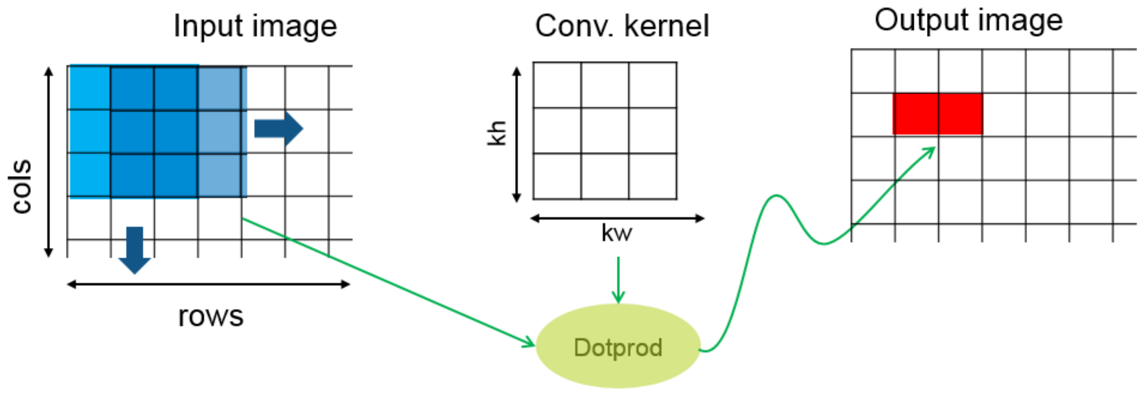
Review: Stencils

Review: Stencils

- Each pixel → function of neighbors

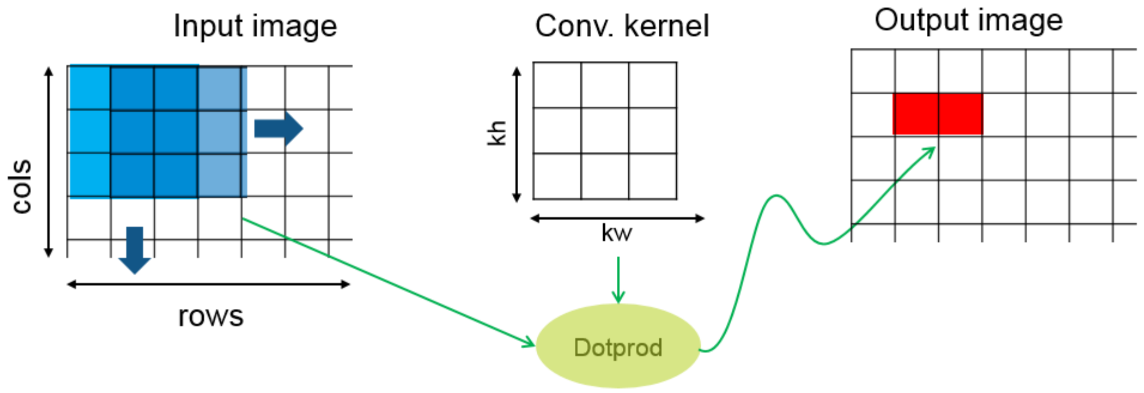
Review: Stencils

- Each pixel → function of neighbors



Review: Stencils

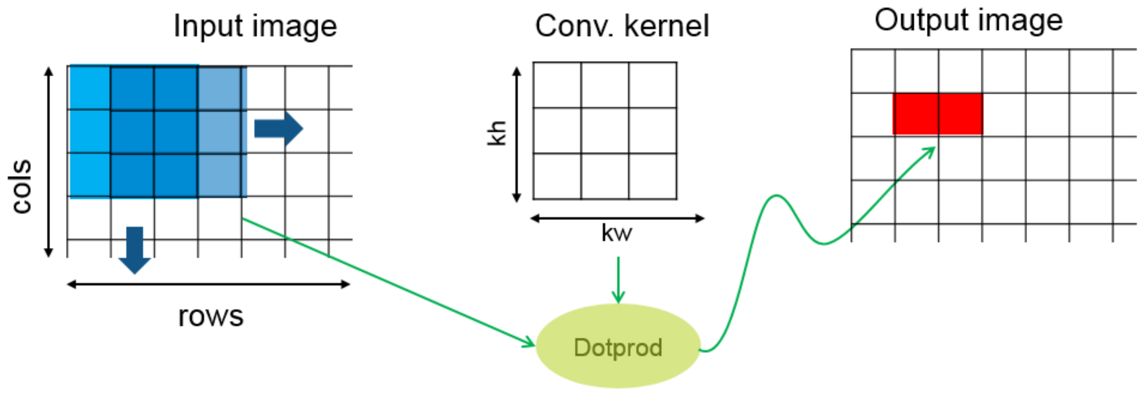
- Each pixel → function of neighbors
- Edge detection:



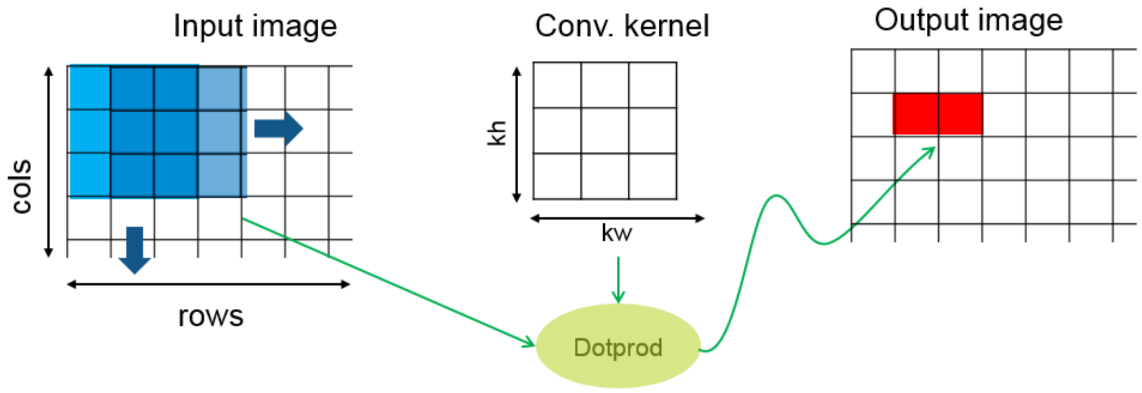
Review: Stencils

- Each pixel → function of neighbors
- Edge detection:

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

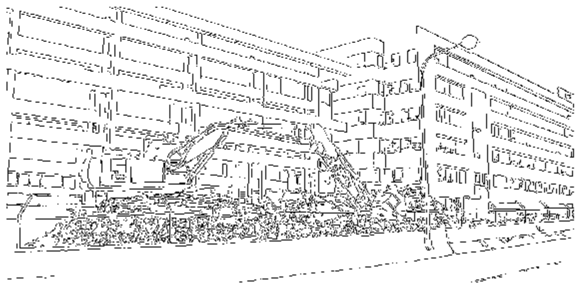


Review: Stencils

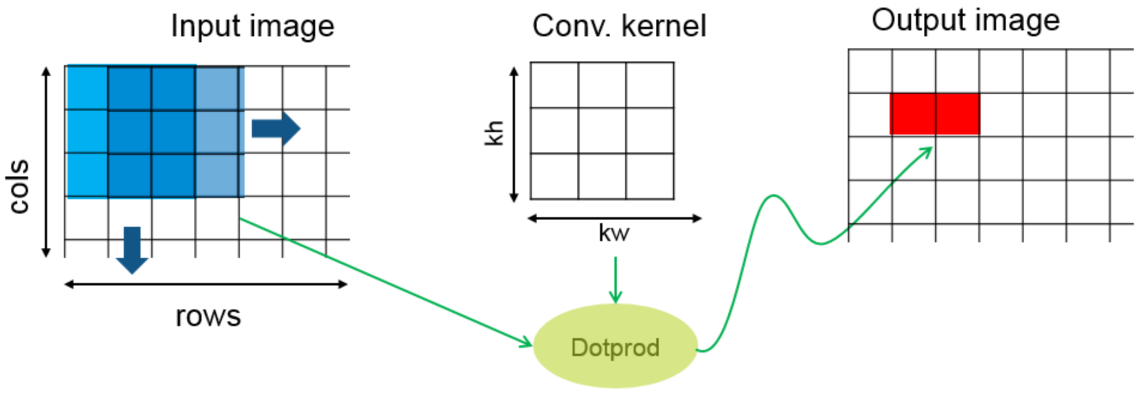


- Each pixel → function of neighbors
- Edge detection:

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

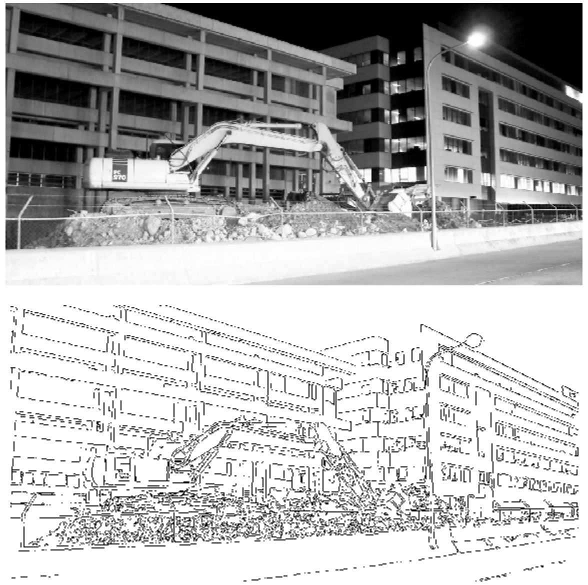


Review: Stencils



- Each pixel → function of neighbors
- Edge detection:

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

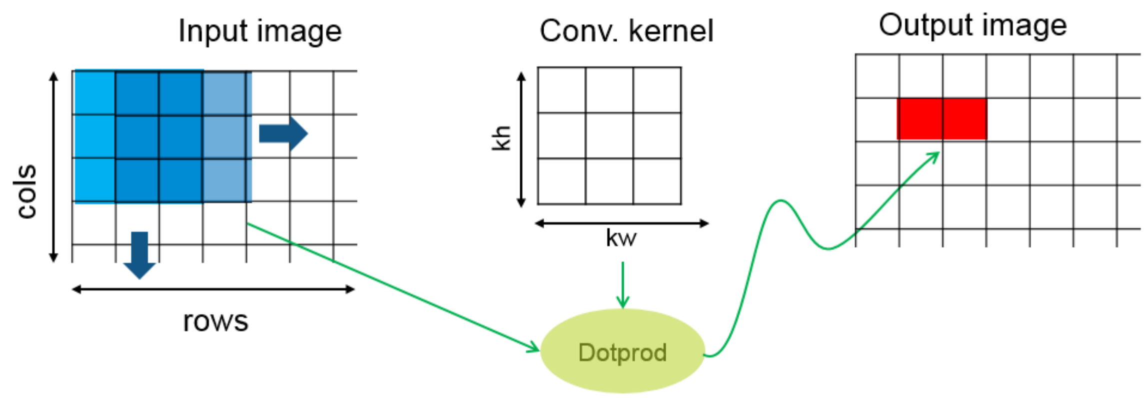


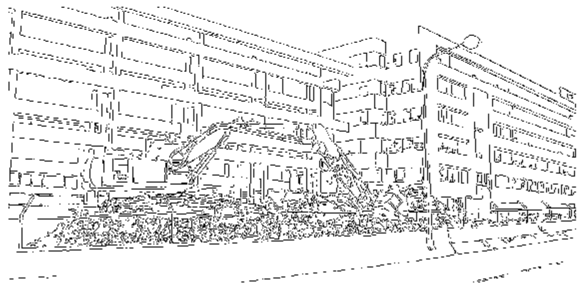
- Blur:

Review: Stencils

- Each pixel → function of neighbors
- Edge detection:

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

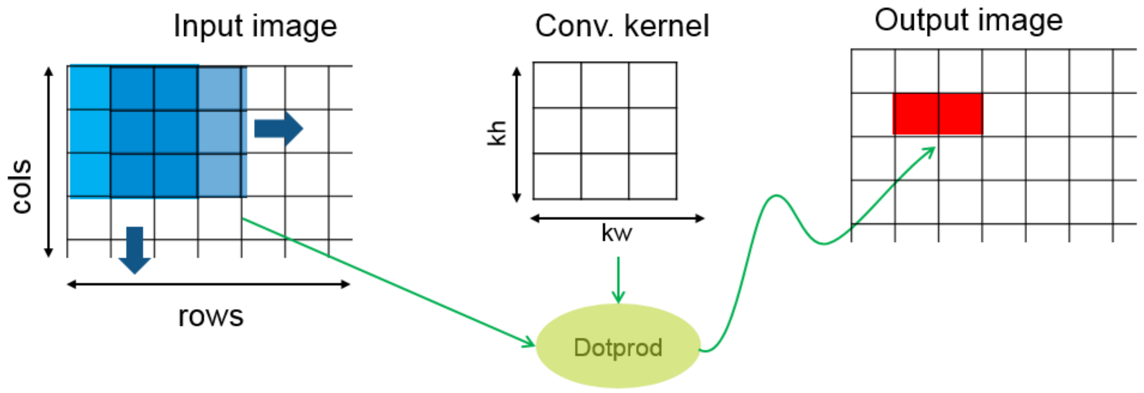




- Blur:

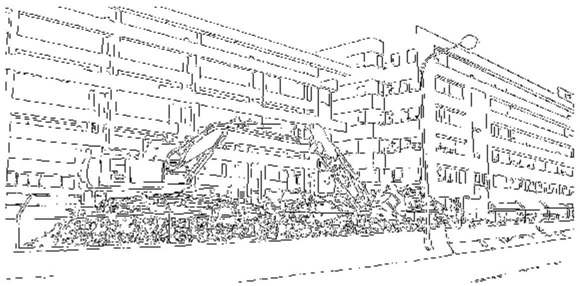
1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Review: Stencils



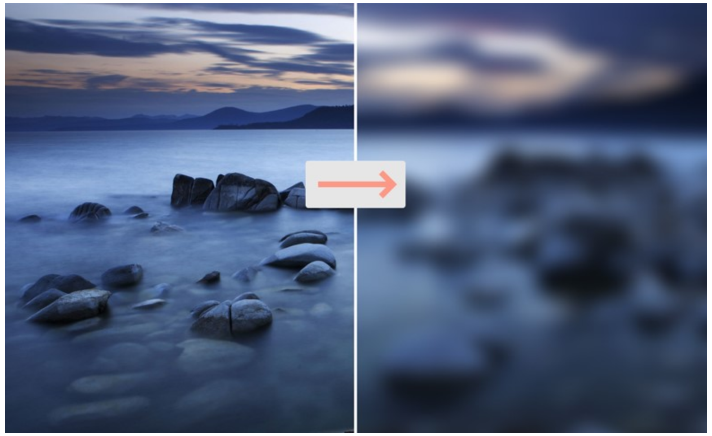
- Each pixel → function of neighbors
- Edge detection:

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



- Blur:

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

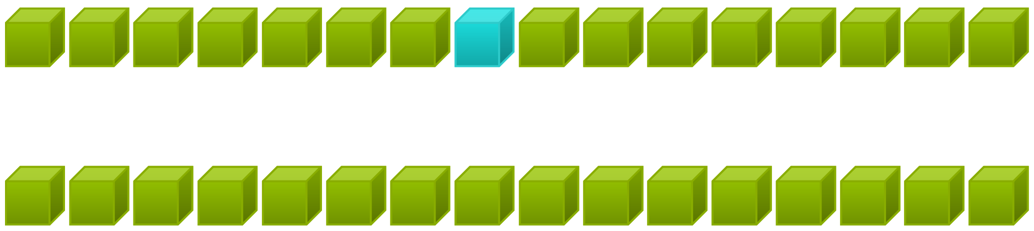


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

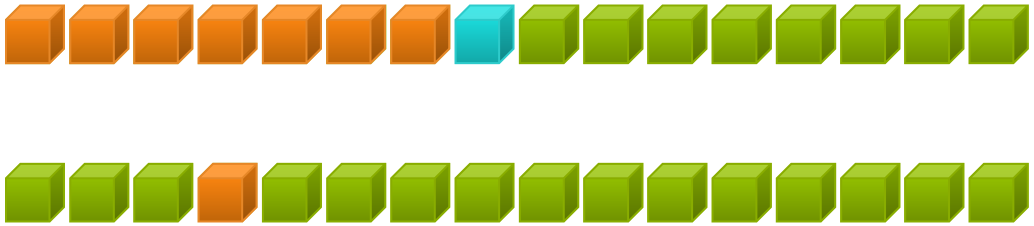


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

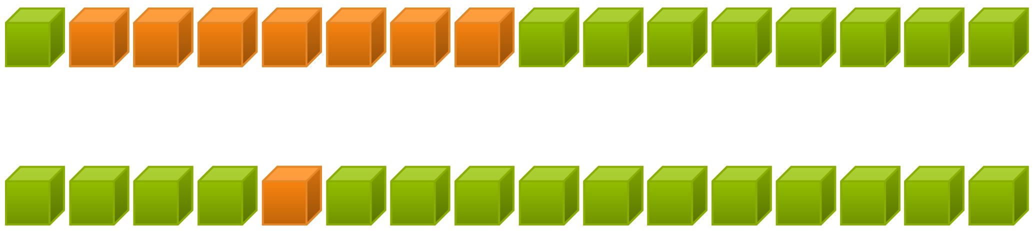


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

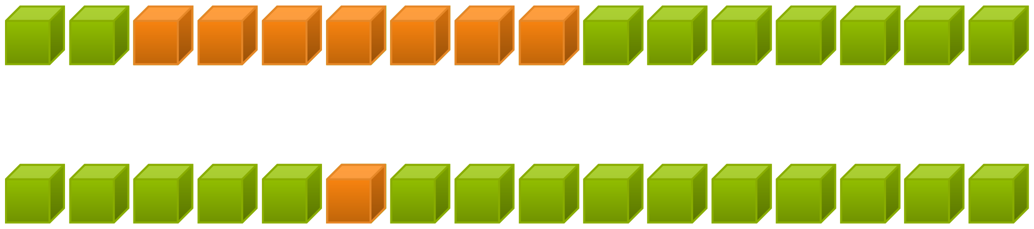


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

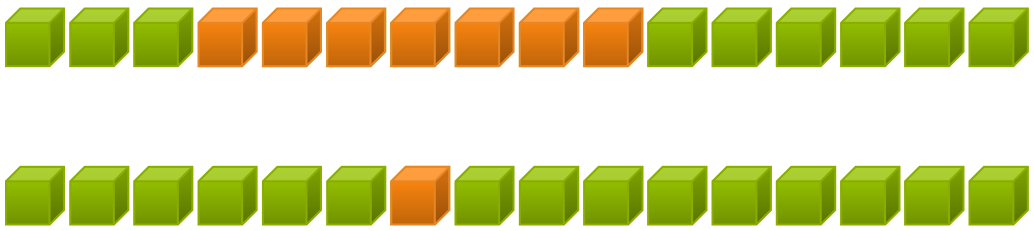


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

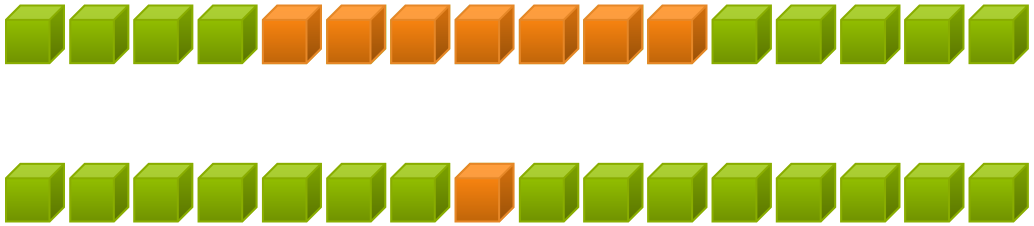


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

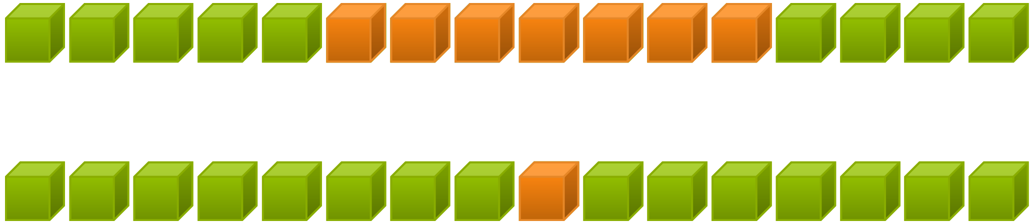


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```



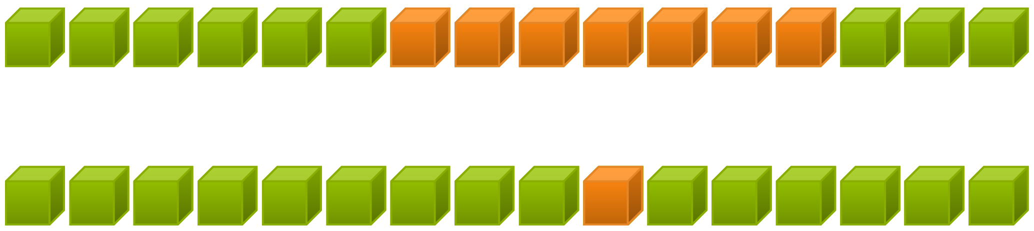
Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block

- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

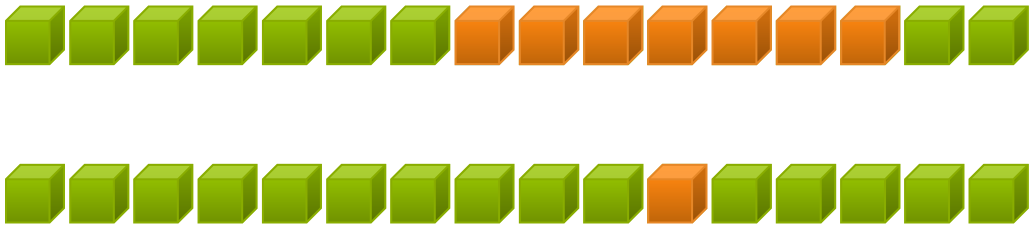


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```

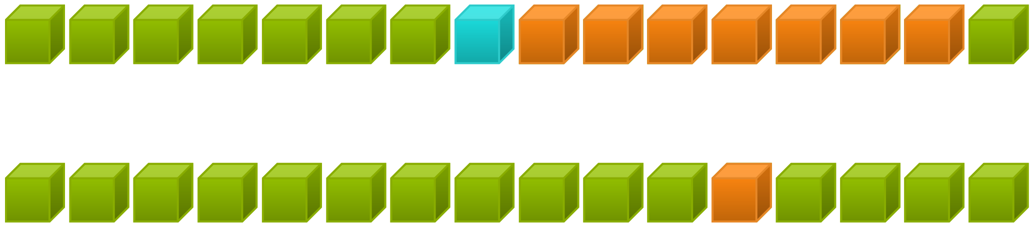


Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```



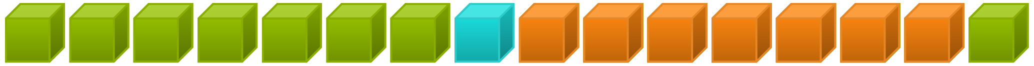
Review: Stencil Implementation within a block

- Each thread: process 1 output element
  - blockDim.x elements per block

- Input elements read many times
  - With radius 3, each input element is read seven 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;
}
```



Solution? Use *Shared Memory*!!!

- **Terminology: within a block, threads share data 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 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

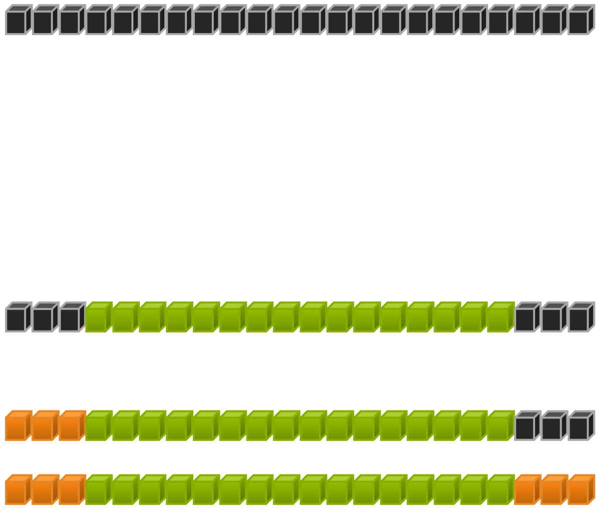
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // 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];
  }
```

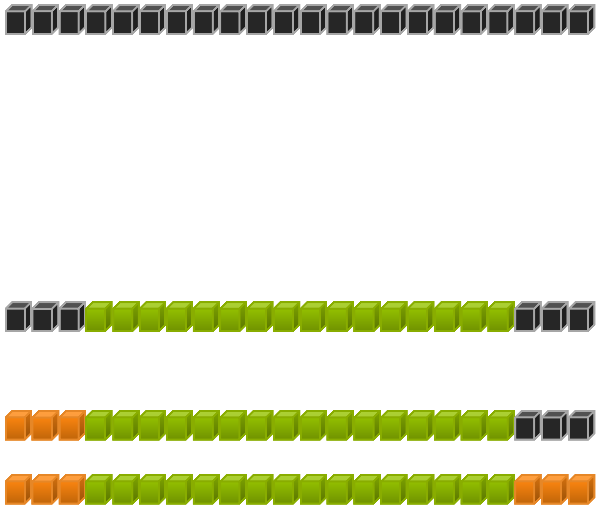


Stencil Kernel

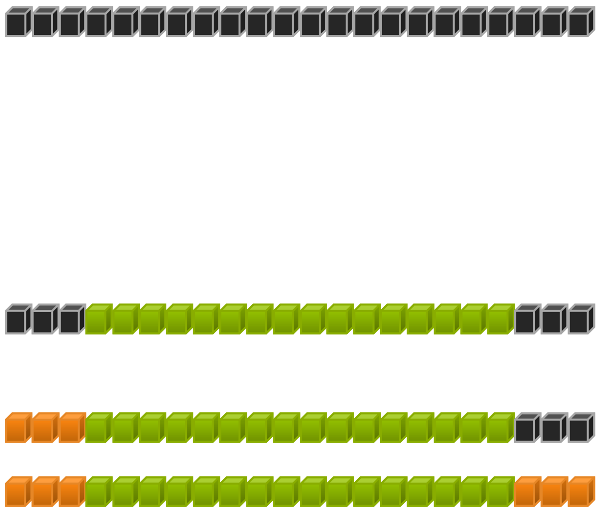
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // 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];
  }
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

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



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];
  }
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

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

Are we done?

Data Race!

- The stencil example will not work…

Data Race!

- The stencil example will not work…
- Suppose thread 15 reads the 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 the 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 the 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 the 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];
  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

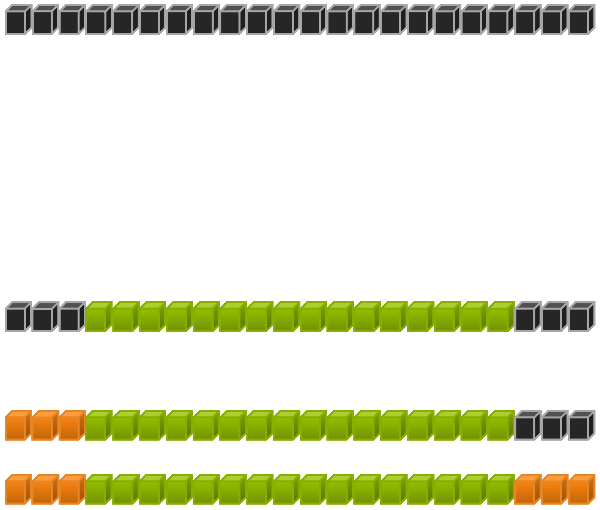
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // 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();
```

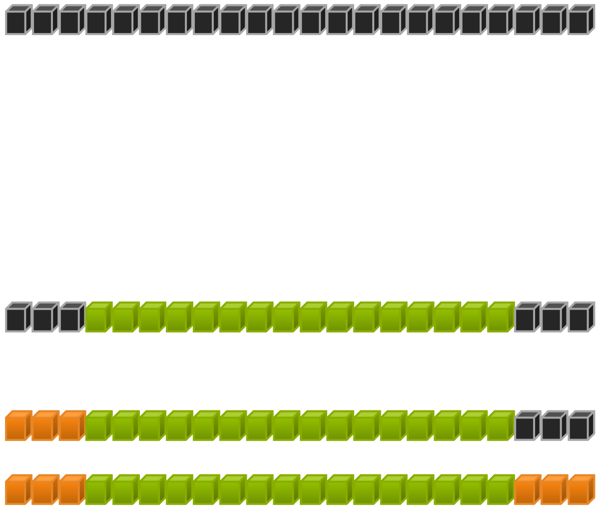


Correct Stencil Kernel

```
__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;
}
```



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 bl
  - Used to prevent RAW / WAR / WAW h

```
__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();
}
```

Atomics

Race conditions –

- Traditional locks are to be avoided
- How do we synchronize?

Read-Modify-Write – atomic

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

Atomics

Race conditions –

- Traditional locks are to be avoided
- How do we synchronize?

Read Modify Write – atomic

atomi
atomi
atomi
atomi

```
__device__ double atomicAdd(double* address, double val) {
    unsigned long long int* address_as_ull = (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull,
                        assumed,
                        __double_as_longlong(val + __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

Recap

- 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

MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

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[†]

[†] requires OS and device support

Device Management

- Appli

dev

- Multi

- A sin

cudaError_t cudaGetDeviceProperties (struct cudaDeviceProp * *prop*,
int *device*
)

Returns in `*prop` the properties of device `dev`. The **cudaDeviceProp** structure is defined as:

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int clockRate;
    size_t totalConstMem;
    int major;
    int minor;
    size_t textureAlignment;
    size_t texturePitchAlignment;
    int deviceOverlap;
    int multiProcessorCount;
```

† requires OS and device support

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[†]

[†] requires OS and device support

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);
```

Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

Compute Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

Compute Capability

- The **compute**
  - Number of
  - Sizes of me
  - Features &

Co
Ca

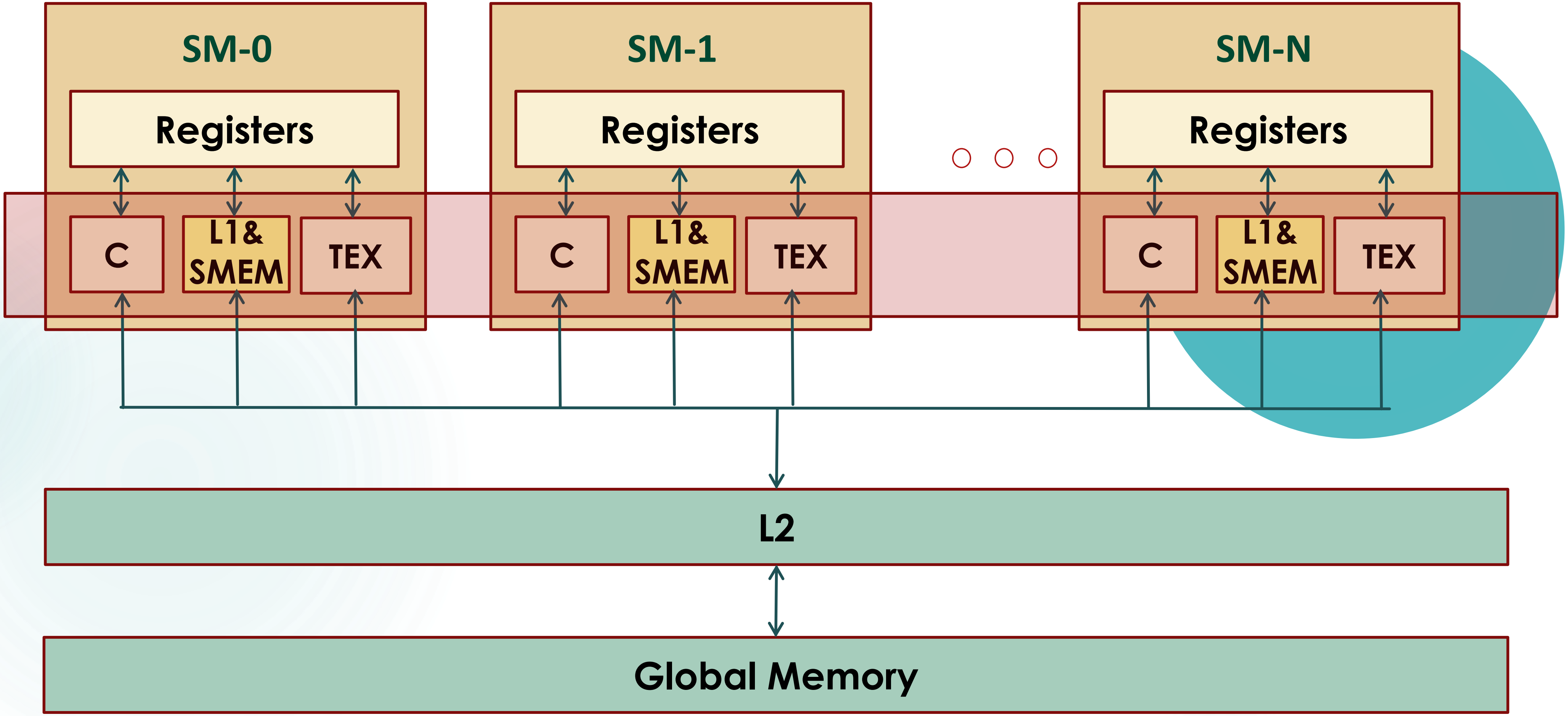
Feature Support	Compute Capability				
(Unlisted features are supported for all compute capabilities)	3.5, 3.7, 5.0, 5.2	5.3	6.x	7.x	8.x
Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)	Yes				
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	Yes				
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())	Yes				
Atomic addition operating on 64-bit floating point values in global memory and shared memory (atomicAdd())	No		Yes		
Warp vote functions (Warp Vote Functions)	Yes				
Memory fence functions (Memory Fence Functions)					
Synchronization functions (Synchronization Functions)					
Surface functions (Surface Functions)					
Unified Memory Programming (Unified Memory Programming)					
Dynamic Parallelism (CUDA Dynamic Parallelism)					
Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No	Yes			
Bfloat16-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion	No				Yes
Tensor Cores	No			Yes	
Mixed Precision Warp-Matrix Functions (Warp matrix functions)	No			Yes	
Hardware-accelerated `memcpy_async` (Asynchronous Data Copies)	No				Yes
Hardware-accelerated Split Arrive/Wait Barrier (Asynchronous Barrier)	No				Yes
L2 Cache Residency Management (Device Memory L2 Access Management)	No				Yes

Note that the KB and K units used in the following table correspond to 1024 bytes (i.e., a KiB) and 1024 respectively.

Table 15. Technical Specifications per Compute Capability

	Compute Capability							
Technical Specifications	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks	2^{31}-1							
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of a thread block	3							
Maximum x- or y-dimension of a block	1024							
Maximum z-dimension of a block	64							
Maximum number of threads per block	1024							
Warp size	32							
Maximum number of resident blocks per SM	16		32					
Maximum number of resident warps per SM	64							

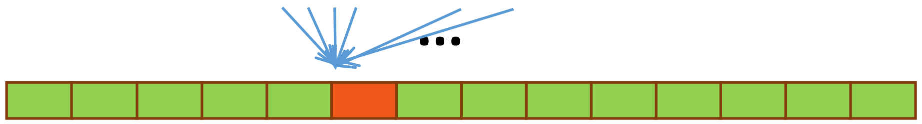
GPU Memory Hierarchy



Constant Cache

- Global variables marked by __constant__
  - constant and can't be changed in device.
- Will be cached by Constant Cache
- Located in global memory
- Good for threads that access the same address

```
__constant__ int a=10;
__global__ void kernel()
{
        a++; //error
}
```

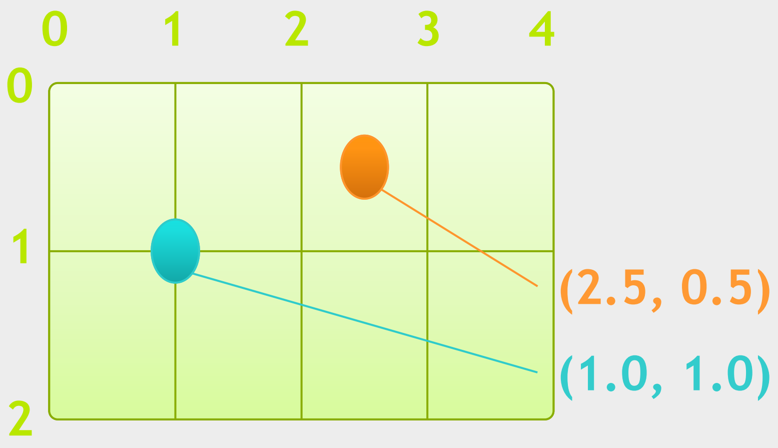


Memory addresses

Texture Cache

- Save Data as Textu
  - Provides hardwar sampling of data
  - Read-only data ca
  - Backed up by the

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware
  (Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
  - What if your problem takes a large number of read-only points as input? ☺

Specialized Libraries

- CUDPP: CUDA Data Parallel Primitives Library
  - CUDPP is a library of data-parallel algorithm primitives such as parallel prefix-sum ("scan"), parallel sort and parallel reduction.

CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle *sparseMatrixHandle*,void * *d_y*,const void * *d_x*)

Perform matrix-vector multiply y = A*x for arbitrary sparse matrix A and vector x.

```
CUDPPScanConfig config;
   config.direction = CUDPP_SCAN_FORWARD; config.exclusivity =
   CUDPP_SCAN_EXCLUSIVE; config.op = CUDPP_ADD;
   config.datatype = CUDPP_FLOAT; config.maxNumElements = numElements;
   config.maxNumRows = 1;
   config.rowPitch = 0;
cudppInitializeScan(&config);
cudppScan(d_odata, d_idata, numElements, &config);
```

CUFFT

- No. of elements<8192 slower than fftw
- >8192, 5x speedup over threaded fftw
  and 10x over serial fftw.

CUBLAS

- Cuda Based Linear Algebra Subroutines
- Saxpy, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae.
    - http://graphics.tu-bs.de/publications/Fernandez08TechReport.pdf