CUDA

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
  - Exam soon!
  - Pedagogical machines
- Agenda
  - CUDA

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

Faux Quiz Questions (5 min, any 2)

1. Why do languages like CUDA and OpenCL have both blocks and threads?
2. What does the __shared__ keyword do? How is it implemented by HW?
3. CUDA kernels have implicit barrier synchronization: __syncthreads(). Is it necessary? Why or why not?
4. Is traditional locking implementable on a GPU? What alternatives are there for synchronizing / avoiding races?
5. How is occupancy defined (in CUDA nomenclature)?
6. What is control flow divergence? How does it impact performance?
7. What is a bank conflict?
8. What is the difference between a thread block scheduler and a warp scheduler?
9. Compare/contrast coarse/fine/simultaneous multi-threading.
10. In CUDA, what is the difference between grids, blocks, and threads? What is their counterpart in OpenCL?

Parallel Algorithms Redux

Parallel Algorithms Redux

Key idea:

Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map

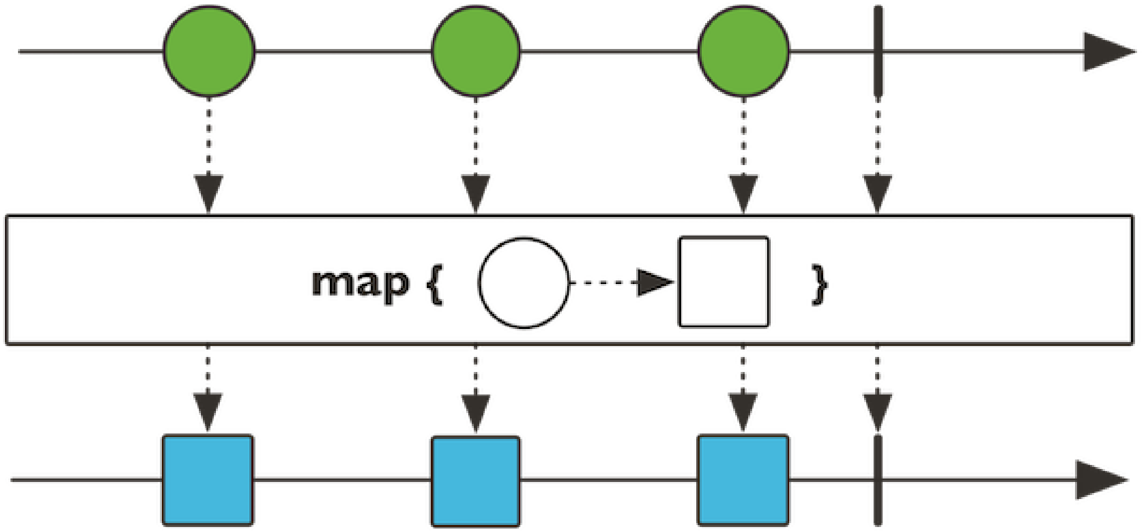
Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map



Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map

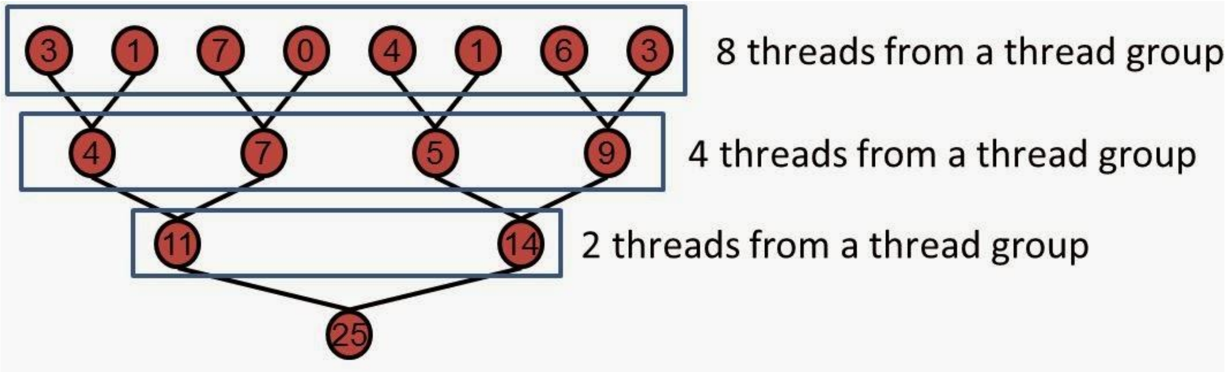
Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map
- Reductions



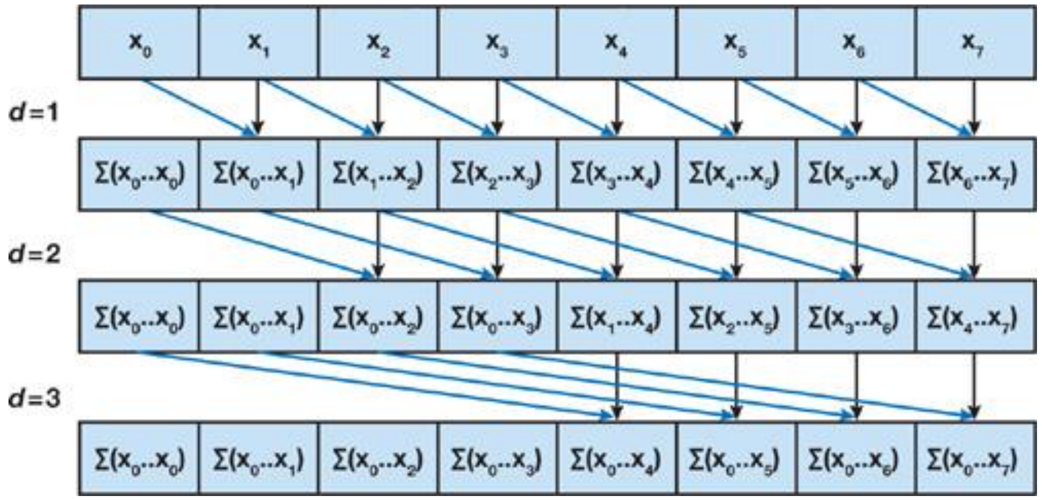
Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map
- Reductions
- Scans



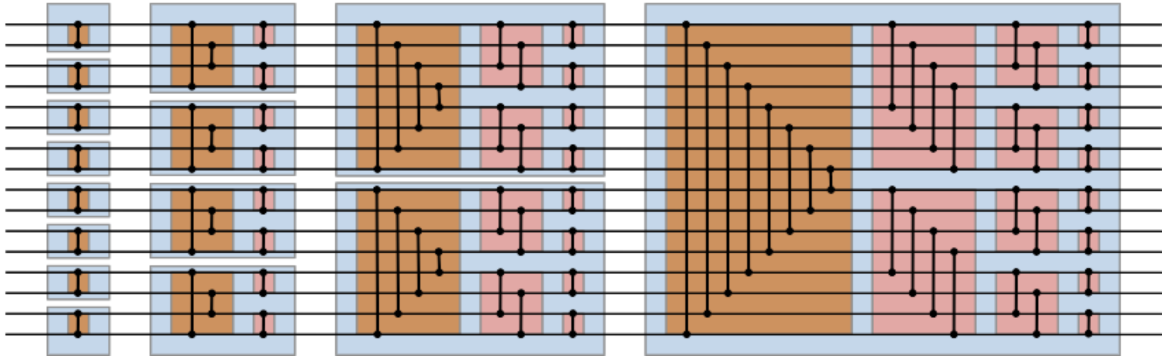
Parallel Algorithms Redux

Key idea:

Express sequential algorithms as combinations of parallel patterns

Examples:

- Map
- Reductions
- Scans
- Re-orderings (scatter/gather/sort)



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

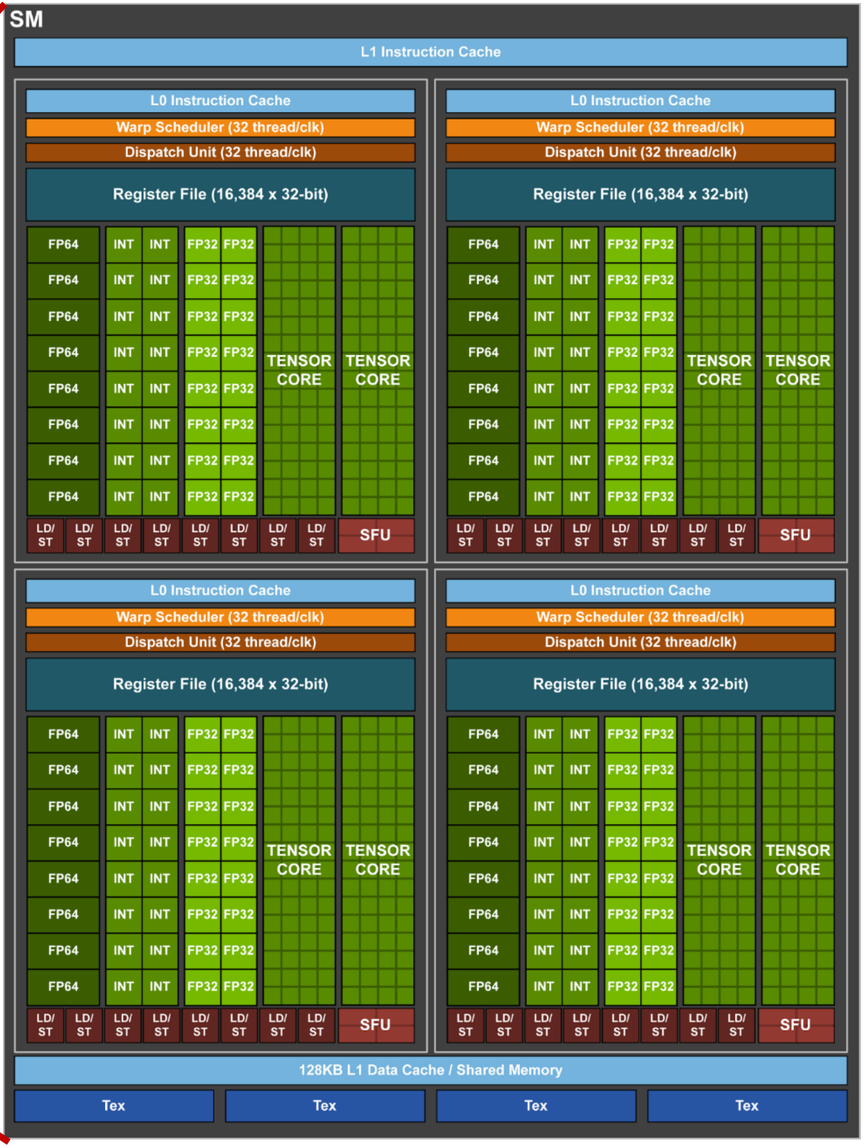
Review

- 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

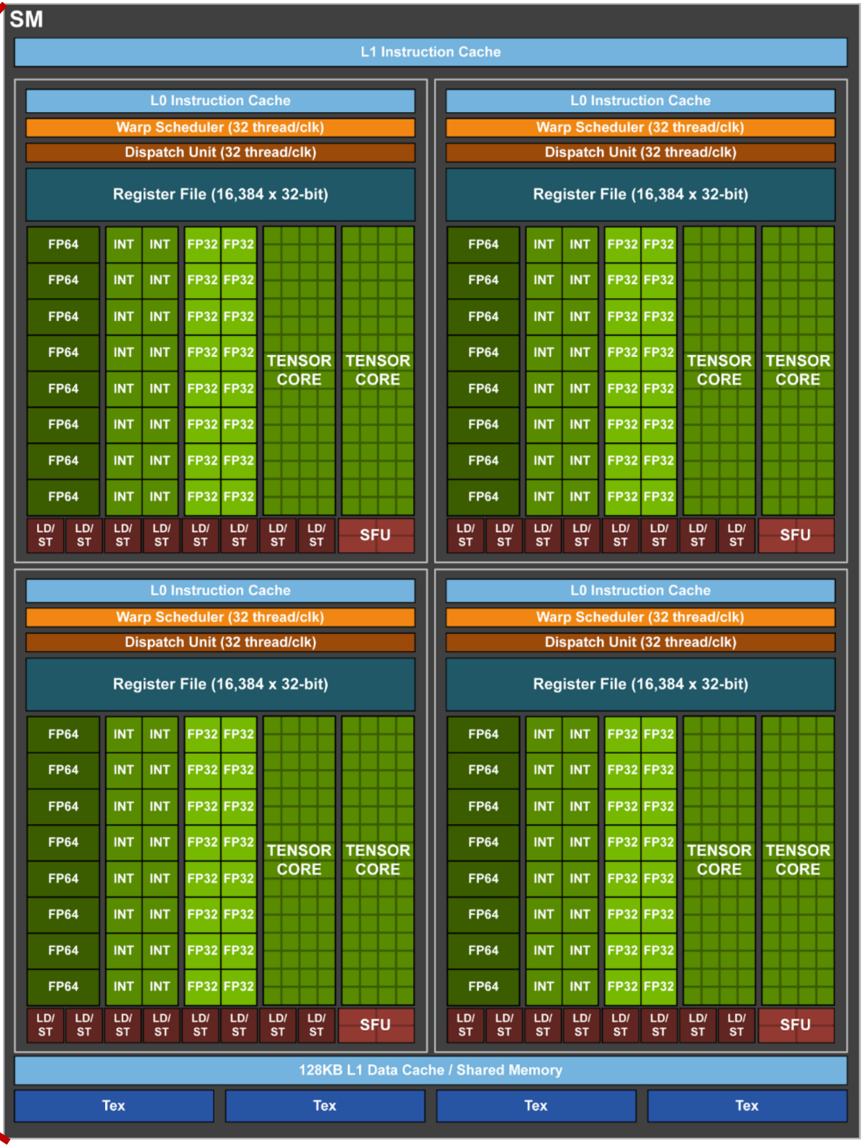
Review



- 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

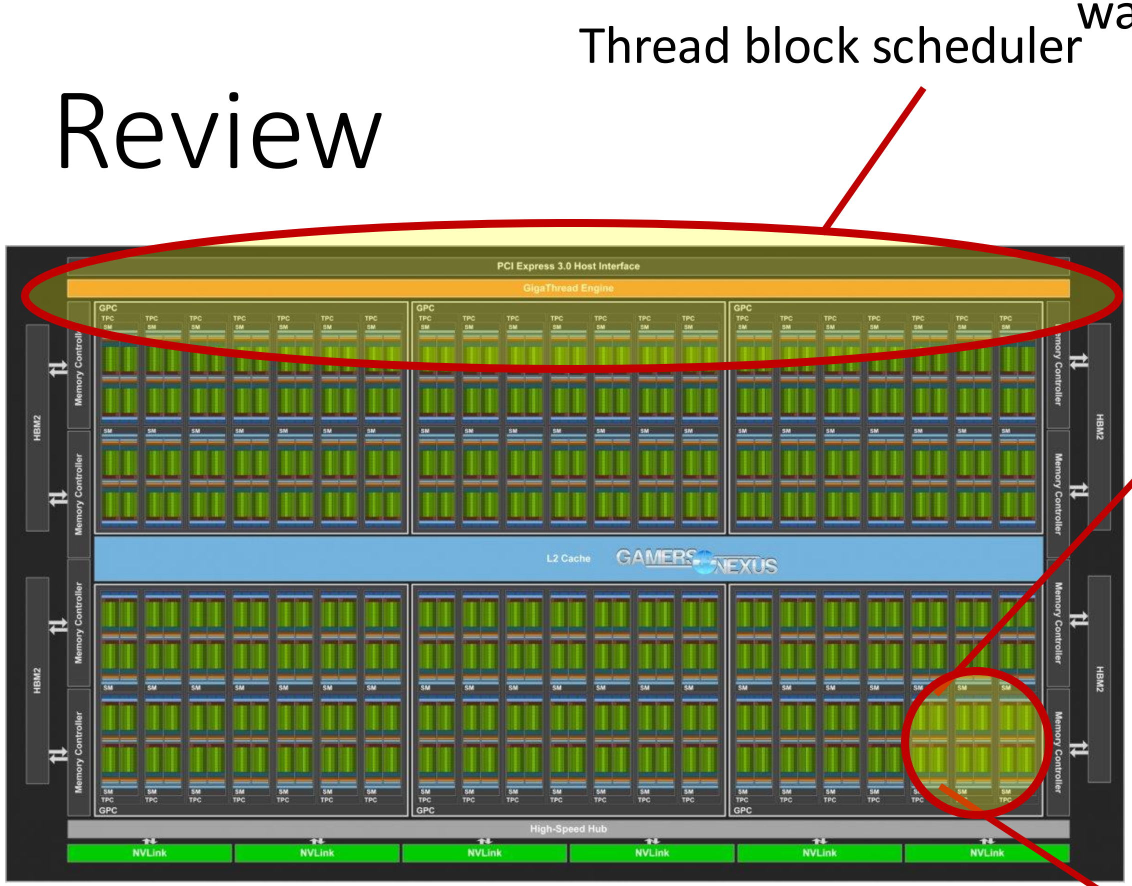
Review

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



- 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

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

CONCEPTS

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

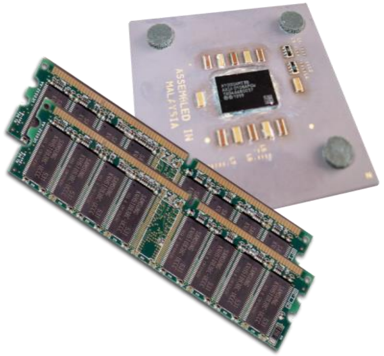
HELLO WORLD!

CONCEPTS

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

Heterogeneous Computing

- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



Host

Device

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

serial code

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N       1024
#define RADIUS    3
#define BLOCK_SIZE 16

__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];
	}

	// Synchronize (ensure all the data is available)
	__syncthreads();

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

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

void fill_ints(int *x, int n) {
	fill_n(x, n, 1);
}

int main(void) {
	int *in, *out;          // host copies of a, b, c
	int *d_in, *d_out;      // device copies of a, b, c
	int size = (N + 2*RADIUS) * sizeof(int);

	// Alloc space for host copies and setup values
	in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
	out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

	// Alloc space for device copies
	cudaMalloc((void **)&d_in,  size);
	cudaMalloc((void **)&d_out, size);

	// Copy to device
	cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
	cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

	// Launch stencil_1d() kernel on GPU
	stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

	// Copy result back to host
	cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

	// Cleanup
	free(in); free(out);
	cudaFree(d_in); cudaFree(d_out);
	return 0;
}
```

parallel fn

serial code

parallel code

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code

Heterogeneous Computing

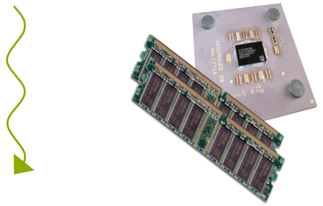
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code



Heterogeneous Computing

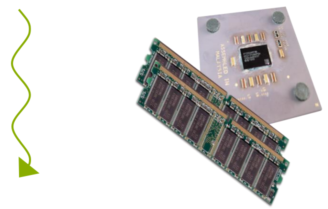
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N        1024
#define RADIUS    3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

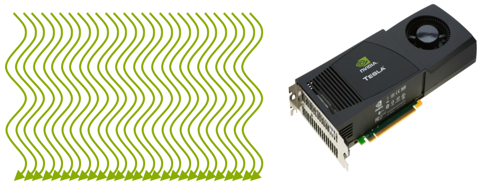
parallel fn

serial code

parallel code

serial code





Heterogeneous Computing

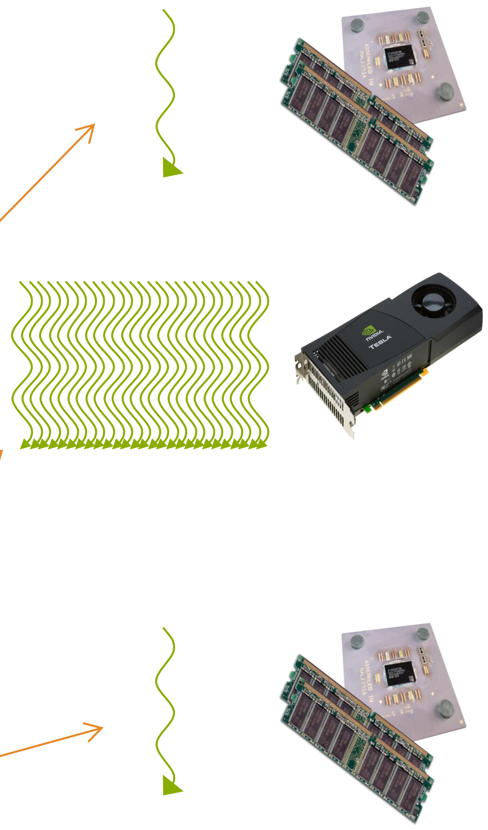
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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];
        }

        // Synchronize (ensure all the data is available)
        __syncthreads();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;          // host copies of a, b, c
        int *d_in, *d_out;      // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

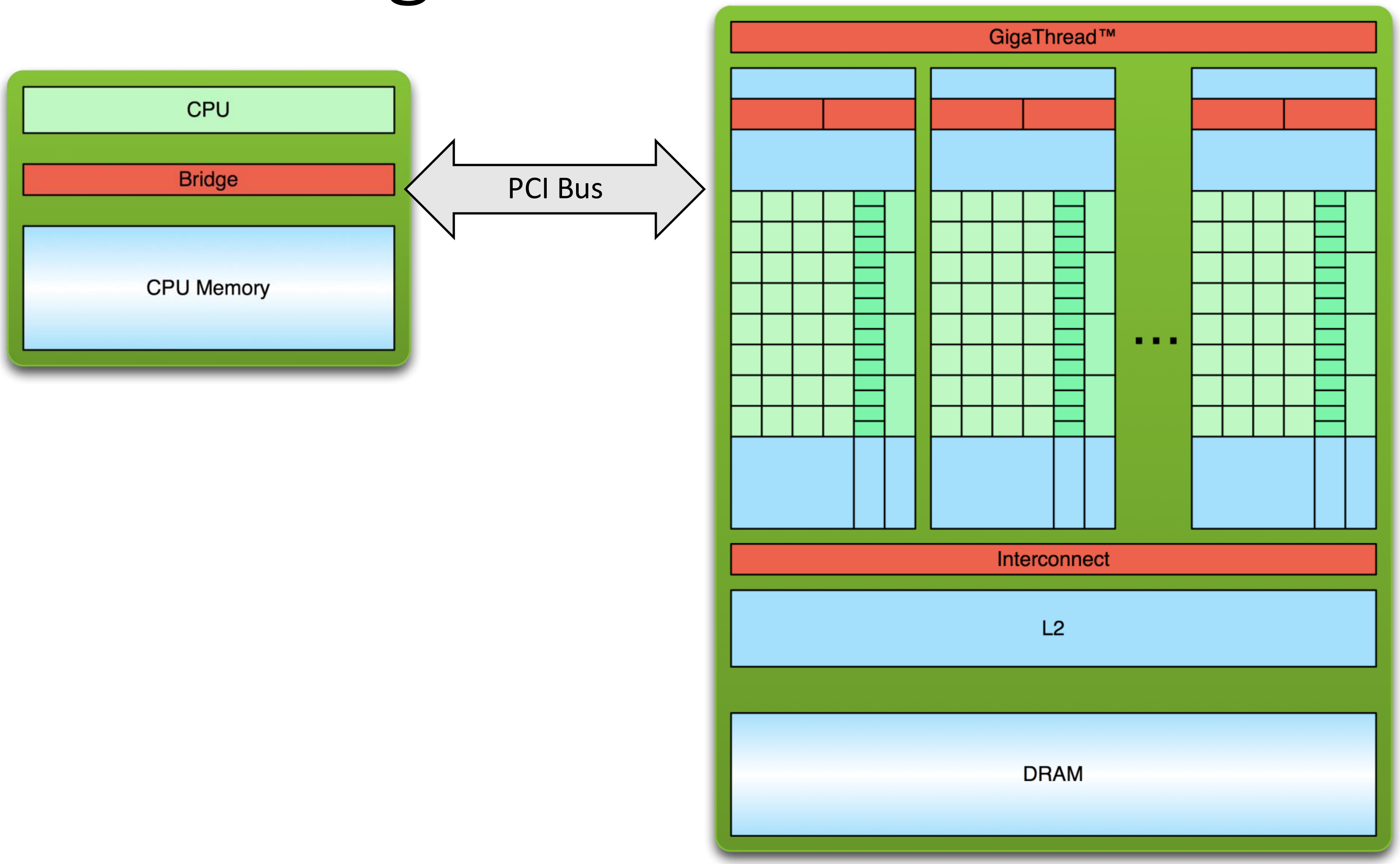
serial code

parallel code

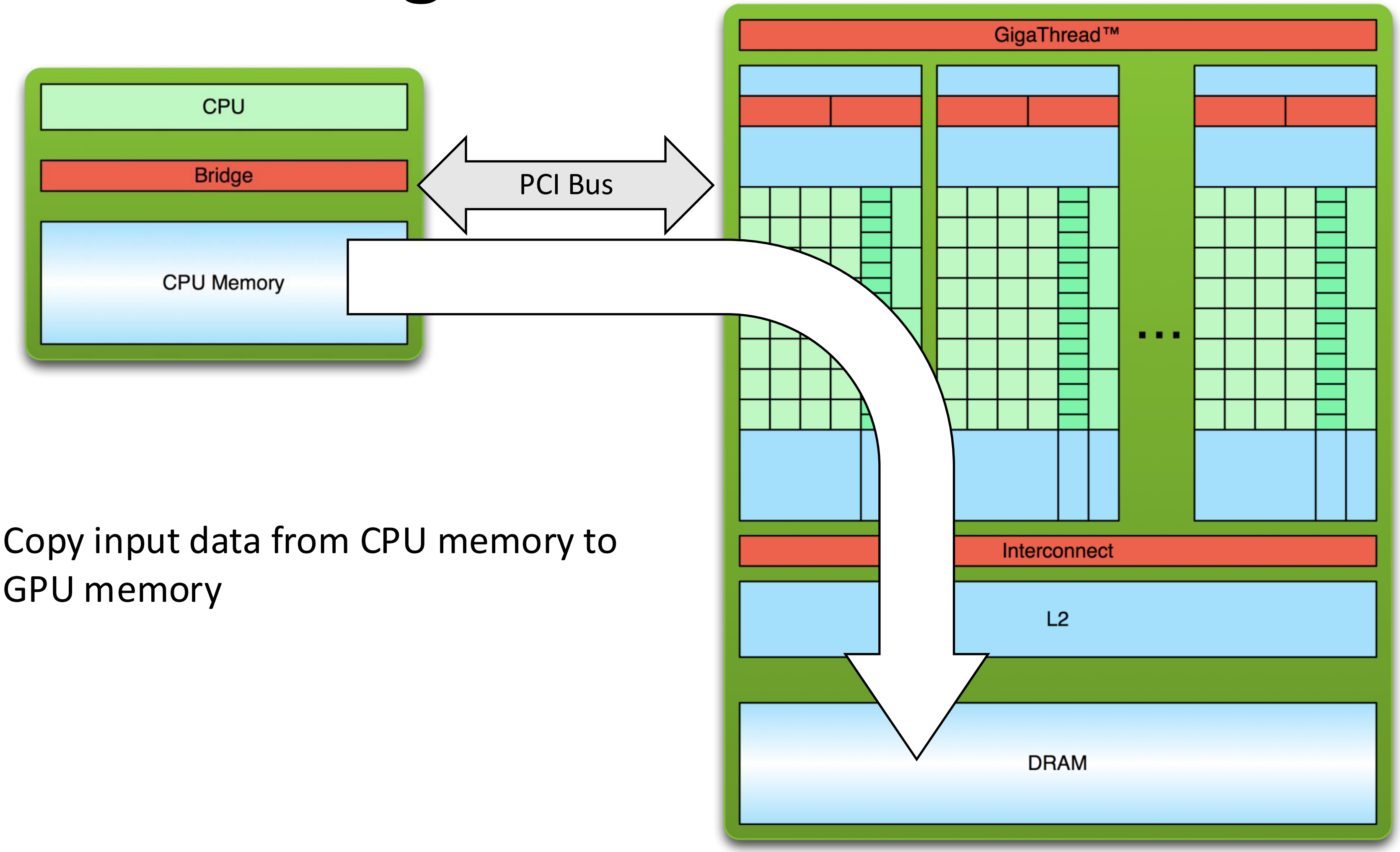
serial code



Simple Processing Flow

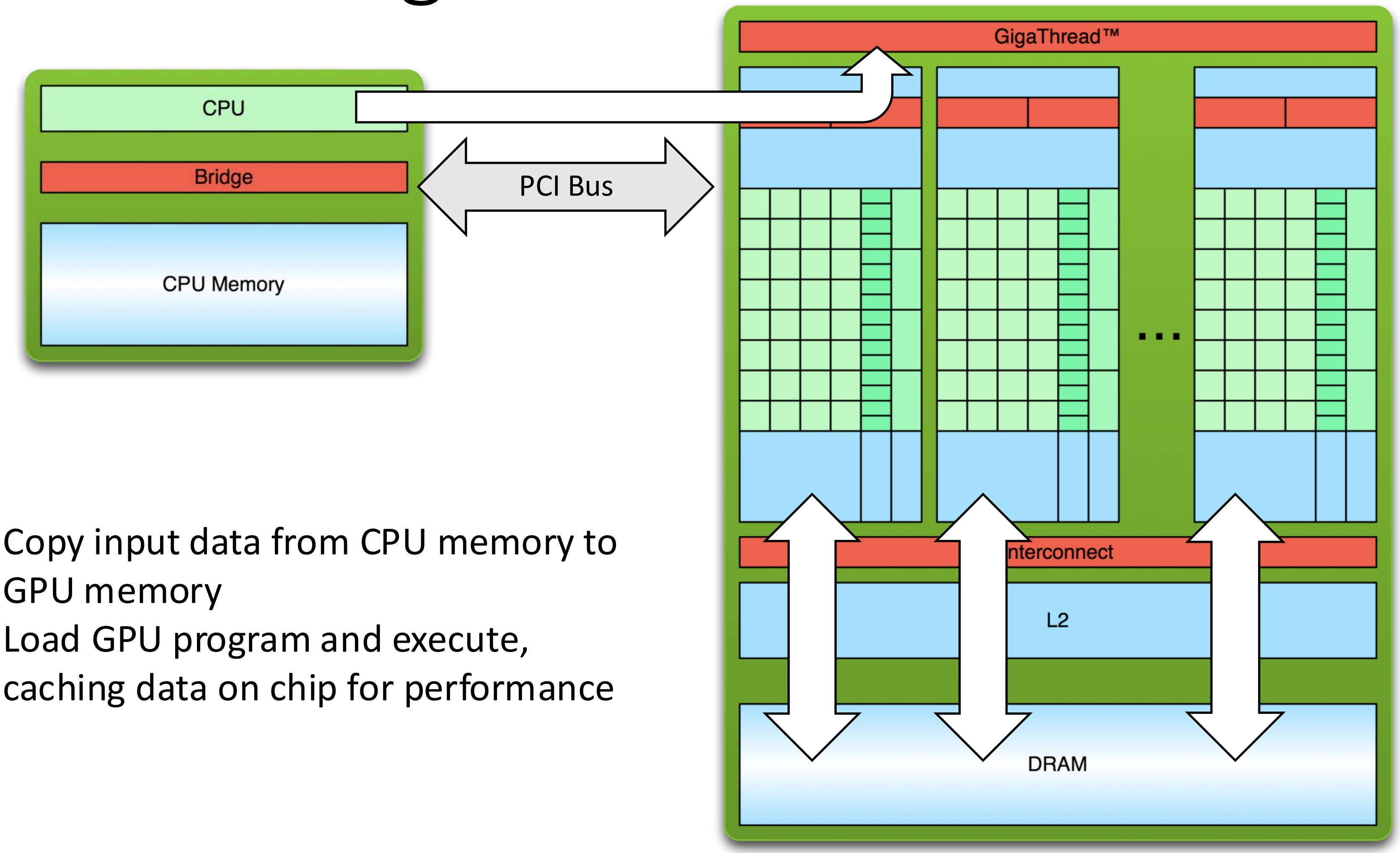


Simple Processing Flow



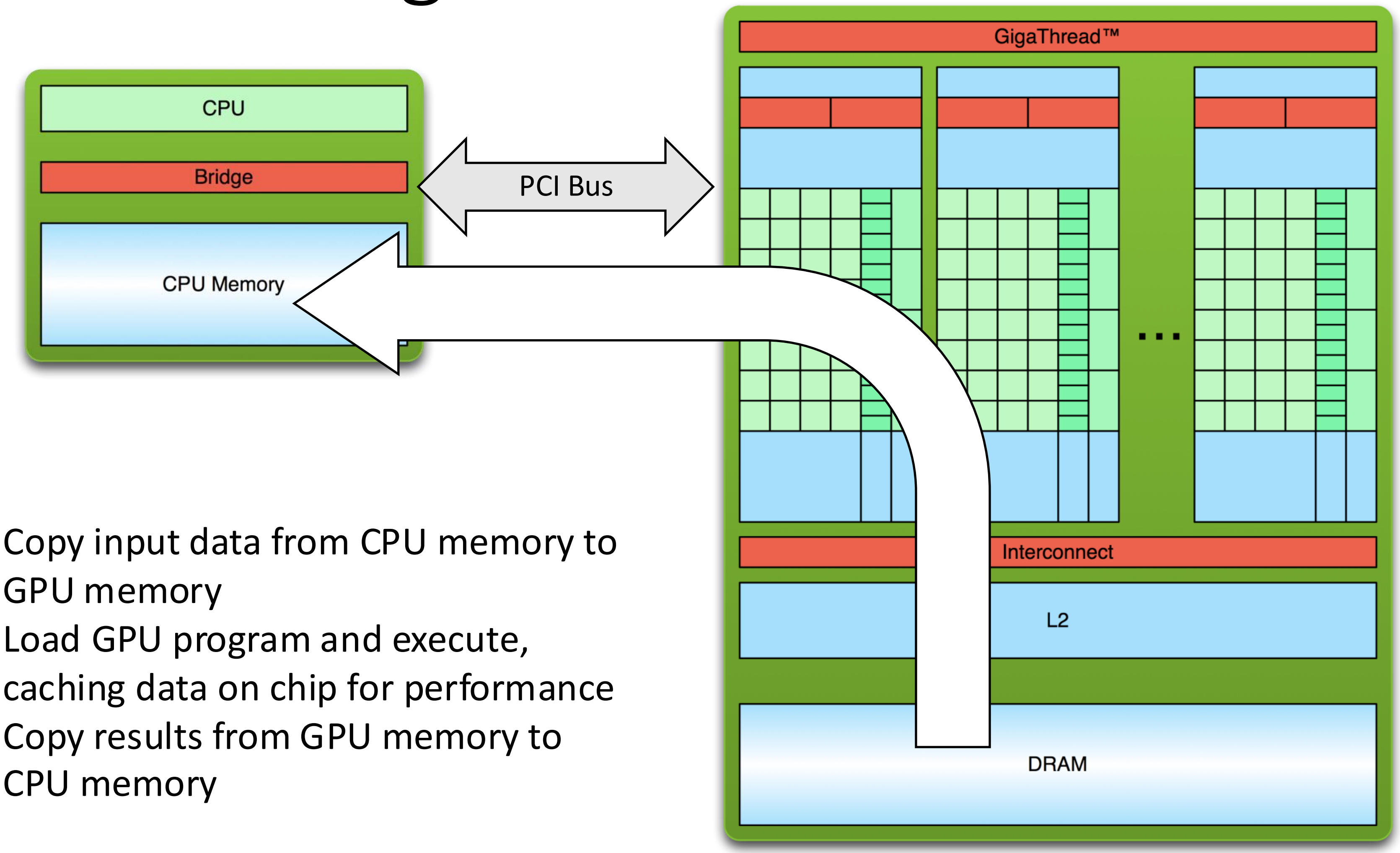
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

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

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

- Two new syntactic elements…

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code
- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - `gcc`, `cl.exe`

Hello World! with Device COde

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

Hello World! with Device COde

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

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

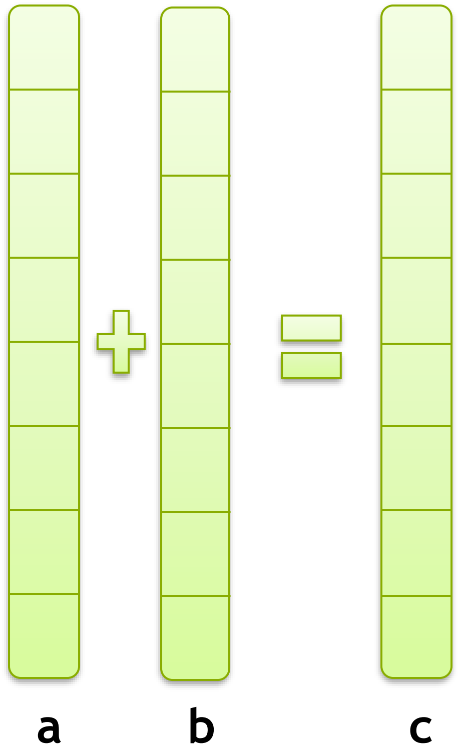
Output:

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

- `mykernel()` does nothing, somewhat anticlimactic!

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!
- We need a more interesting example…
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

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

Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

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

Addition on the Device

- Note that we use pointers for the variables

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

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory

- We need to allocate memory on the GPU

Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory

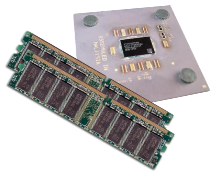
    May be passed to/from host code

    May *not* be dereferenced in host code

  - *Host* pointers point to CPU memory

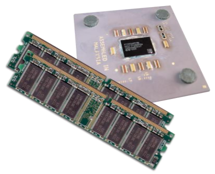
    May be passed to/from device code

    May *not* be dereferenced in device code



Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code



- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Addition on the Device: `add()`

- Returning to our **`add()`** kernel

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

- Let’s take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
```

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                 // host copies of a, b, c
        int *d_a, *d_b, *d_c;        // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
```

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
```

Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);
```

Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
```

Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

RUNNING IN PARALLEL

CONCEPTS

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

RUNNING IN PARALLEL

CONCEPTS

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

Getting Parallel

- GPU computing is about massive parallelism
    - So how do we run code in parallel on the device?

Getting Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

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

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

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is a block

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition

- Terminology: each parallel invocation of `add()` is a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

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

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

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

Vector Addition on the Device

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

- On the device, each block can execute in parallel:

Block 0	Block 1	Block 2	Block 3
`c[0] = a[0] + b[0];`	`c[1] = a[1] + b[1];`	`c[2] = a[2] + b[2];`	`c[3] = a[3] + b[3];`

Vector Addition on the Device: `add()`

- Returning to our parallelized **`add()`** kernel

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

- Let’s take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
   int *a  *b  *c                        // host copies of a, b, c
   int *d_a, *d_b, *d_c;                 // device copies of a, b, c
```

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a  *b  *c                    // host copies of a, b, c
    int *d_a, *d_b, *d_c;             // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
```

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
   int *a  *b  *c                         // host copies of a, b, c
   int *d_a, *d_b, *d_c;                  // device copies of a, b, c
   int size = N * sizeof(int);

   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);

   // Alloc space for host copies of a, b, c and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);
```

Vector Addition on the Device: `main()`

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
```

Vector Addition on the Device: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

INTRODUCING THREADS

CONCEPTS

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

CUDA Threads

- Terminology: a block can be split into parallel threads

- Change `add()` to use parallel *threads* instead of parallel *blocks:*

- Use **`threadIdx.x`** instead of **`blockIdx.x`**

CUDA Threads

- Terminology: a block can be split into parallel threads
- Change `add()` to use parallel *threads* instead of parallel *blocks:*

- Use **`threadIdx.x`** instead of **`blockIdx.x`**
- Need to make one change in **`main()`**…

CUDA Threads

- Terminology: a block can be split into parallel threads

- Change `add()` to use parallel *threads* instead of parallel *blocks:*

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

- Use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`…

Vector Addition Using Threads: `main()`

```
#define N 512
int main(void) {
    int *a, *b, *c;                    // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Vector Addition Using Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a);
    cudaFree(
    return 0;
}
```

Traditional CPU	Graphics Shaders	CUDA	OpenCL
SIMD lane	thread	thread	work-item
~thread	-	warp	-
	thread group	block	work group
	-	grid	N-D range

CONCEPTS

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

COMBINING THREADS AND BLOCKS

Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with *one thread* each (M:1)
  - One block with *many threads* (1:M)

Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with *one thread* each (M:1)
  - One block with *many threads* (1:M)

- How to make vector addition to use both blocks and threads?

Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with *one thread* each (M:1)
  - One block with *many threads* (1:M)

- How to make vector addition to use both blocks and threads?

- How to deal with blockIdx.* vs threadIdx.*?

Indexing Arrays with Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

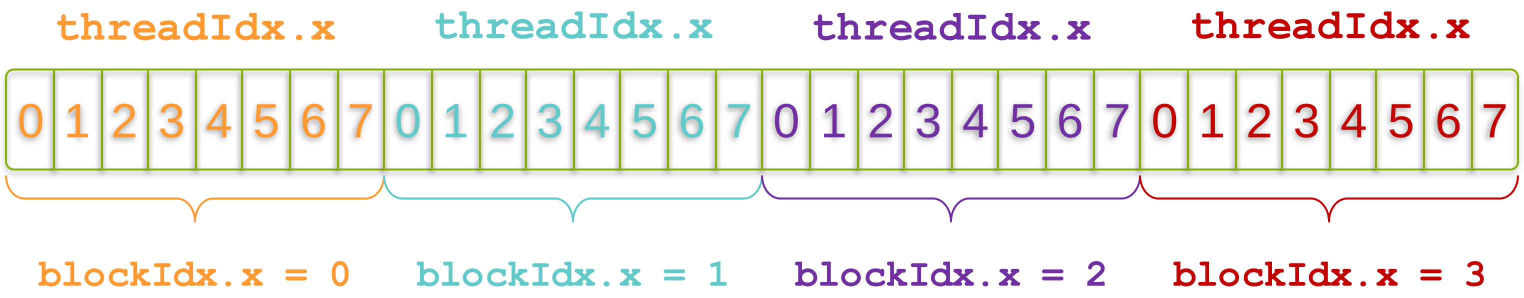
Indexing Arrays with Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7

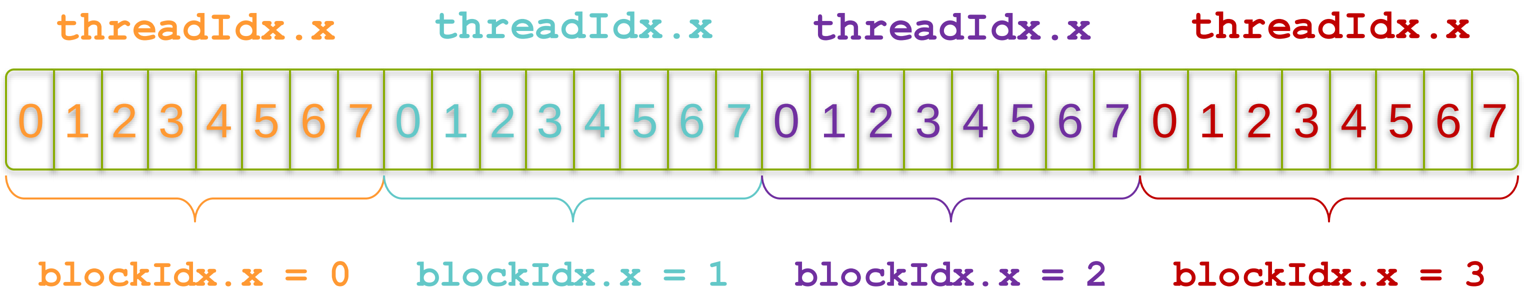
Indexing Arrays with Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)



Indexing Arrays with Blocks and Threads

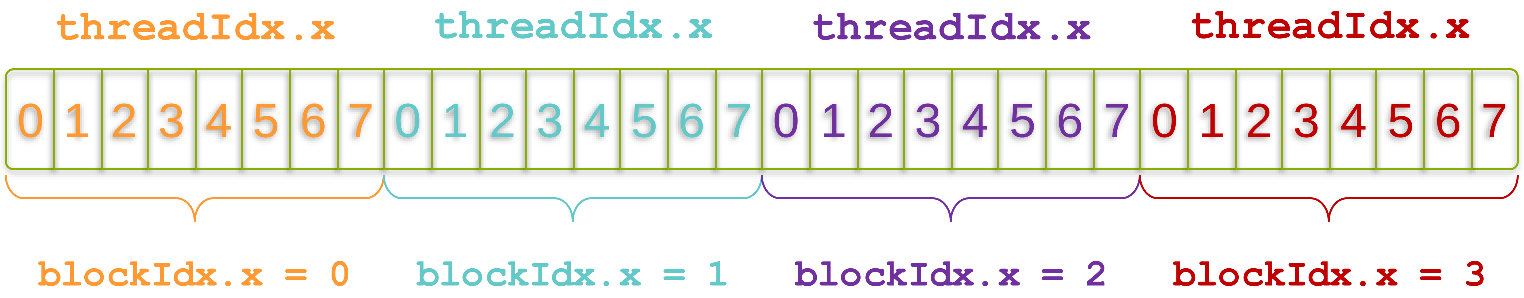
- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)



- With M threads/block, unique index per thread is :

Indexing Arrays with Blocks and Threads

- Most kernels use ***both*** `blockIdx.x` and `threadIdx.x`
  - Index an array with one elem. per thread (8 threads/block)

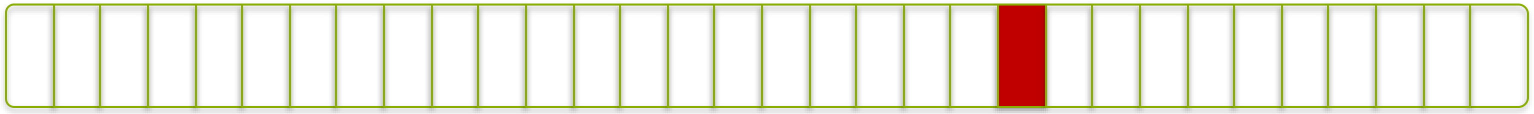


- With M threads/block, unique index per thread is :

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

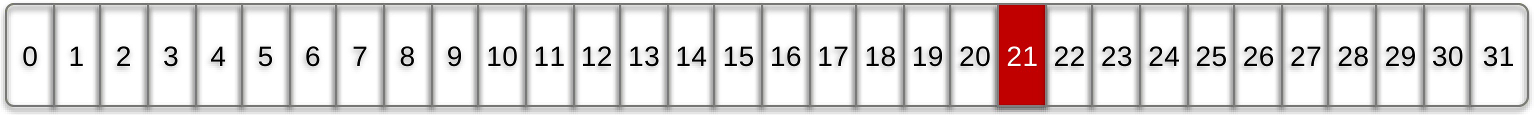
Indexing Arrays: Example

- Which thread will operate on the red element?
  - M=8 Threads, 4 blocks



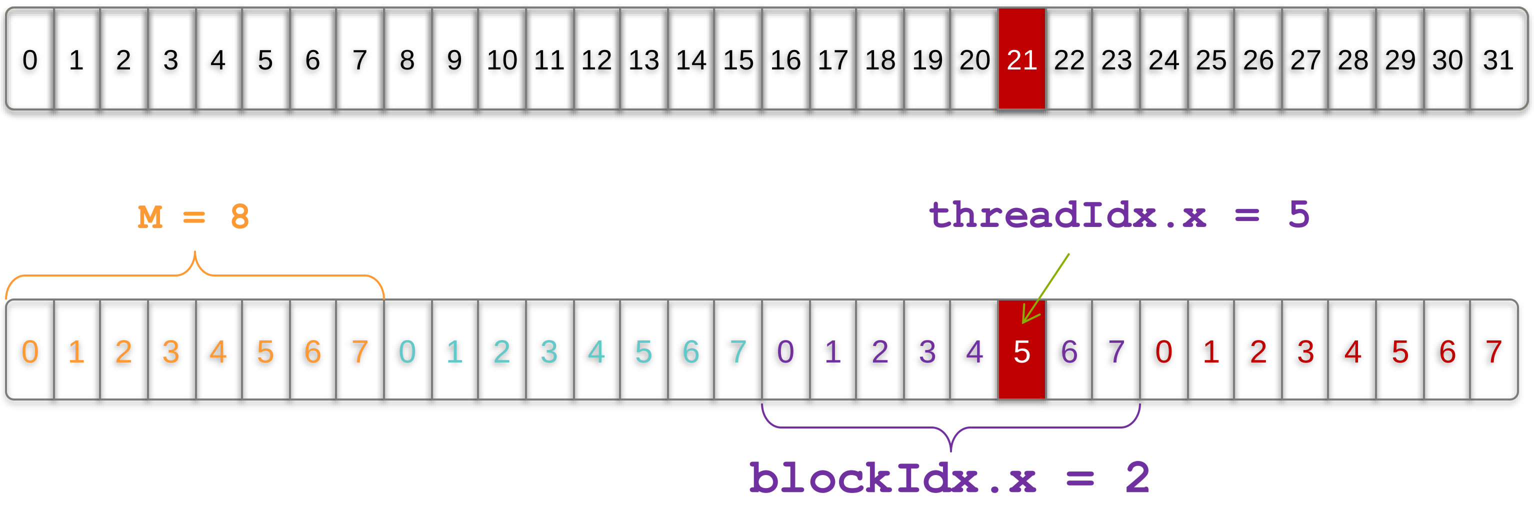
Indexing Arrays: Example

- Which thread will operate on the red element?
  - M=8 Threads, 4 blocks



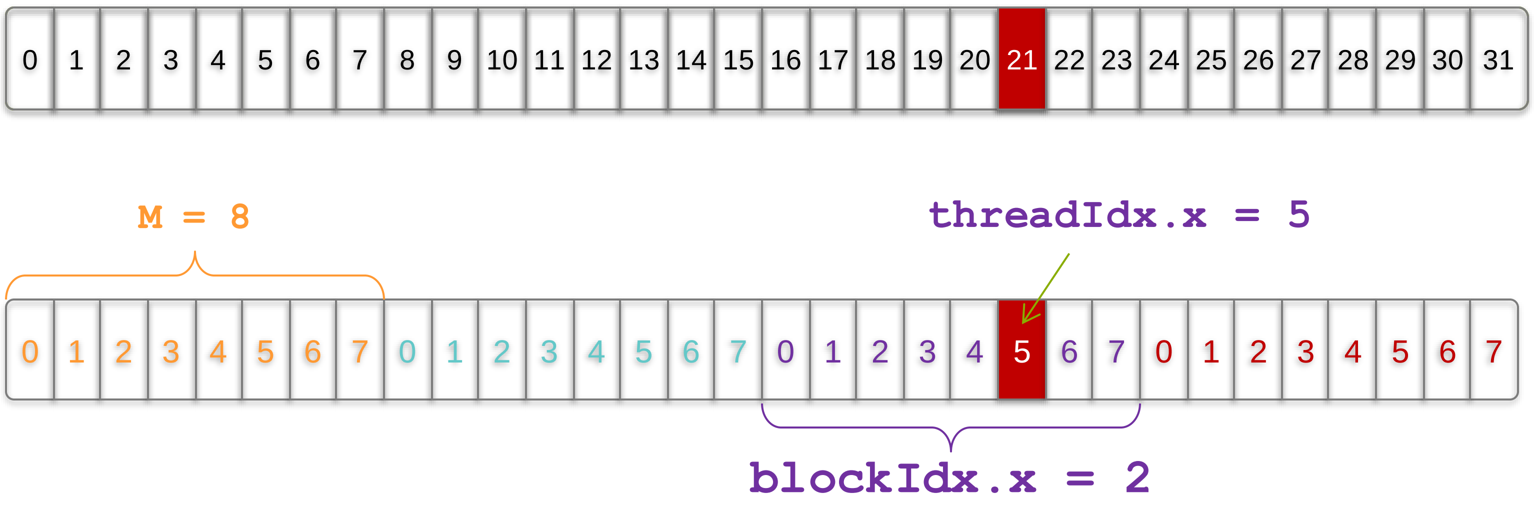
Indexing Arrays: Example

- Which thread will operate on the red element?
  - M=8 Threads, 4 blocks



Indexing Arrays: Example

- Which thread will operate on the red element?
    - M=8 Threads, 4 blocks




```
int index = threadIdx.x + blockIdx.x * M;
          =         5   +        2    * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** for threads per block

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

Vector Addition with Blocks and Threads

- Use the built-in variable **`blockDim.x`** for threads per block

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

- Combined `add()` using parallel threads *and* blocks

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined `add()` using parallel threads *and* blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;               // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Addition with Blocks and Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Anyone see a problem?

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

- Update the kernel launch:

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

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

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

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

- To look closer, we need a new example…

COOPERATING THREADS

CONCEPTS

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

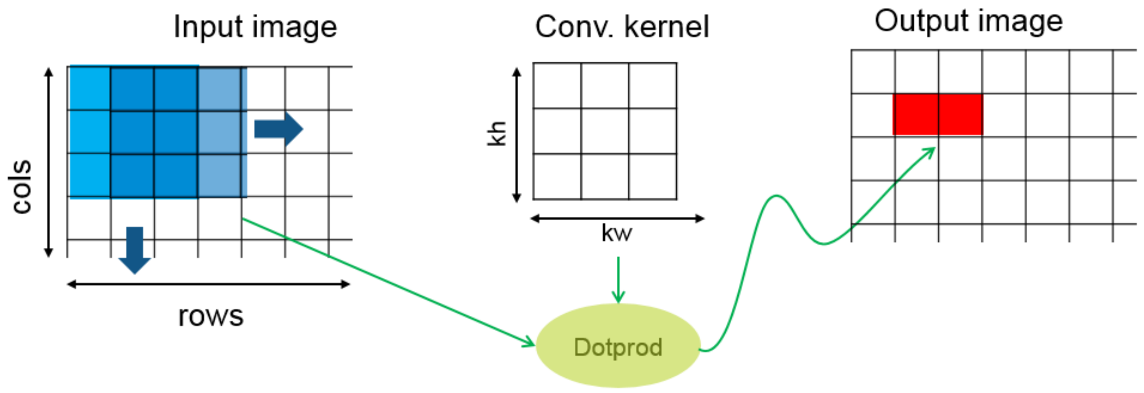
Stencils

Stencils

- Each pixel → function of neighbors

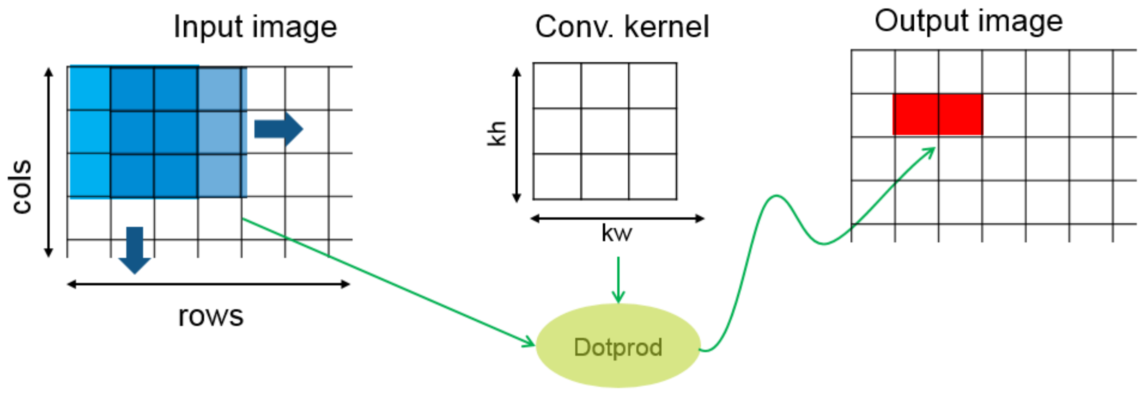
Stencils

- Each pixel → function of neighbors



Stencils

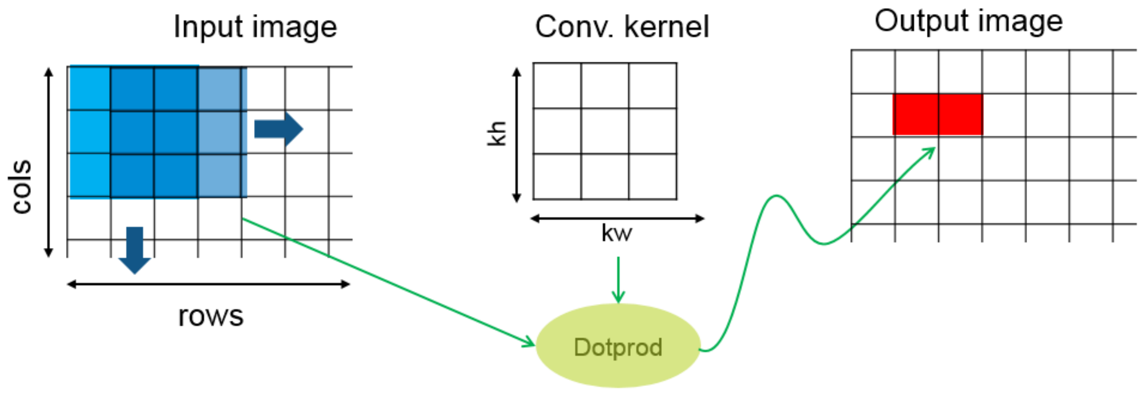
- Each pixel → function of neighbors
- Edge detection:



Stencils

- Each pixel → function of neighbors
- Edge detection:

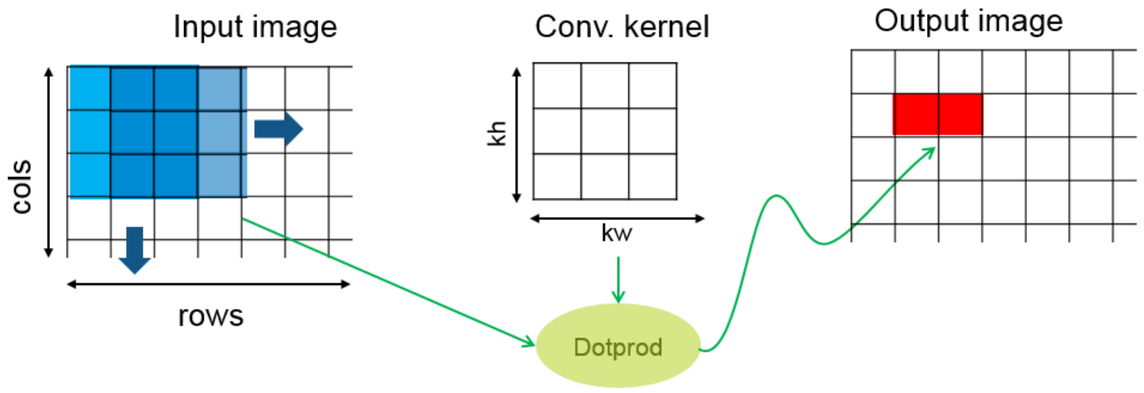
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

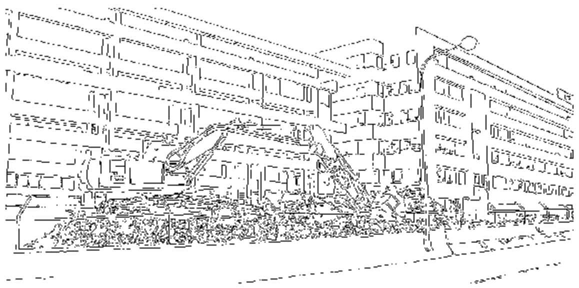


Stencils

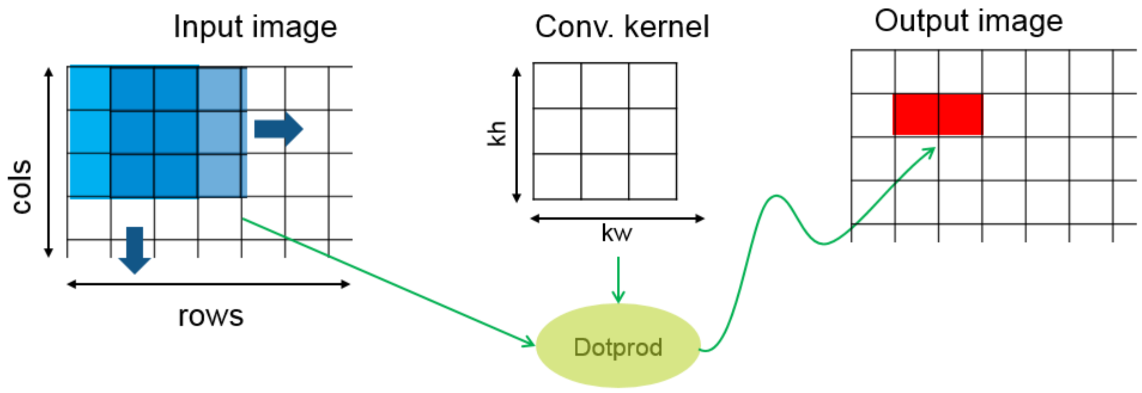
- Each pixel → function of neighbors
- Edge detection:

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



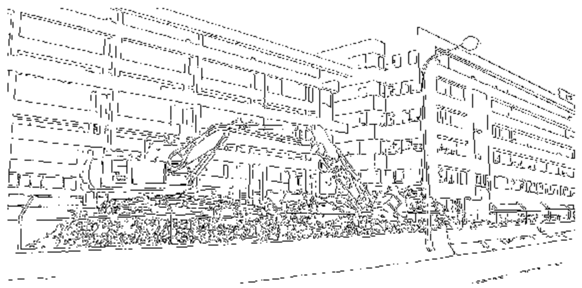


Stencils



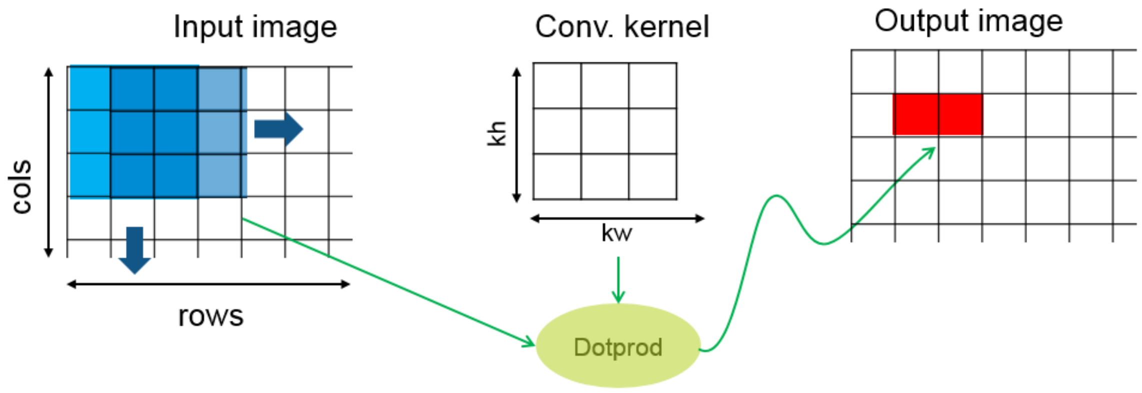
- Each pixel → function of neighbors
- Edge detection:

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



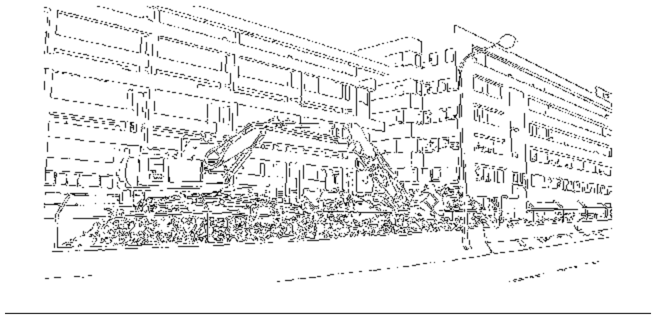
- Blur:

Stencils



- Each pixel → function of neighbors
- Edge detection:

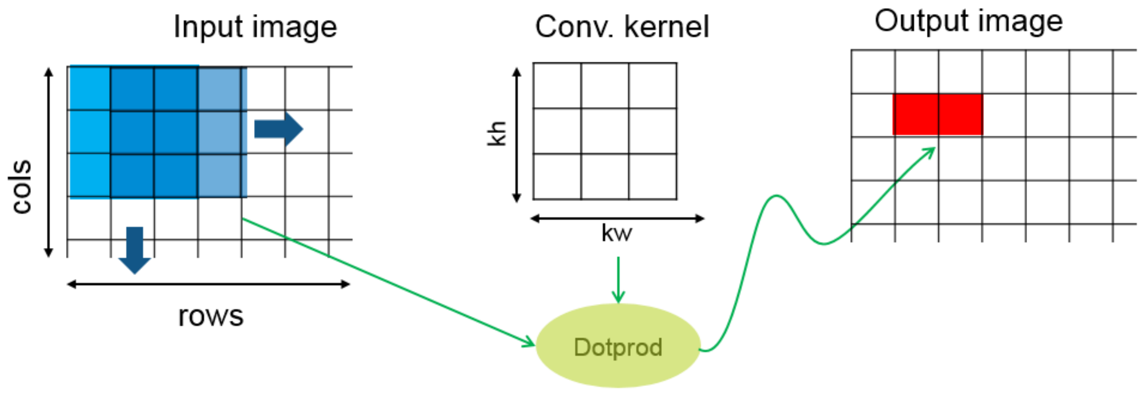
$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$



- Blur:

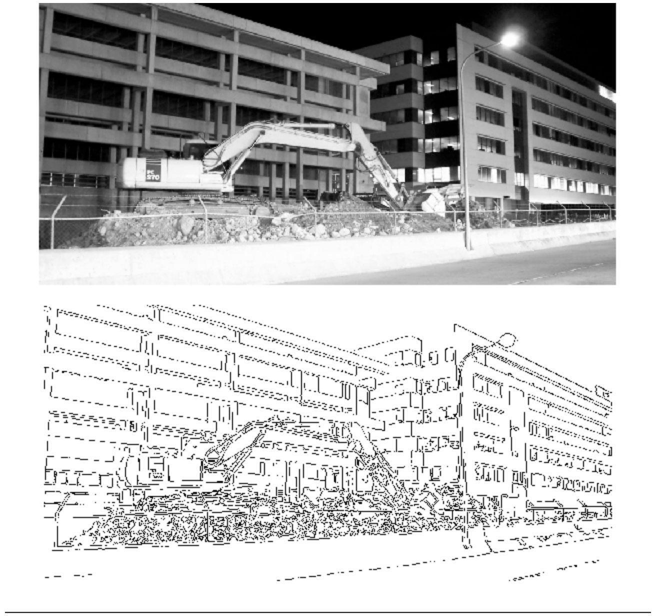
1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Stencils



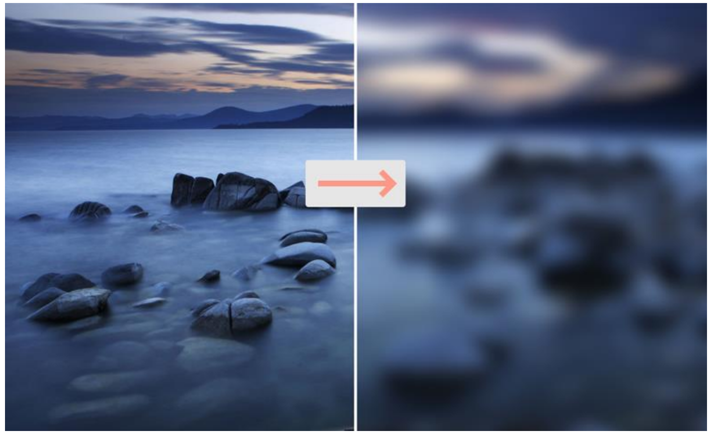
- Each pixel → function of neighbors
- Edge detection:

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



- 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

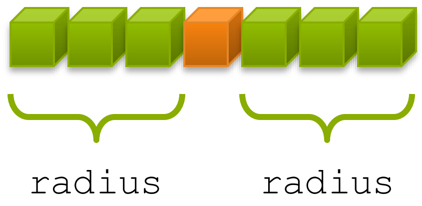
- Consider 1D stencil over 1D array of elements
  - Each output element is the sum of input elements within a radius

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

Implementation within a block

- Each thread: process 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

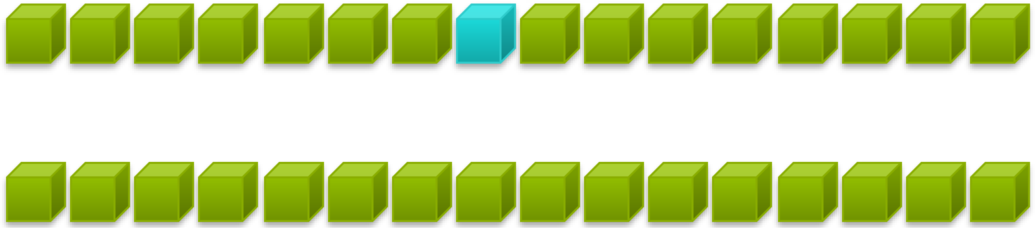
- Each thread: process 1 output element
  - blockDim.x elements per block
- Input elements read many times
  - With radius 3, each input element is read seven times

```
__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;
}
```

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

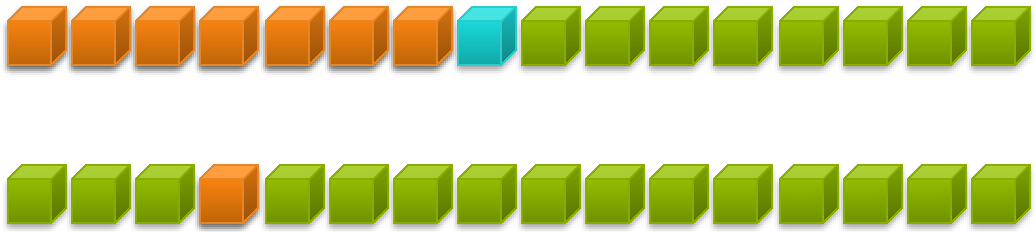
```
__global__ void stencil_1d(int *in, int *out) {
    // note: idx comp & edge conditions omitted…
    int result = 0;
    for (int offset = -R; offset <= R; offset++)
      result += in[idx + offset];

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



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

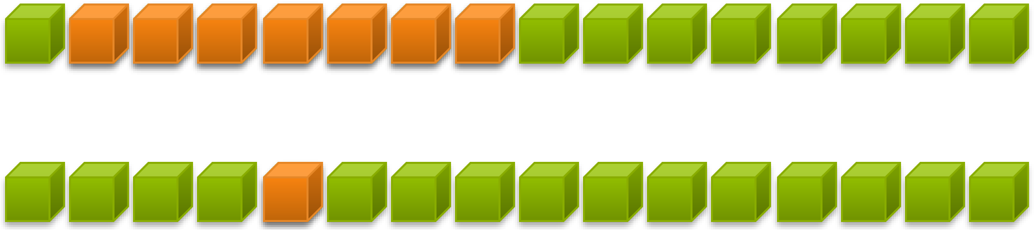
```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

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



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

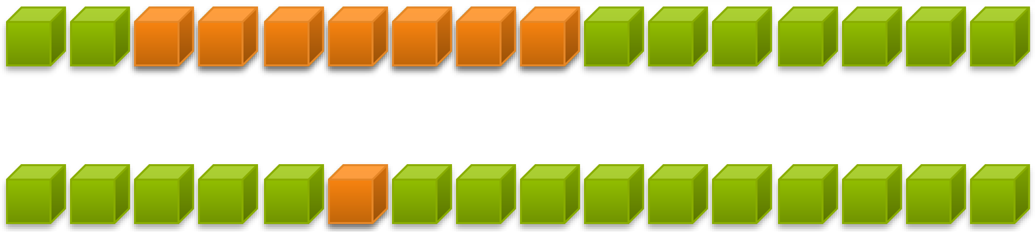
```
__global__ void stencil_1d(int *in, int *out) {
   // note: idx comp & edge conditions omitted…
   int result = 0;
   for (int offset = -R; offset <= R; offset++)
     result += in[idx + offset];

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



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

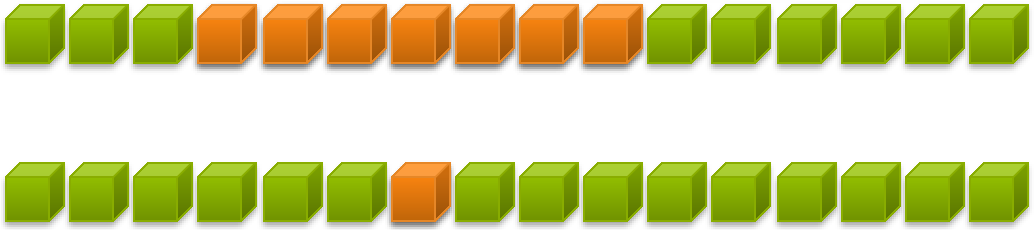
```
__global__ void stencil_1d(int *in, int *out) {
   // note: idx comp & edge conditions omitted…
   int result = 0;
   for (int offset = -R; offset <= R; offset++)
     result += in[idx + offset];

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



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

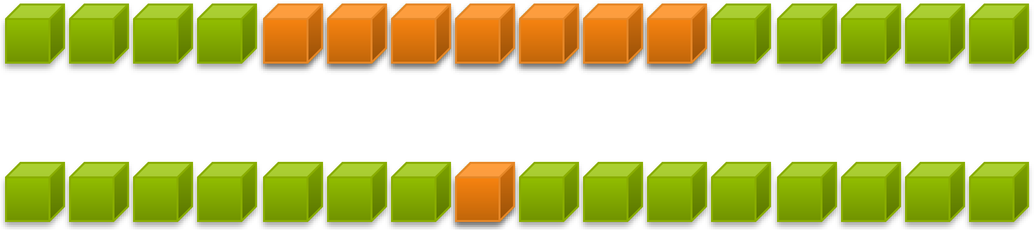
```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

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



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

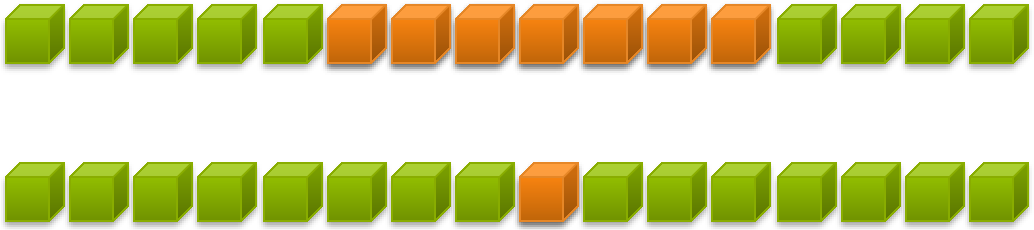
```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

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



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

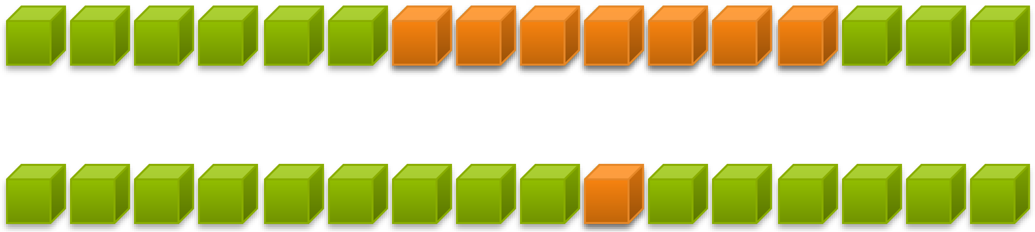
```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

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



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;
}
```

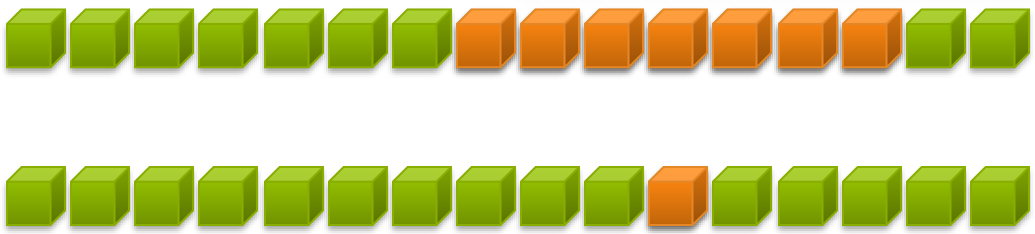


Implementation within a block

- 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

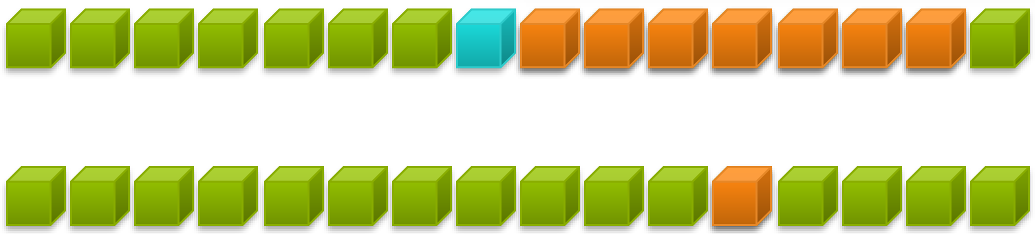
```
__global__ void stencil_1d(int *in, int *out) {
    // note: idx comp & edge conditions omitted…
    int result = 0;
    for (int offset = -R; offset <= R; offset++)
      result += in[idx + offset];

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



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

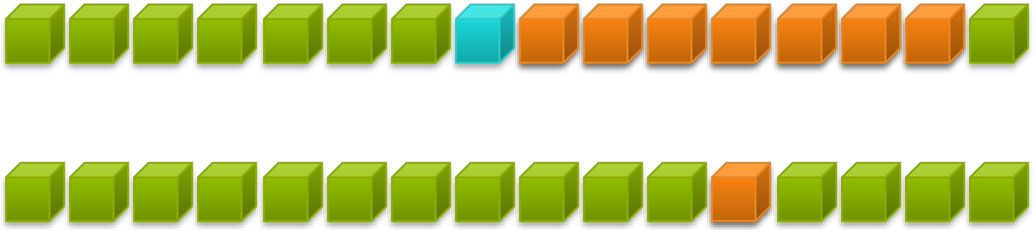
```
__global__ void stencil_1d(int *in, int *out) {
  // note: idx comp & edge conditions omitted…
  int result = 0;
  for (int offset = -R; offset <= R; offset++)
    result += in[idx + offset];

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



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;
}
```



Why is this a problem?

Sharing Data Between Threads

Sharing Data Between Threads

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

Sharing Data Between Threads

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

- Extremely fast on-chip memory, user-managed

Sharing Data Between Threads

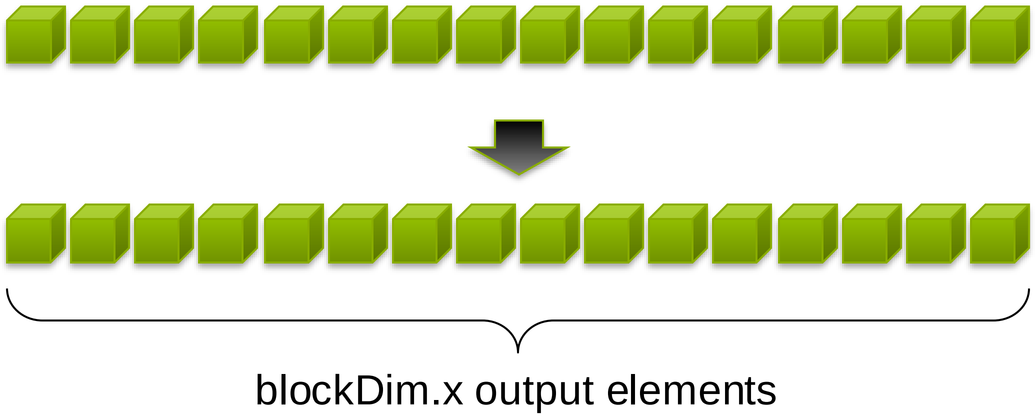
- Terminology: within a block, threads share data via ***shared memory***
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block

Sharing Data Between Threads

- Terminology: within a block, threads share 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 with Shared Memory

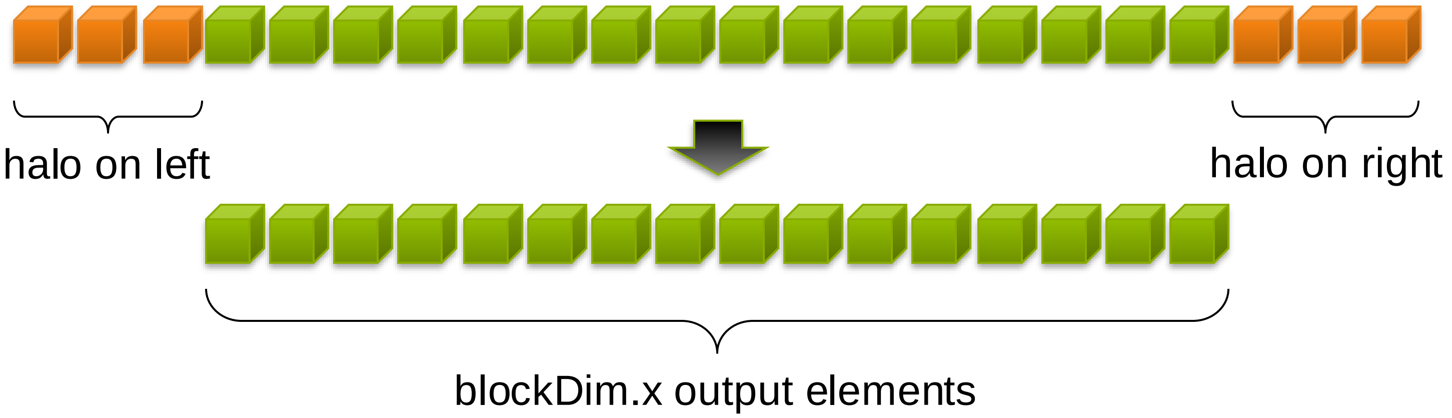
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) elements from memory to shared
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory



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];
```

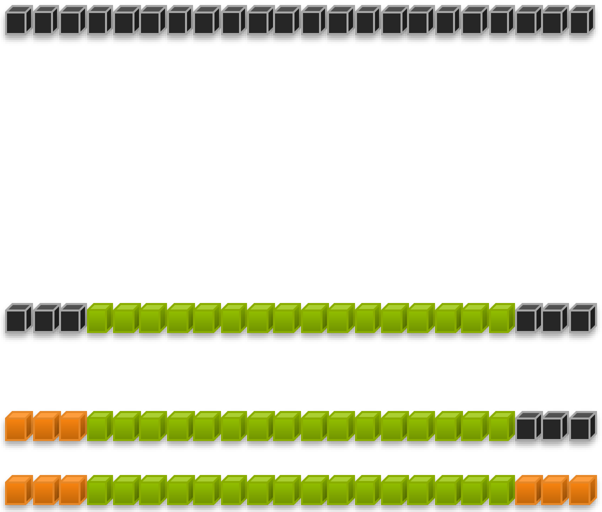
```
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // 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];
```

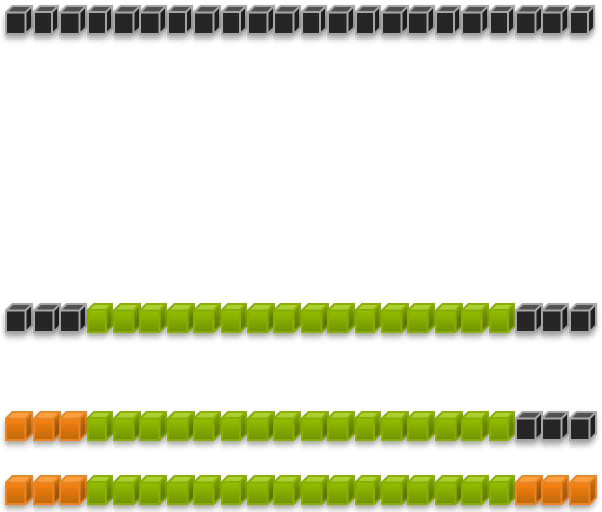
Stencil Kernel

```
__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];
  }
```



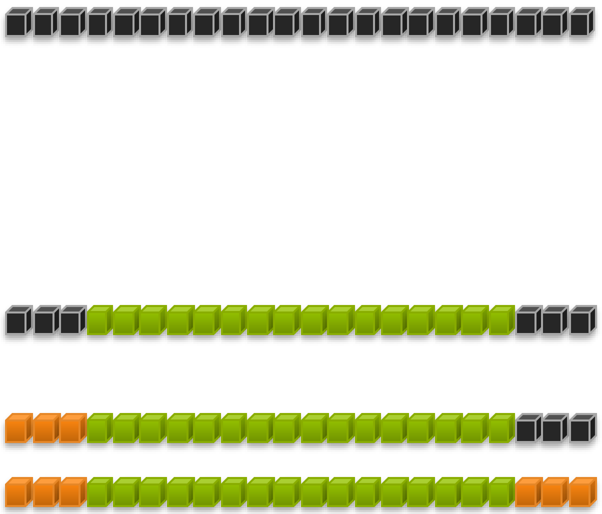
Stencil Kernel

```
__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];
  }
  // 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…

Store at temp[18]

```
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] 

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

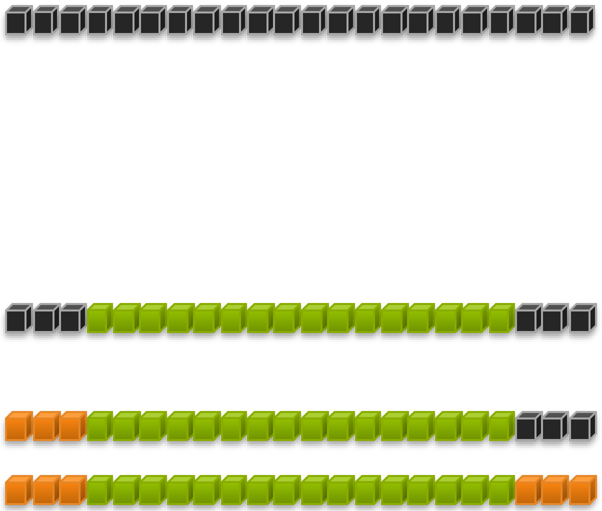
```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // 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];
  }
```

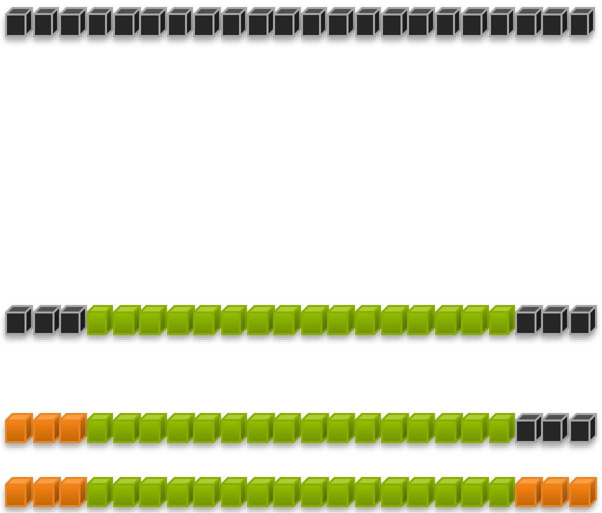
Correct Stencil Kernel

```
__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 __synchthreads();`

- 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

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 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

- The following presentations concentrate on Fermi devices
  - Compute Capability >= 2.0

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)	Yes				
Synchronization functions (Synchronization Functions)	Yes				
Surface functions (Surface Functions)	Yes				
Unified Memory Programming (Unified Memory Programming)	Yes				
Dynamic Parallelism (CUDA Dynamic Parallelism)	Yes				
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

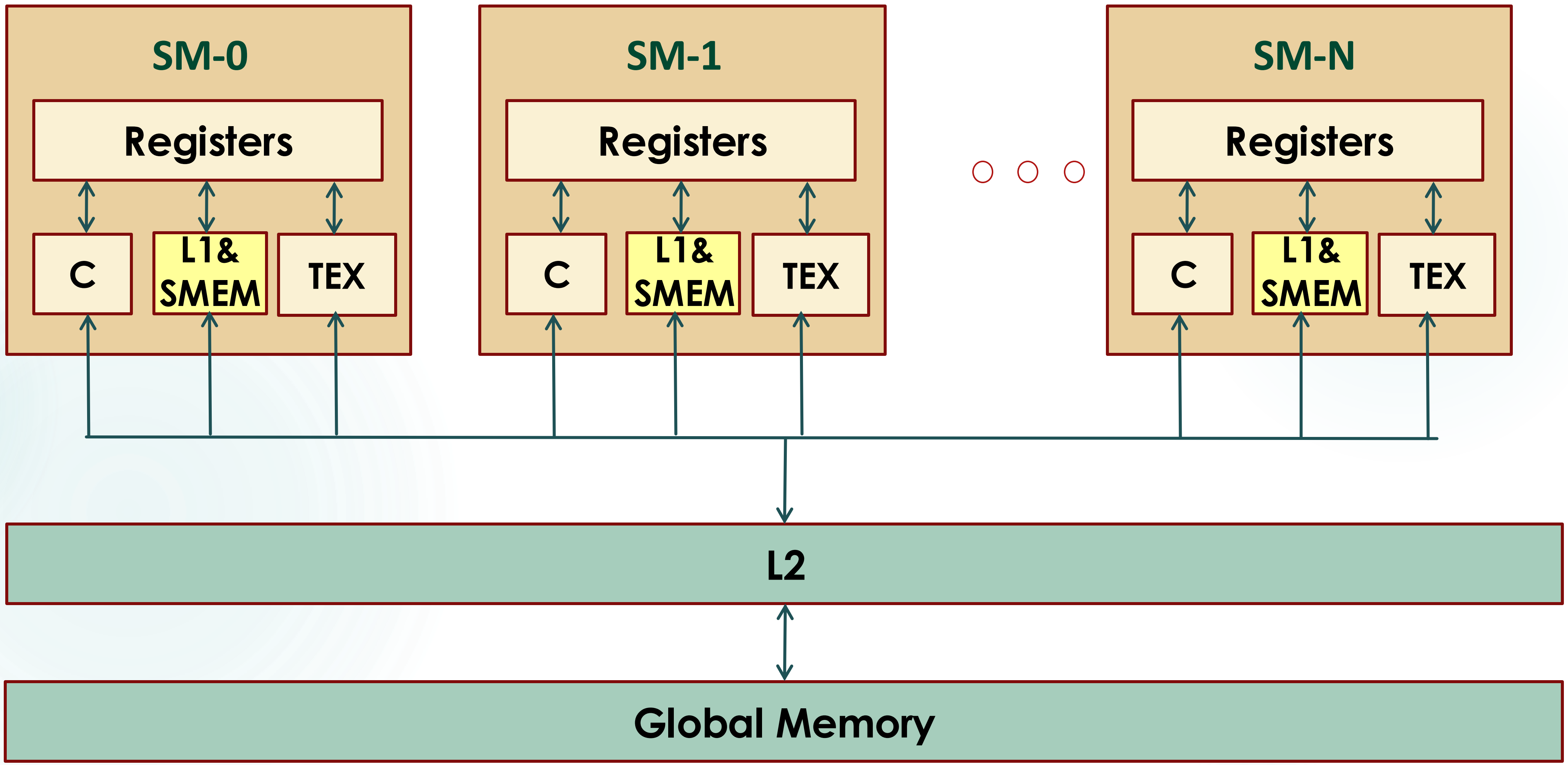
Note that the KB and K units used in the following table correspond to 1024 bytes (i.e., a KiB) and 1024 respectively.

Table 15. Technical Specifications per Compute Capability

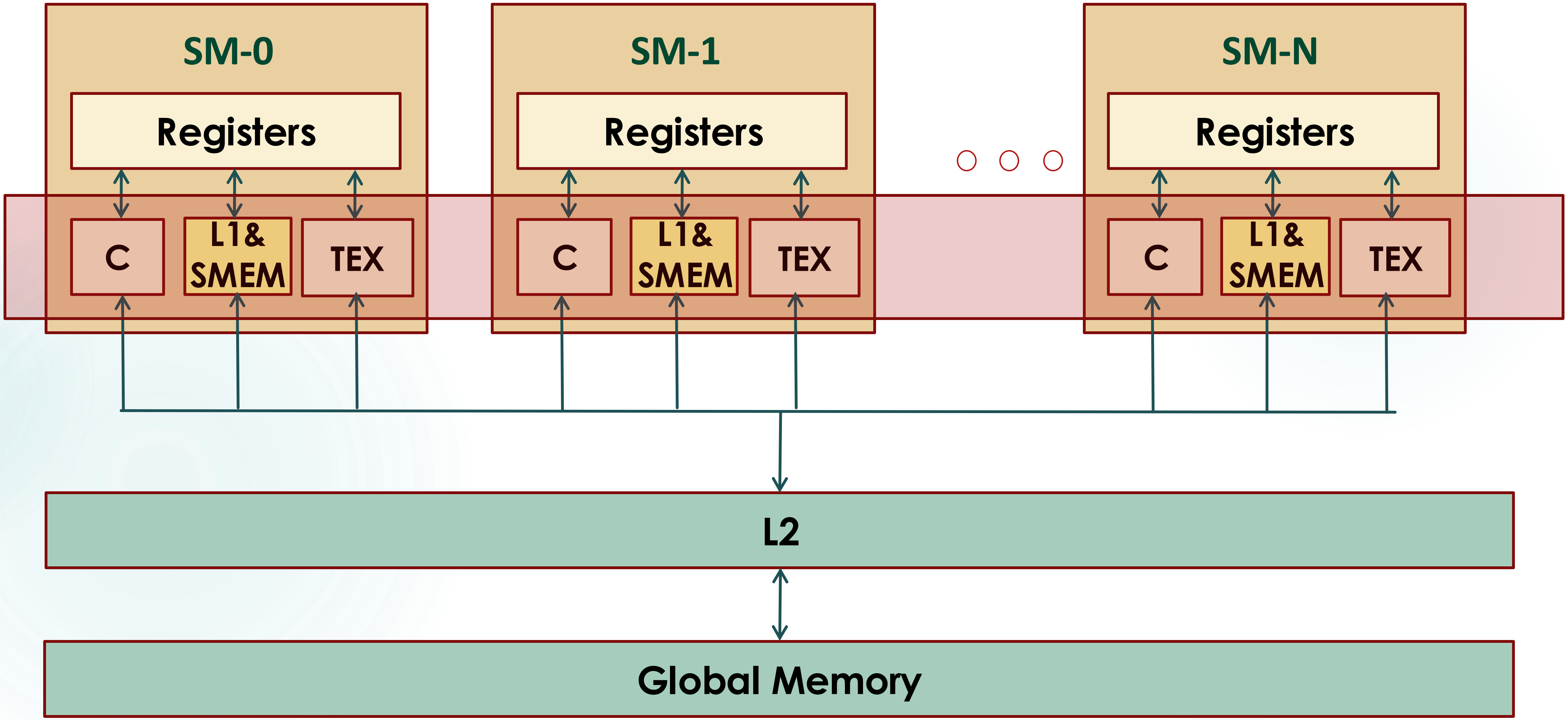
Technical Specifications	Compute Capability							
	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
Maximum number of resident grids per device (Concurrent Kernel Execution)	32				16	128	32	16
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$							
Maximum 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							

- Th
  - Compute Capability >= 2.0

GPU Memory Hierarchy



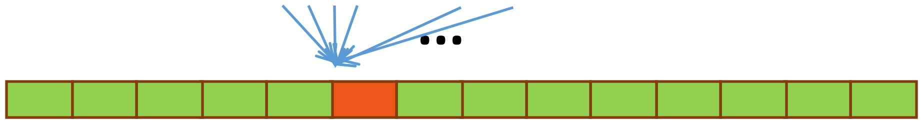
GPU Memory Hierarchy



Constant Cache

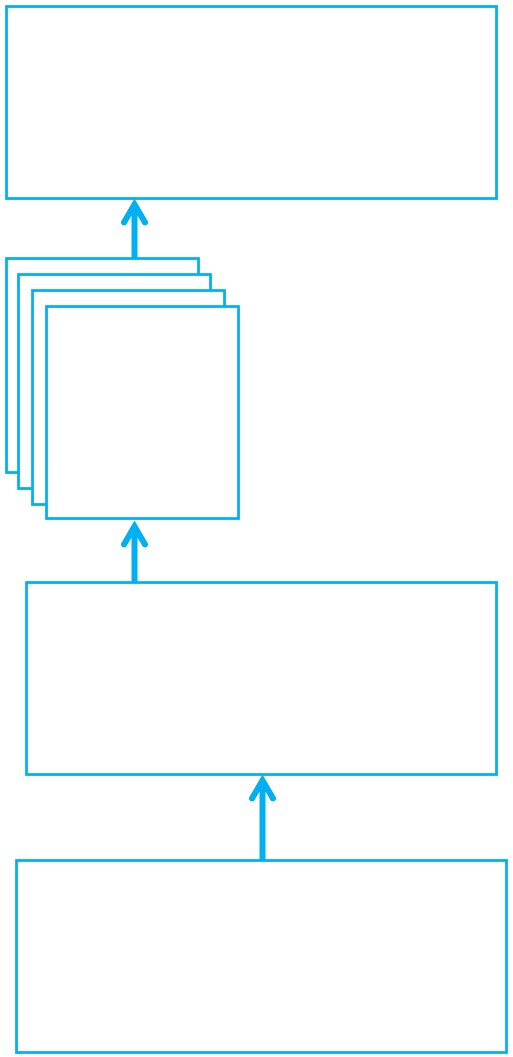
- Global variables marked by __constant__
  - constant and can't be changed in device.
- Will be cached by Constant Cache
- Located in global memory
- Good for threads that access the same address

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



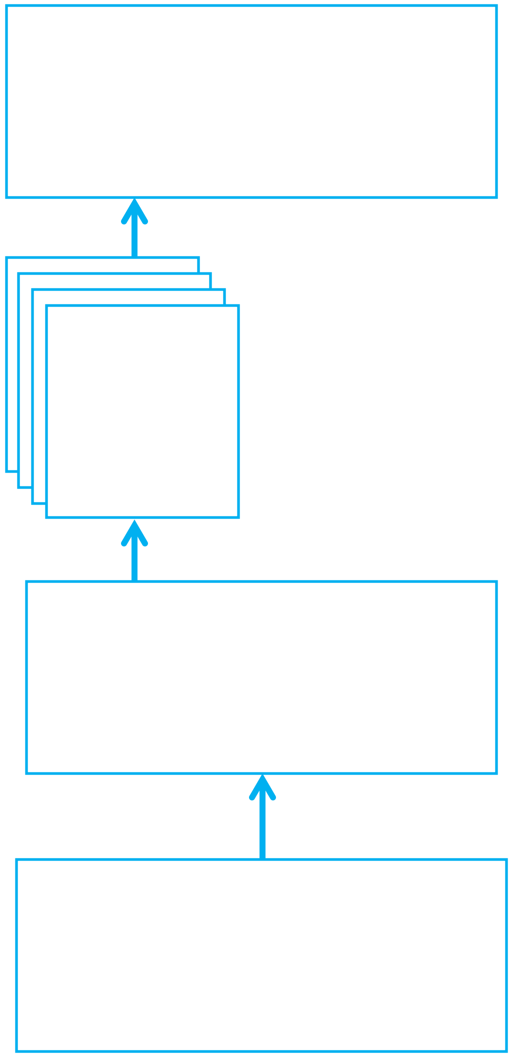
Memory addresses

Texture Cache



Texture Cache

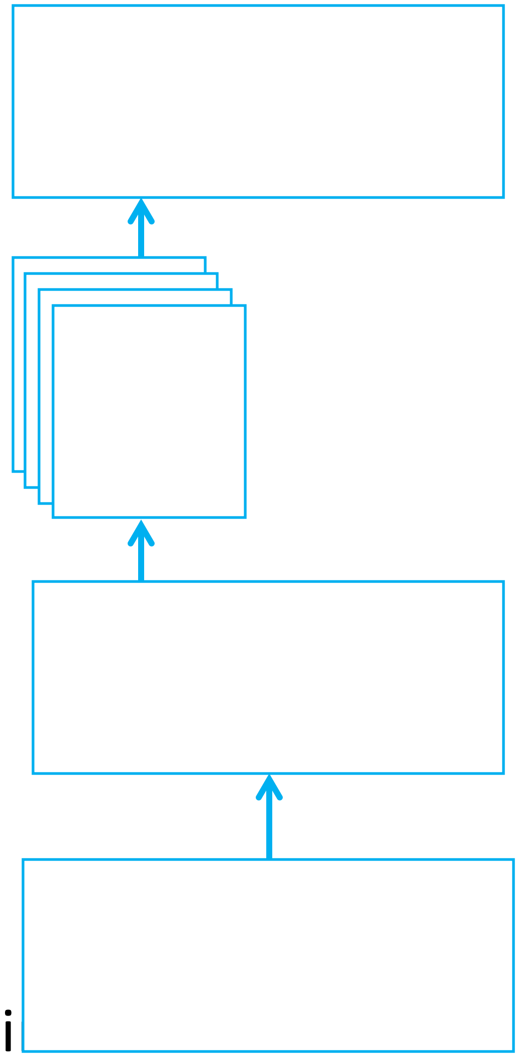
- Save Data as Texture :
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache



Texture Cache

- Save Data as Texture :
  - Provides hardware accelerated filtered sampling of data (1D, 2D, 3D)
  - Read-only data cache holds fetched samples
  - Backed up by the L2 cache

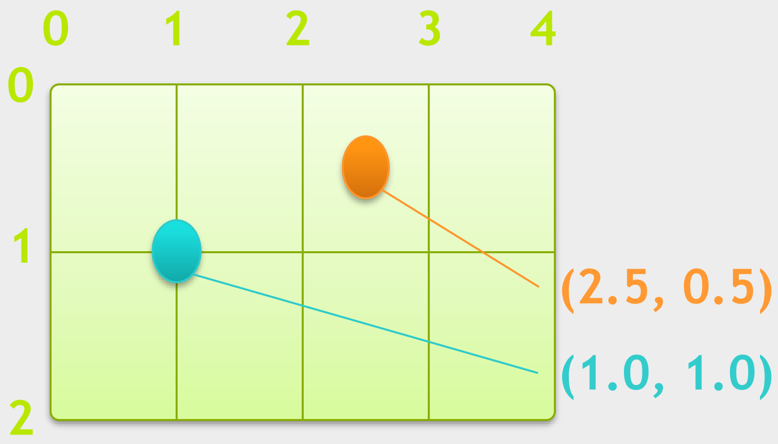
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
  - What if your problem takes a large number of read-only poi input?



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 poi input?

How many threads/blocks should I use?

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

How many threads/blocks should I use?

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

How many threads/blocks should I use?

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

- Usually things are correct if grid*block dims >= input size
- Getting good performance is another matter

Internals

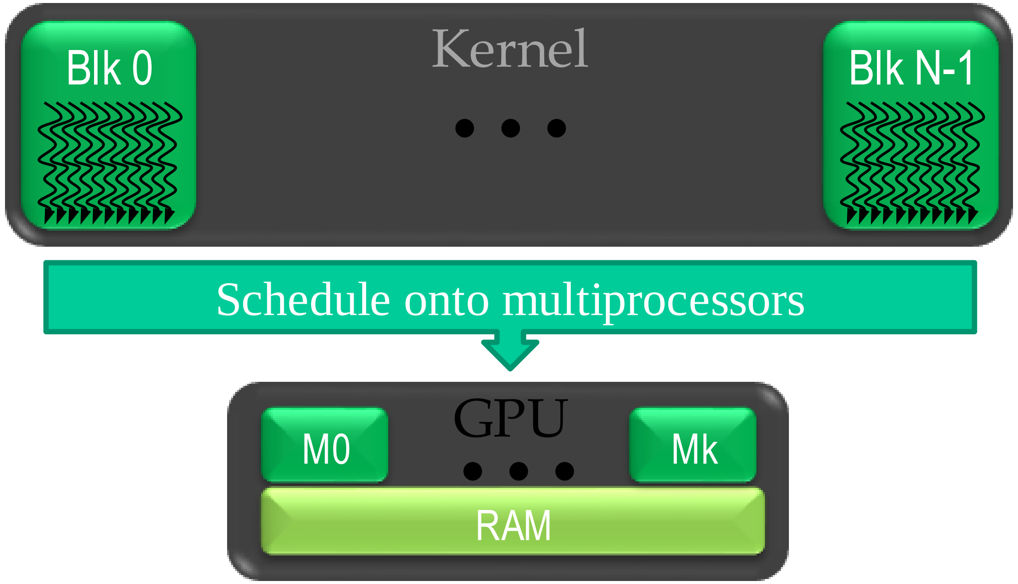
```
__host__
Void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
 vecAddKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```

Internals

```
__host__
Void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
 vecAddKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

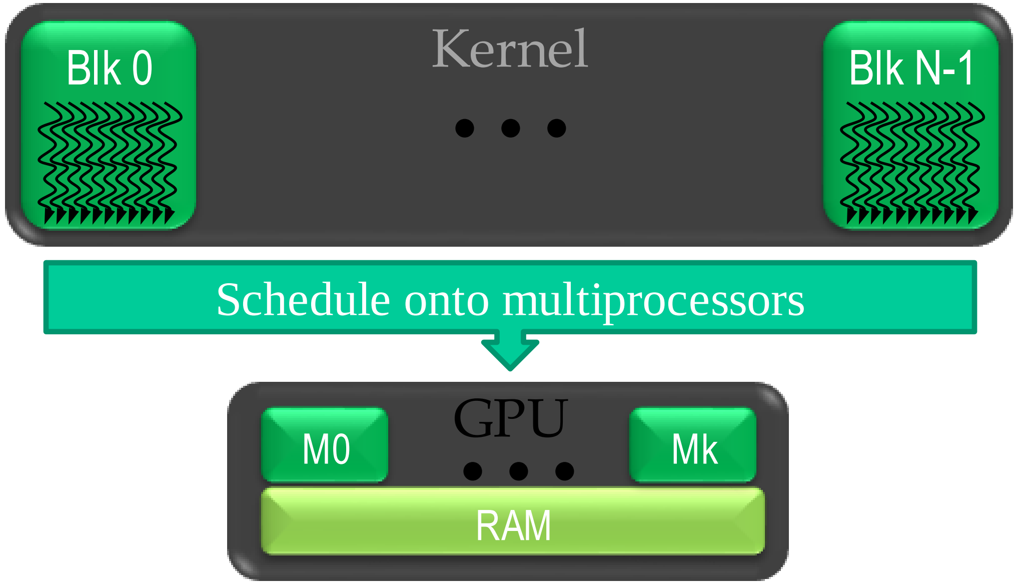
```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



Internals

```
__host__
Void vecAdd()
{
  dim3 DimGrid = (ceil(n/256,1,1);
  dim3 DimBlock = (256,1,1);
 vecAddKernel<<<DGrid,DBlock>>>(A_d,B_d,C_d,n);
}
```

```
__global__
void vecAddKernel(float *A_d,
     float *B_d, float *C_d, int n)
{
   int i = blockIdx.x * blockDim.x
             + threadIdx.x;

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



How are threads scheduled?

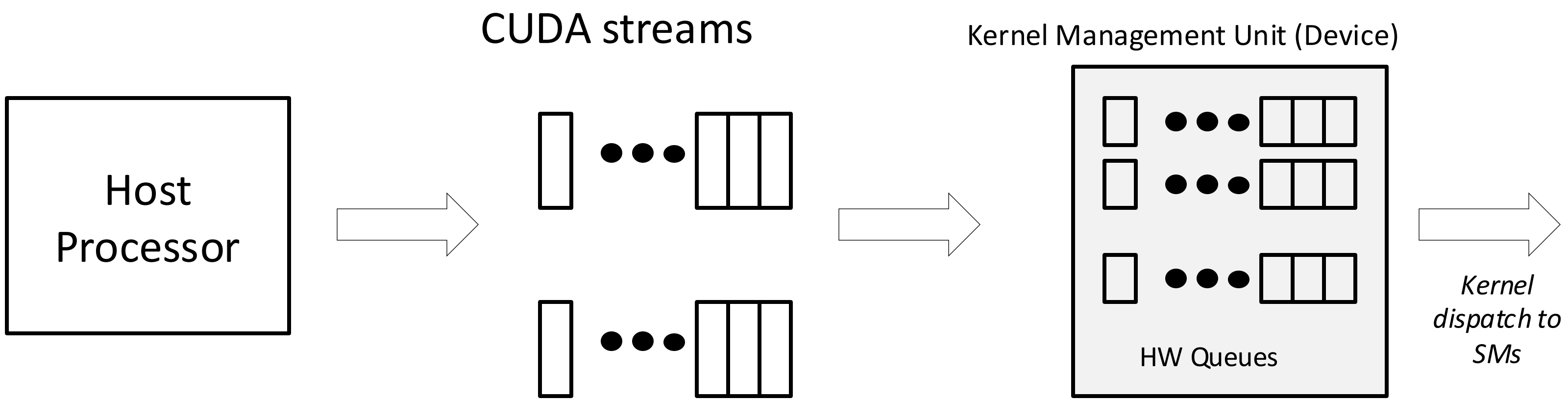
Kernel Launch

Kernel Launch

- Commands by host issued through ***streams***

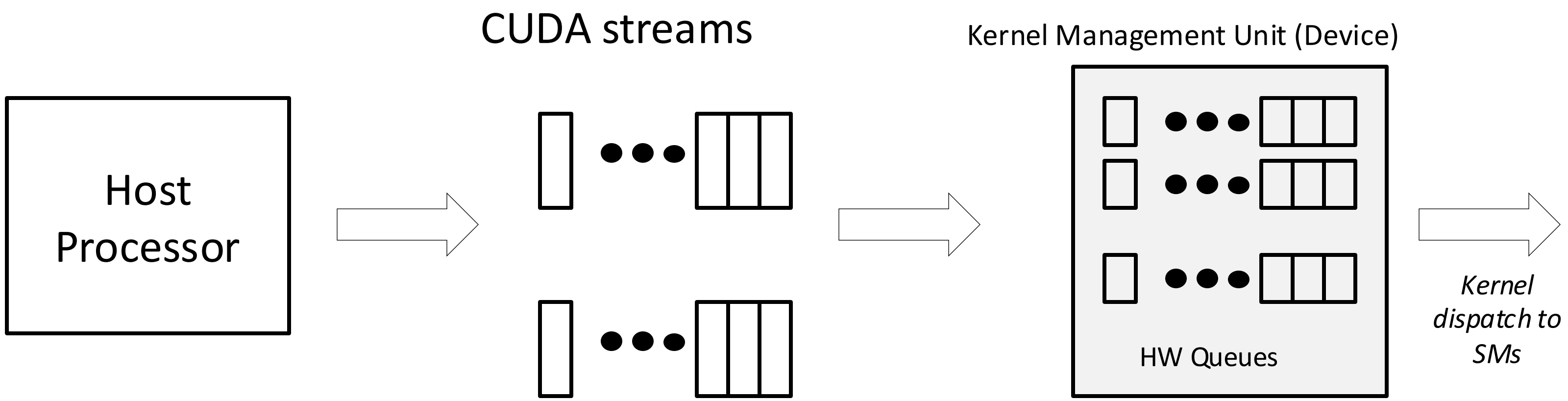
Kernel Launch

- Commands by host issued through ***streams***



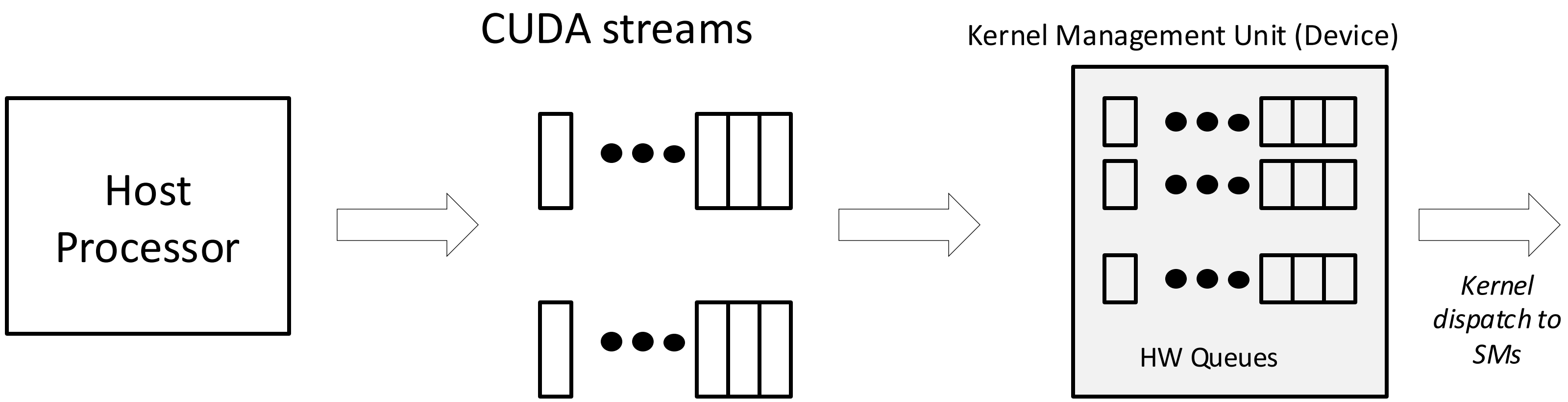
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially



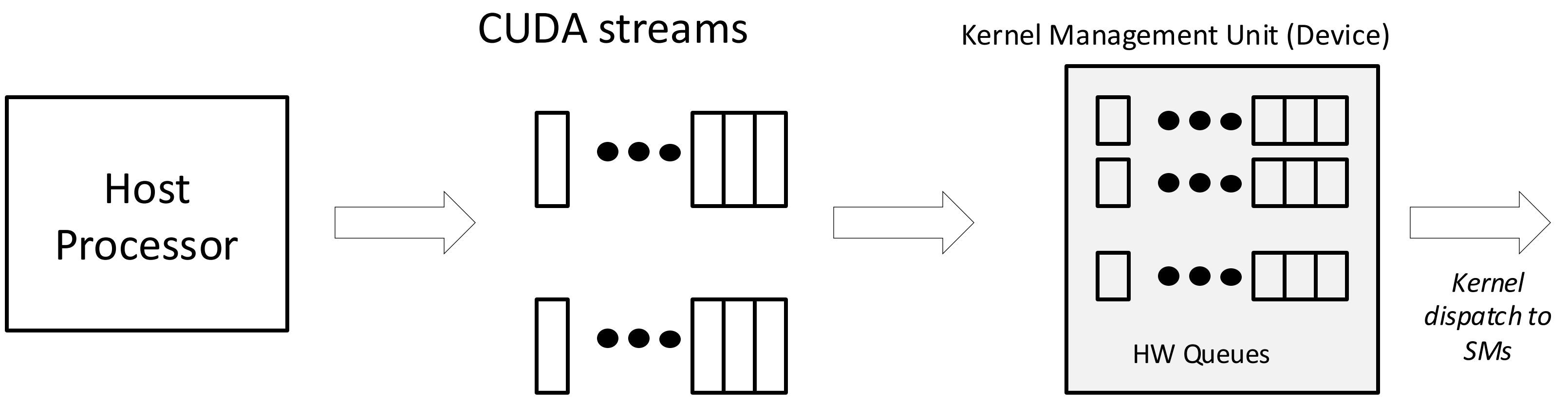
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently



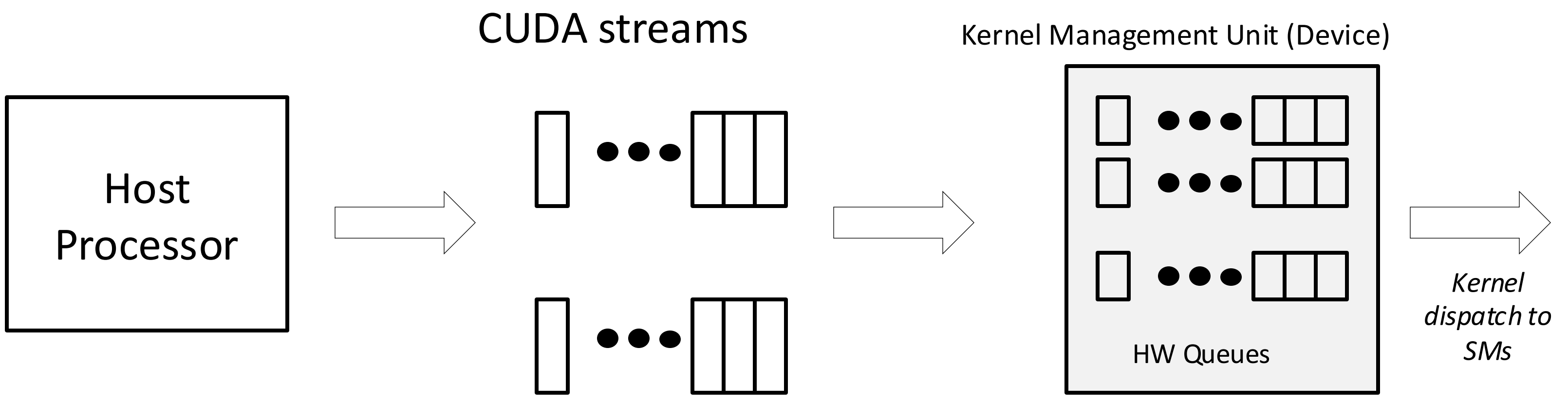
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues



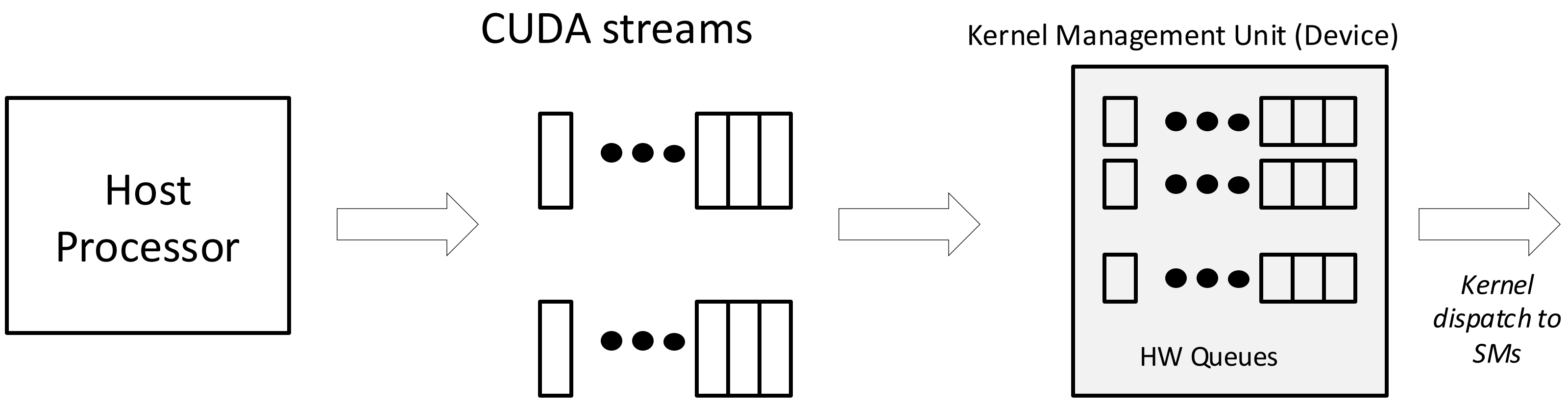
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by “kernel management unit” (KMU)



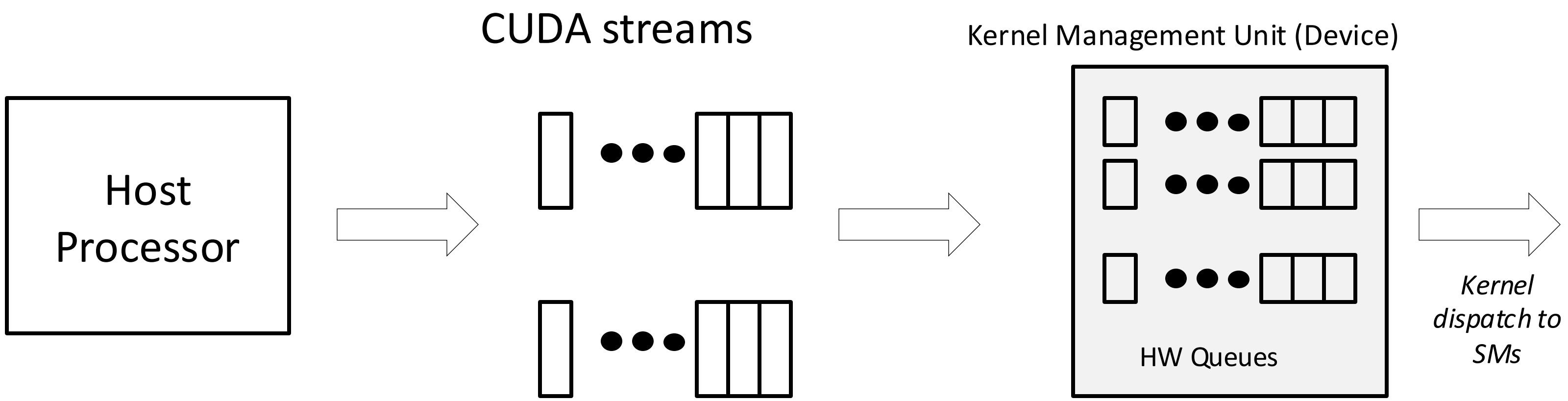
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by "kernel management unit" (KMU)
  - Multiple streams mapped to each queue → serializes some kernels



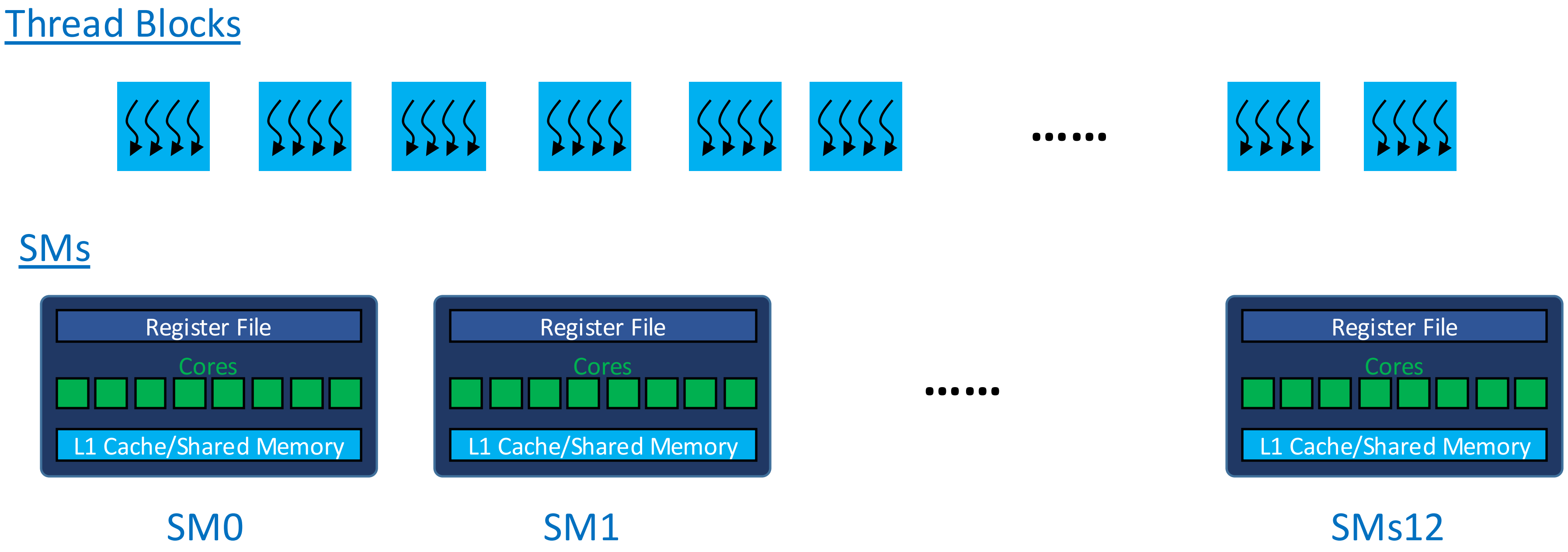
Kernel Launch

- Commands by host issued through ***streams***
  - Kernels in the same stream executed sequentially
  - Kernels in different streams may be executed concurrently
- Streams mapped to GPU HW queues
  - Done by "kernel management unit" (KMU)
  - Multiple streams mapped to each queue → serializes some kernels
- Kernel launch distributes thread blocks to SMs



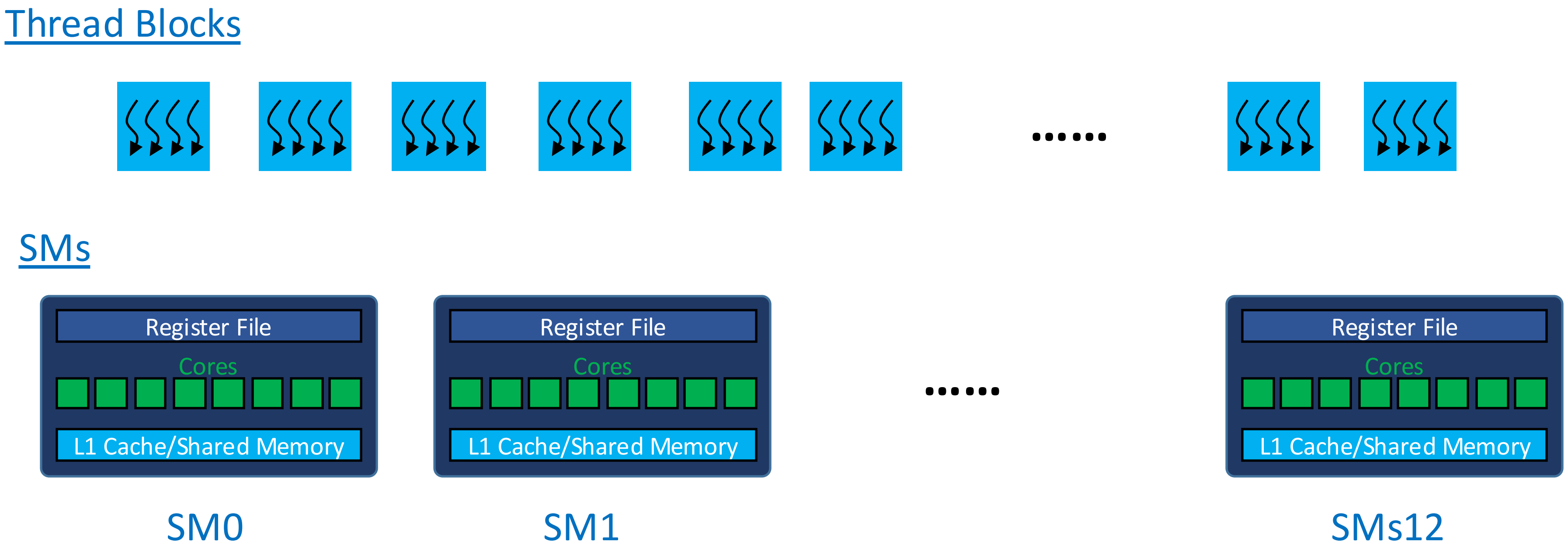
Thread Blocks, Warps, Scheduling

- What's a warp?
- Imagine one TB (threadblock) has 64 threads (2 warps)
  - On K20c GK110 GPU: 13 SMs, max 64 warps/SM, max 32 concurrent kernels



Thread Blocks, Warps, Scheduling

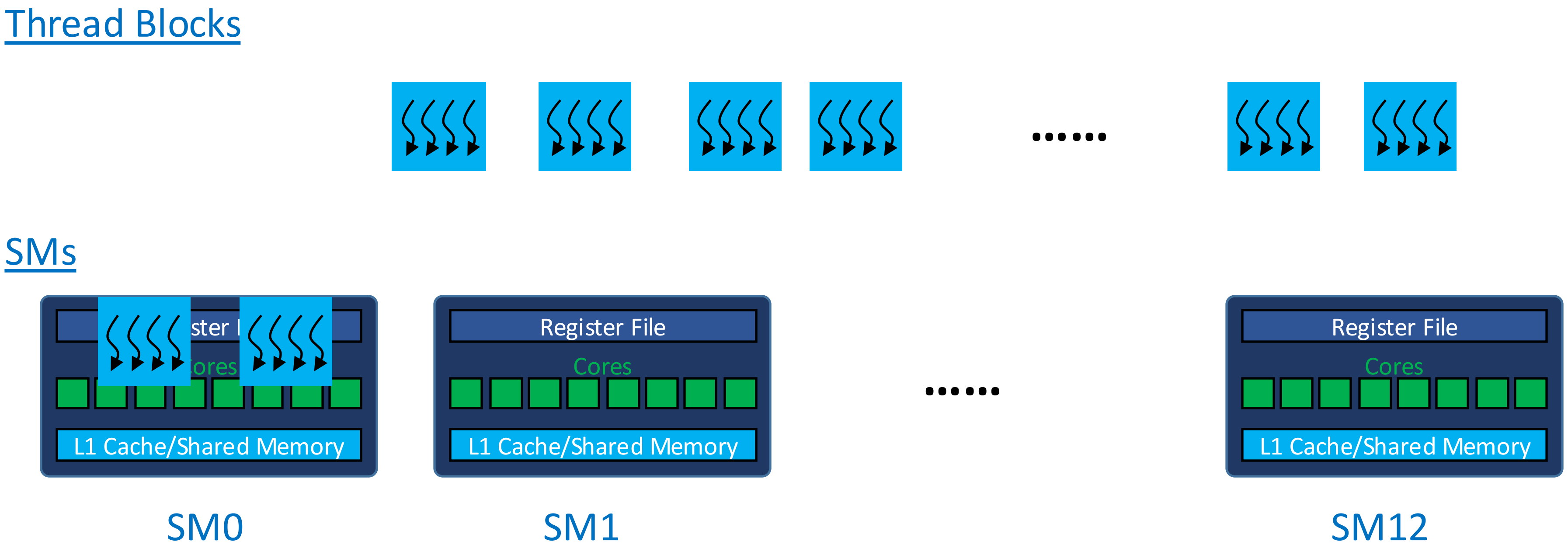
- What's a warp?
- Imagine one TB (threadblock) has 64 threads (2 warps)
  - On K20c GK110 GPU: 13 SMs, max 64 warps/SM, max 32 concurrent kernels



- SMs split blocks into warps
- Unit of HW scheduling for SM
- 32 threads each

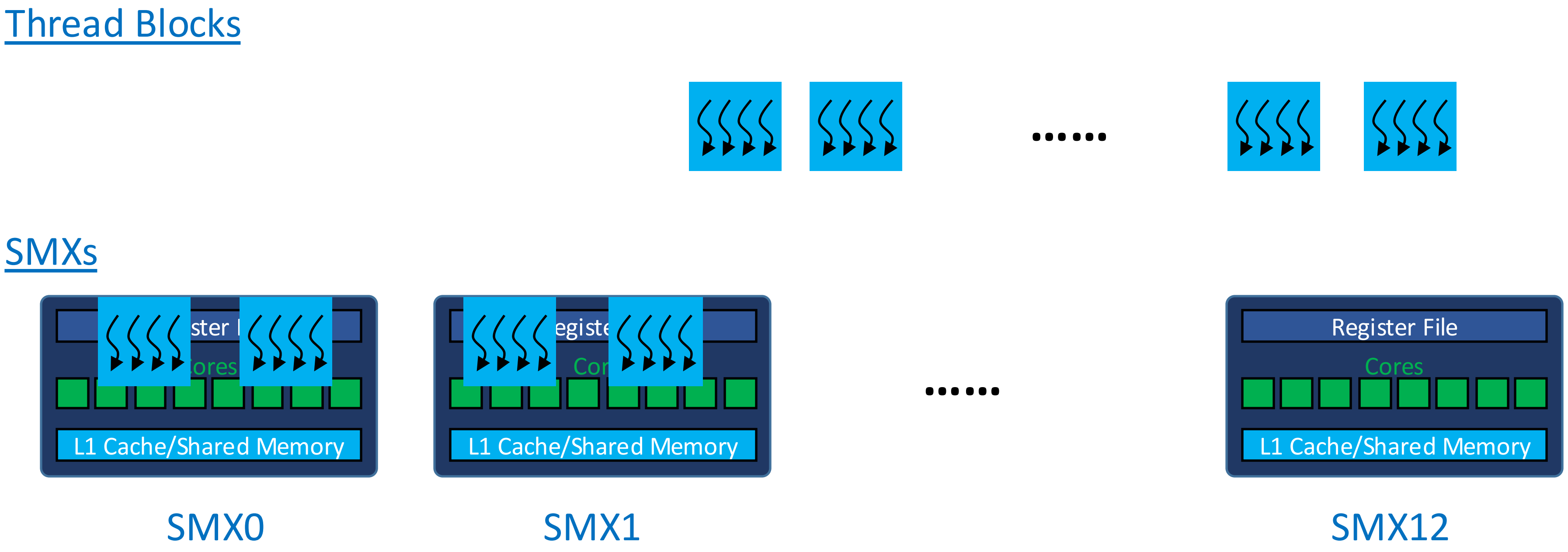
TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
  - On K20c GK110 GPU: 13 SMX, max 64 warps/SM, max 32 concurrent kernels



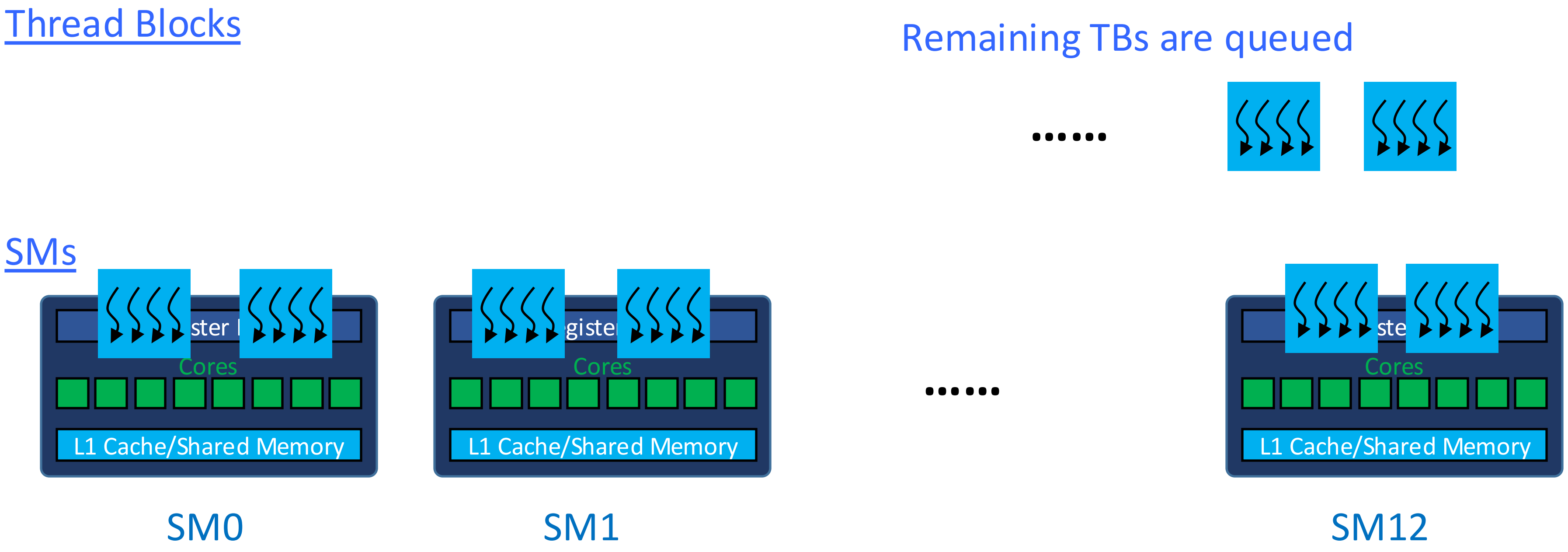
TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
- On K20c GK110 GPU: 13 SMX, max 64 warps/SMX, max 32 concurrent kernels



TBs, Warps, & Utilization

- One TB has 64 threads or 2 warps
  - On K20c GK110 GPU: 13 SMX, max 64 warps/SM, max 32 concurrent kernels



NVIDIA GK110 (Kepler)

Thread Block Scheduler

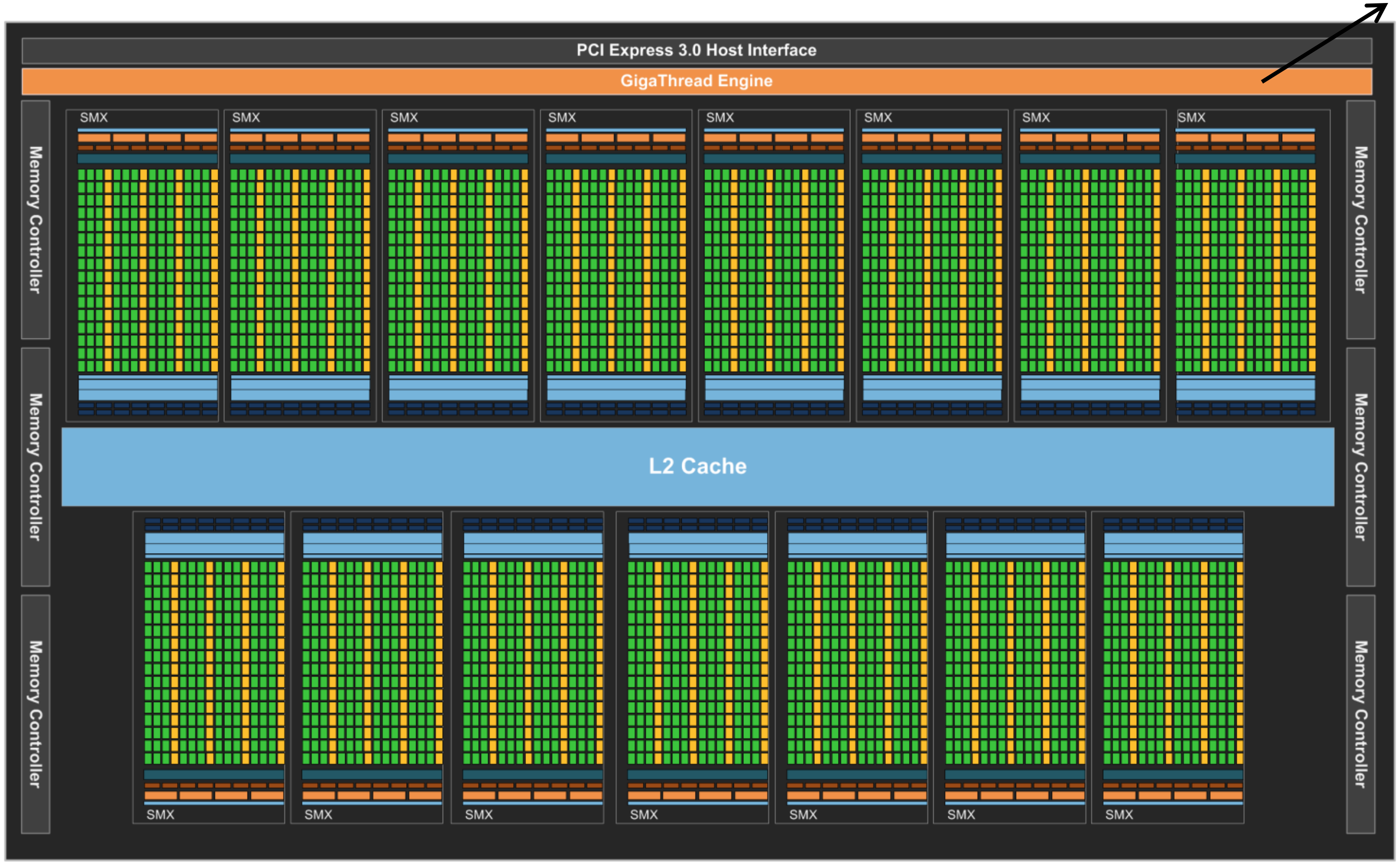


Image from http://mandetech.com/2012/05/20/nvidia-new-gpu-and-visualization/

NVIDIA GK110 (Kepler)

Thread Block Scheduler

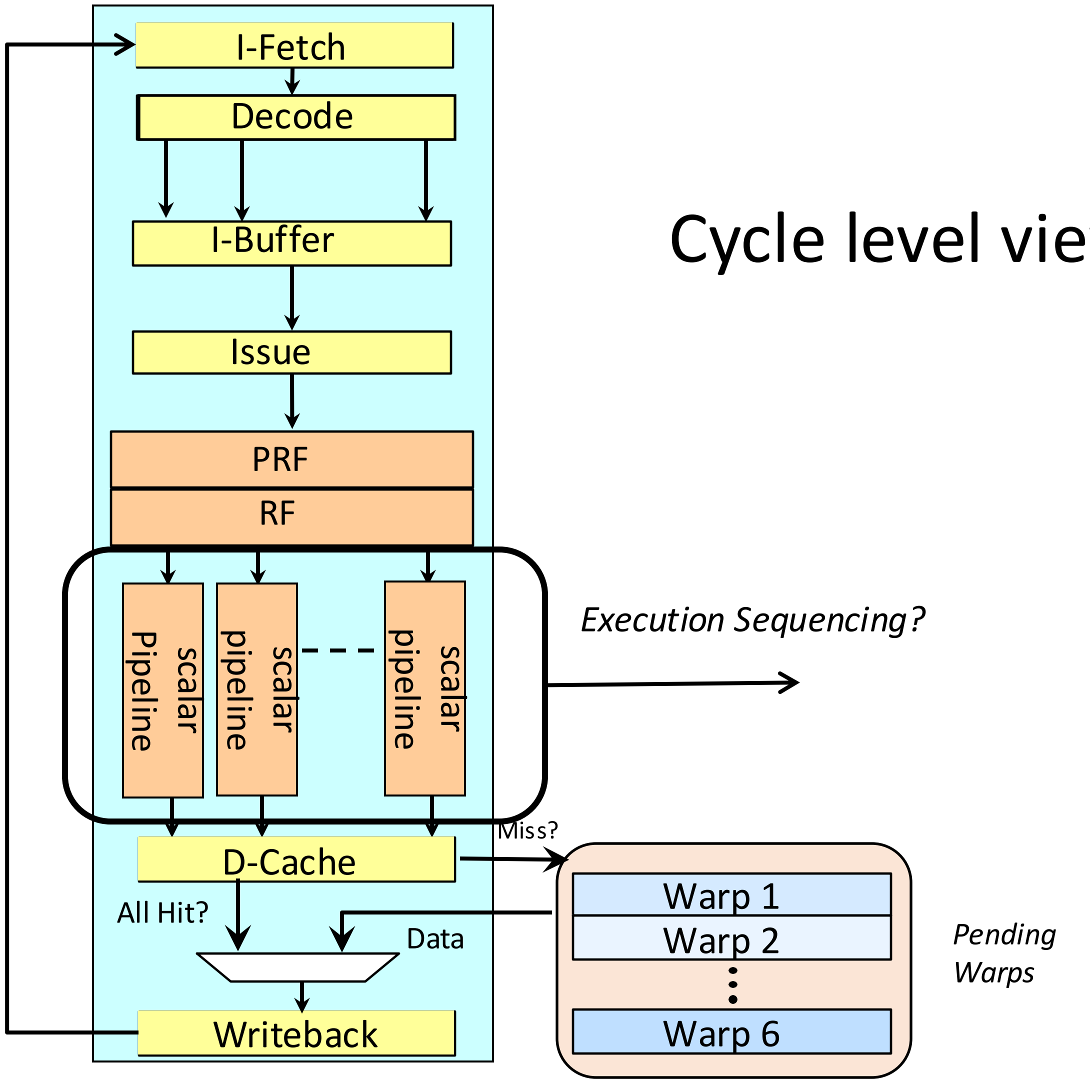
Multiple Warp Schedulers

64K 32-bit registers

192 cores – 6 clusters of 32 cores each

Image from http://mar

Execution Sequencing in a Single SM



Cycle level view: GPU kernel execution

Example: VectorAdd on GPU

CUDA:

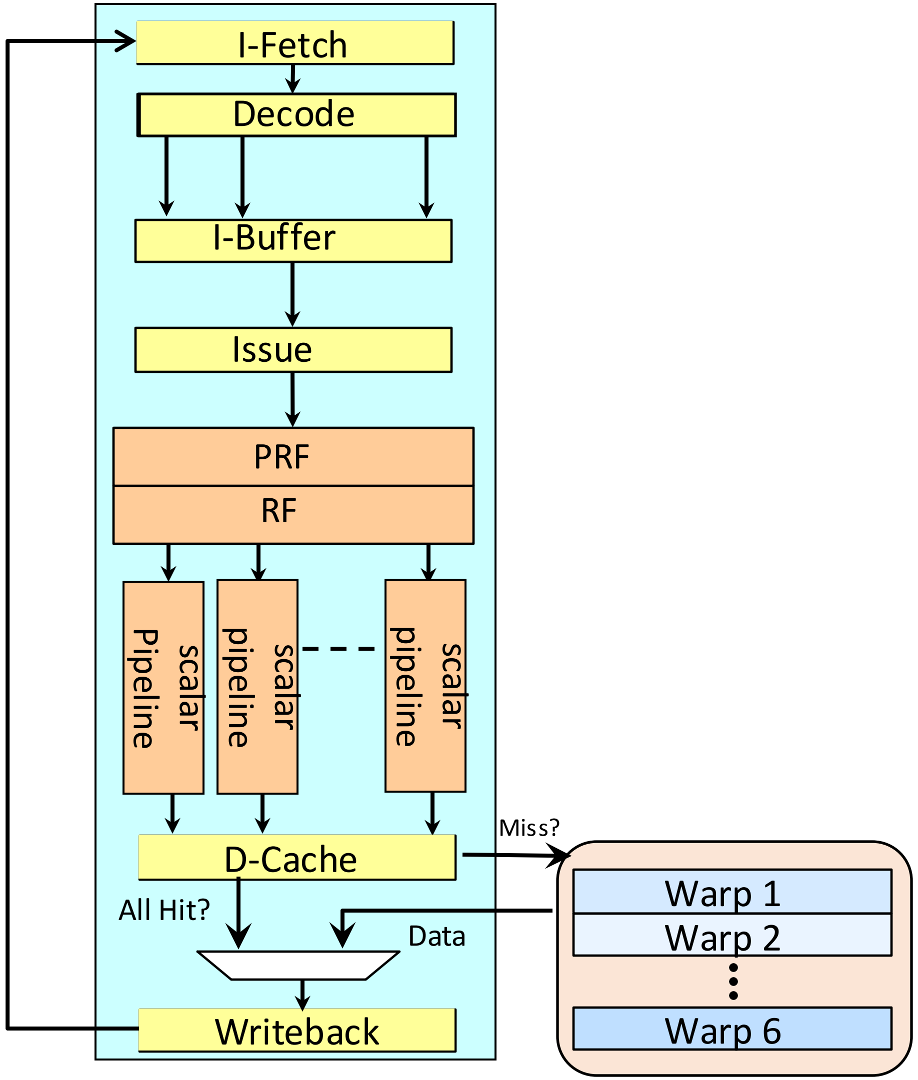
```
__global__  vector_add
  ( float *a, float *b, float *c, int N) {
int index = blockIdx.x *  blockDim.x +
threadIdx.x;

if (index < N)
   c[index] = a[index]+b[index];
}
```

PTX (Assembly):

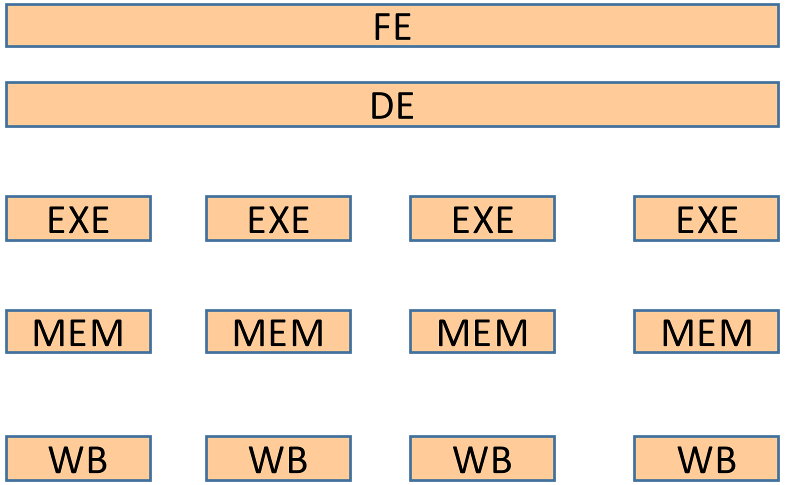
```
setp.lt.s32 %p, %r5, %rd4;  //r5 = index, rd4 = N
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];  //r6 = &a[index]
ld.global.f32 %f2, [%r7];  //r7 = &b[index]
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3; //r8  =  &c[index]

L2:
ret;
```

Example: VectorAdd on GPU

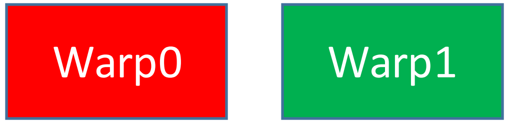
- N=8, 8 Threads, 1 block, warp size = 4
- 1 SM, 4 Cores
- Pipeline:
  - Fetch:
    - One instruction from each warp
    - Round-robin through all warps
  - Execution:
    - In-order execution within warps
  - With proper data forwarding
  - 1 Cycle each stage
- How many warps?



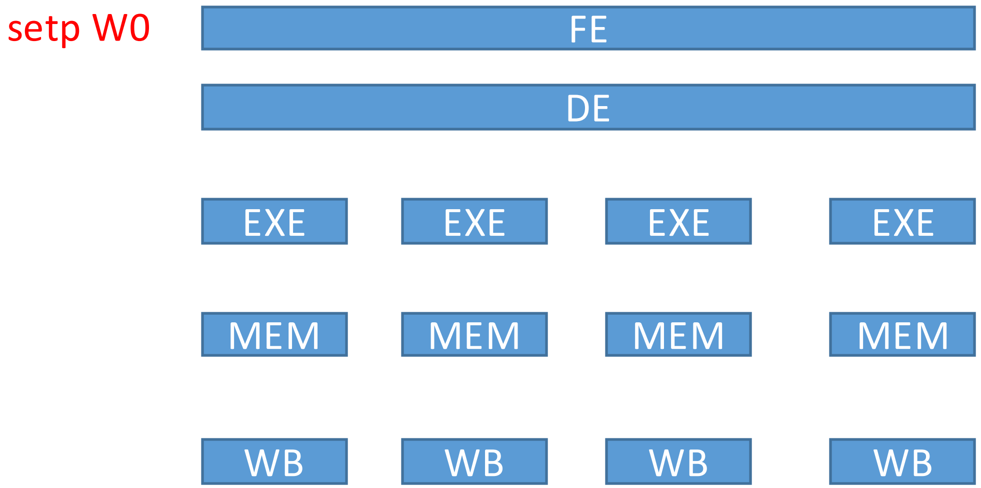
Execution Sequence



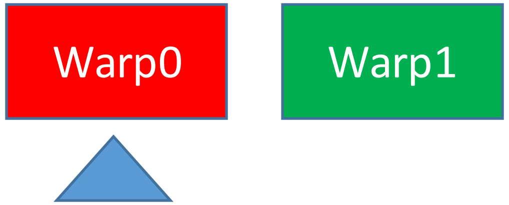

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



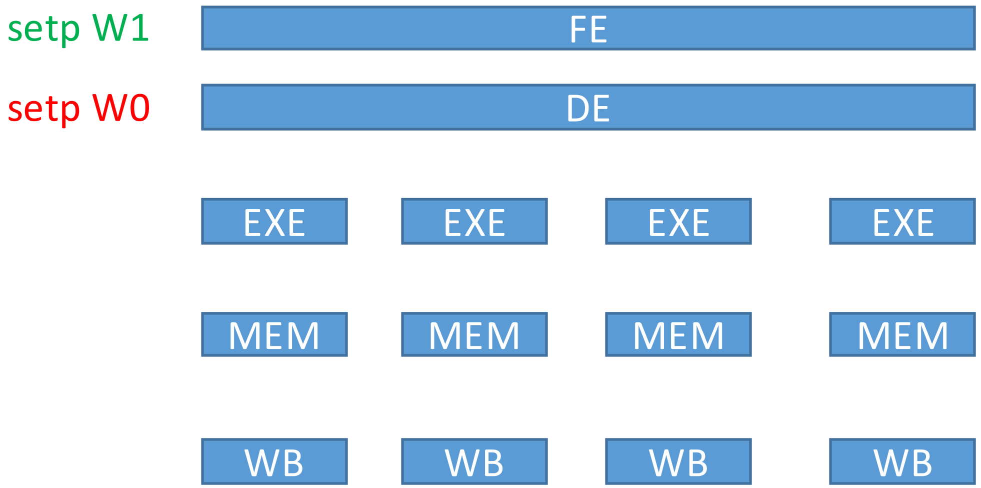
Execution Sequence (cont.)



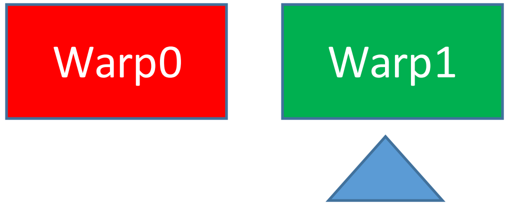

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



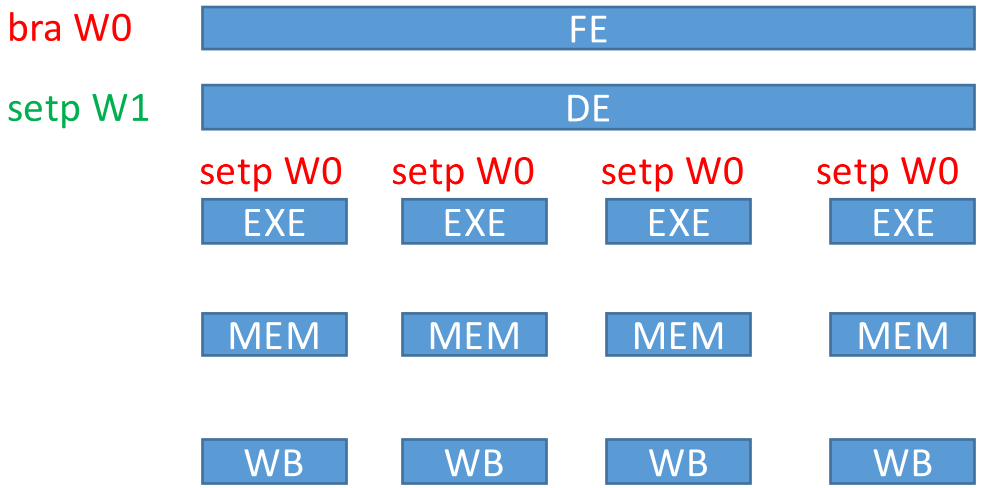
Execution Sequence (cont.)



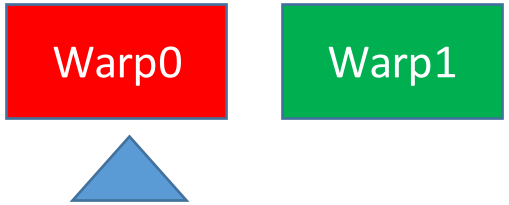

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



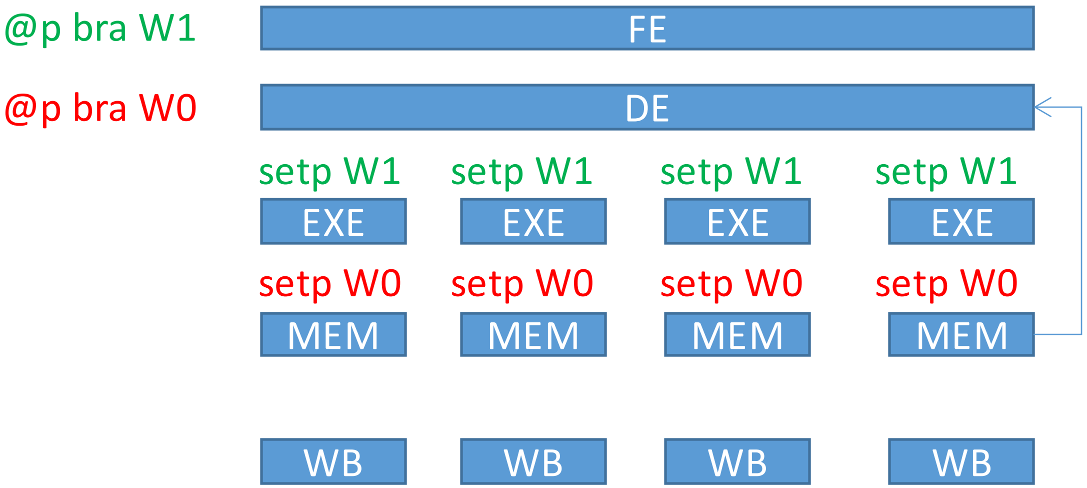
Execution Sequence (cont.)



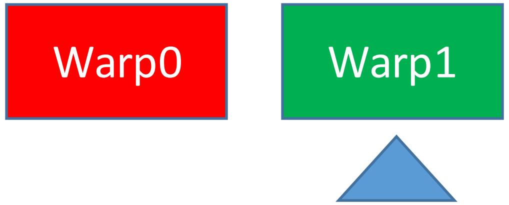

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



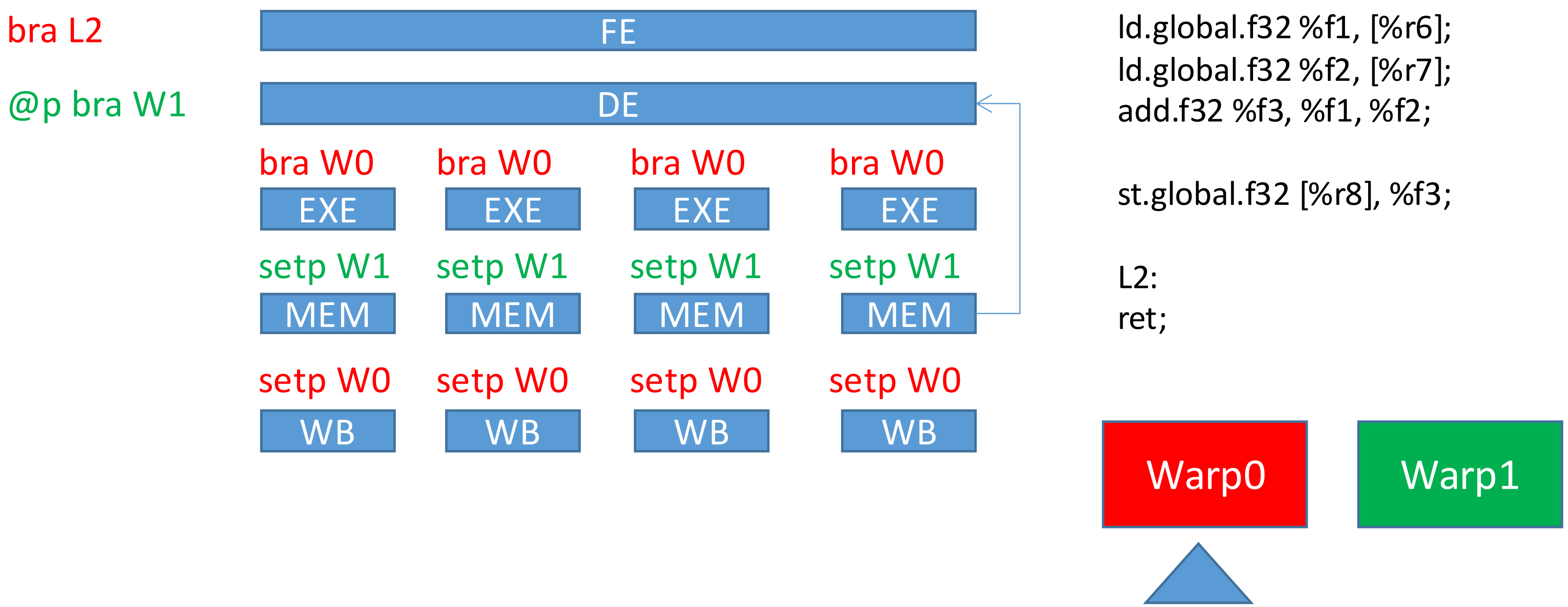
Execution Sequence (cont.)




```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

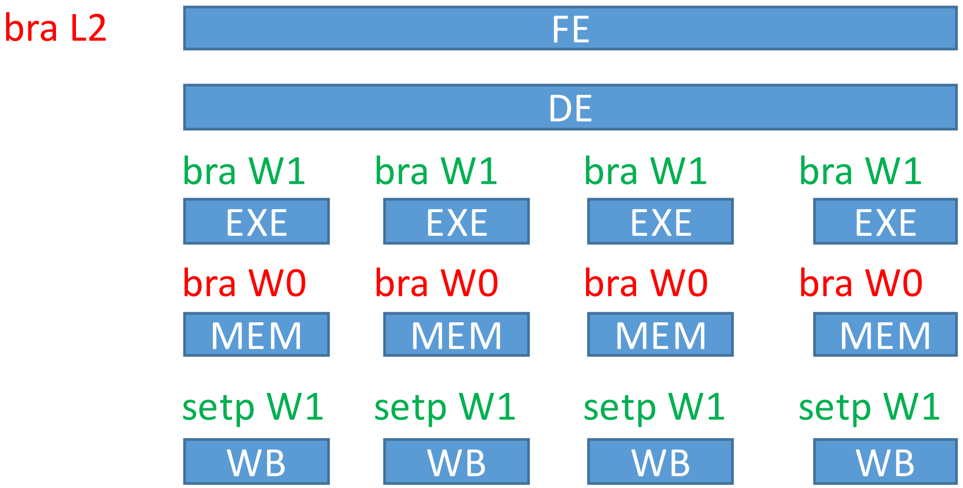


Execution Sequence (cont.)

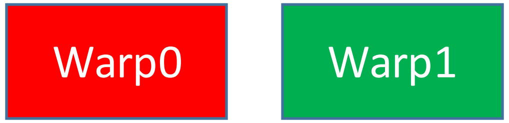



```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

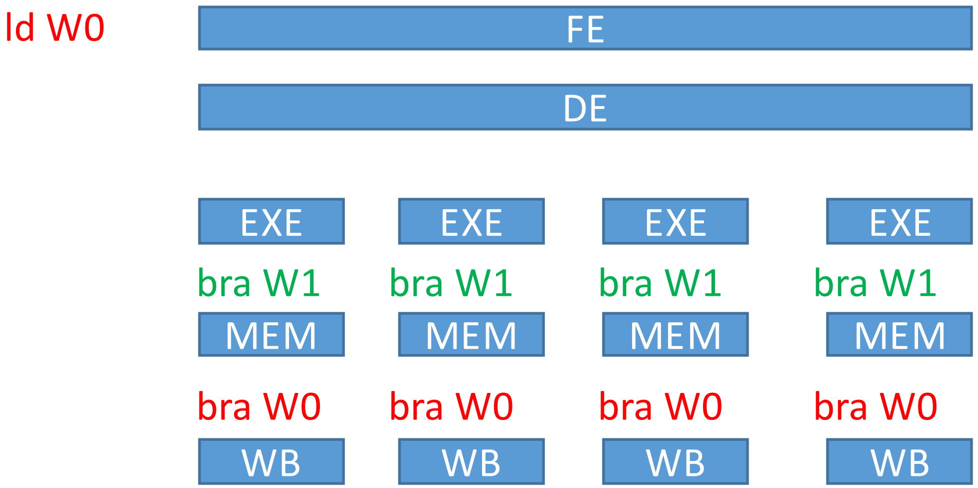
Execution Sequence (cont.)




```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

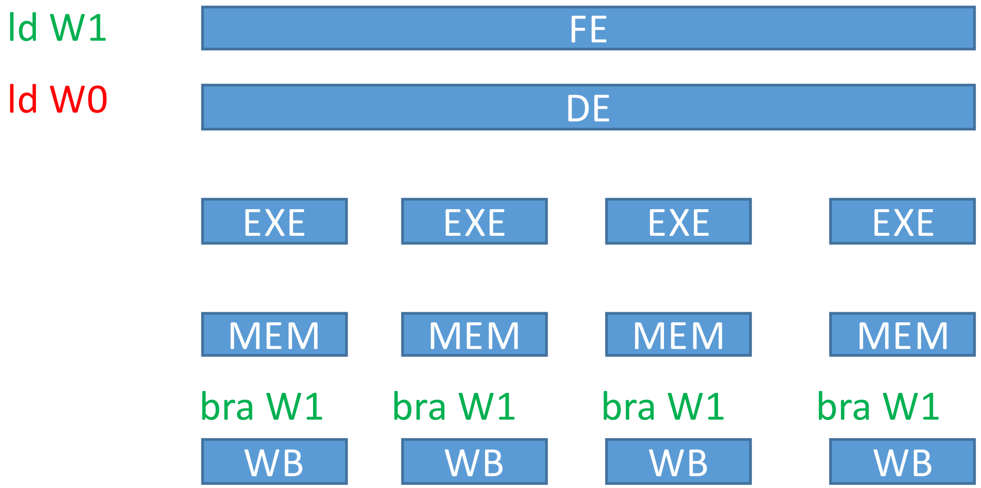


Execution Sequence (cont.)

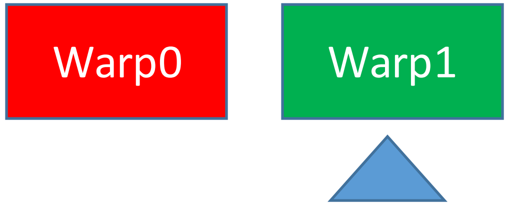



```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

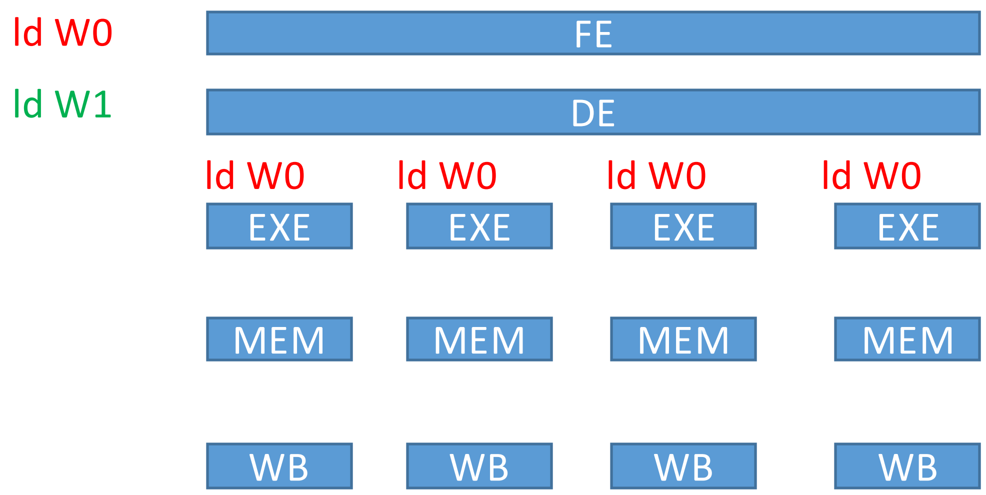
Execution Sequence (cont.)



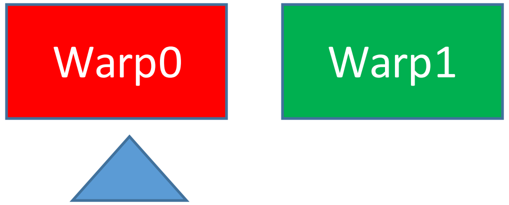

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



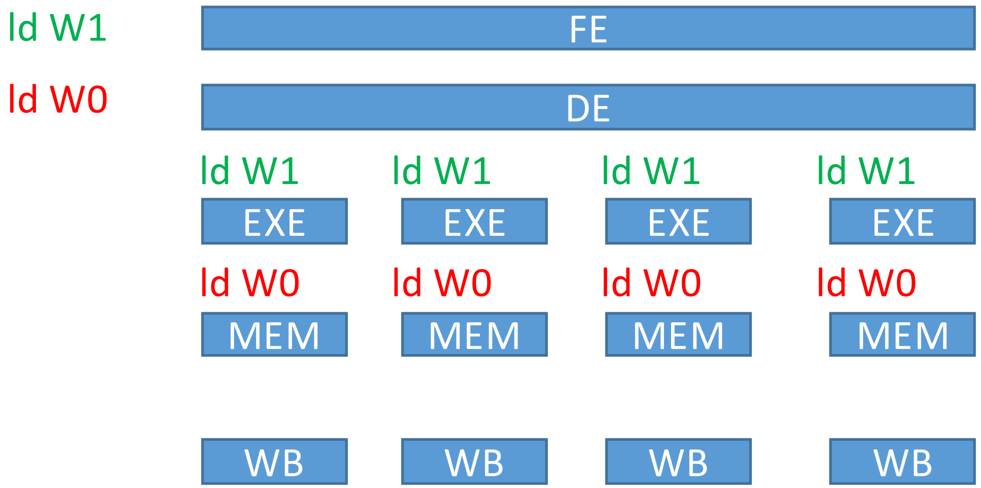
Execution Sequence (cont.)



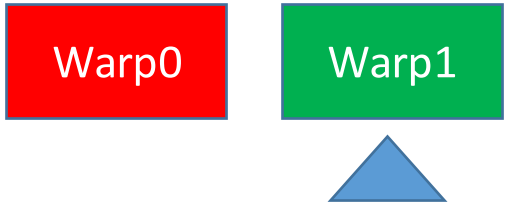

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



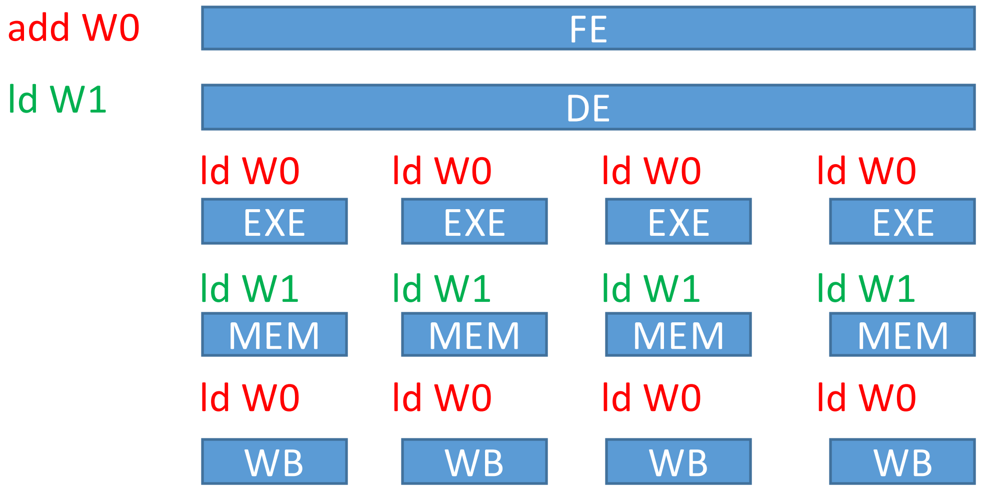
Execution Sequence (cont.)



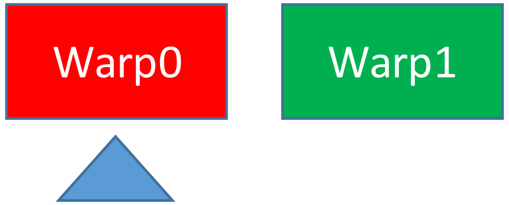

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



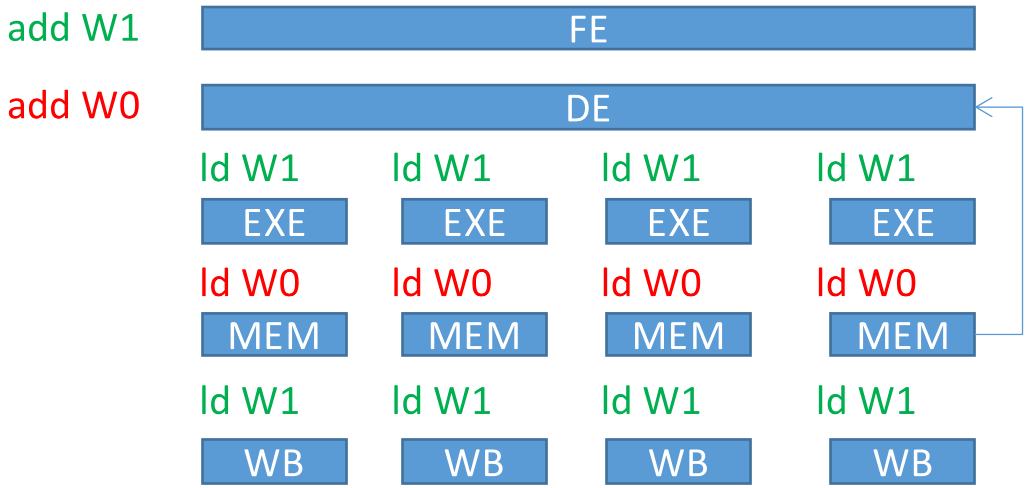
Execution Sequence (cont.)



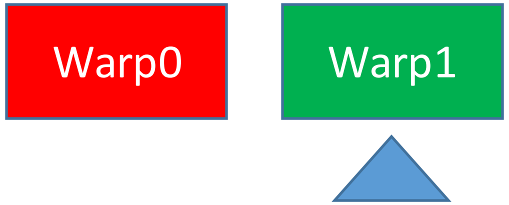

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



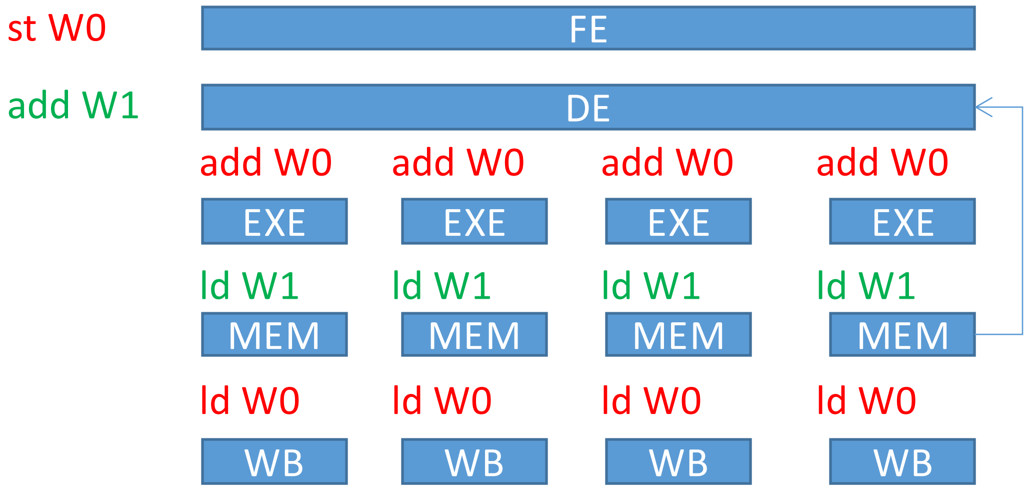
Execution Sequence (cont.)



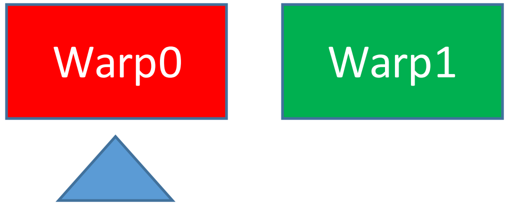

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



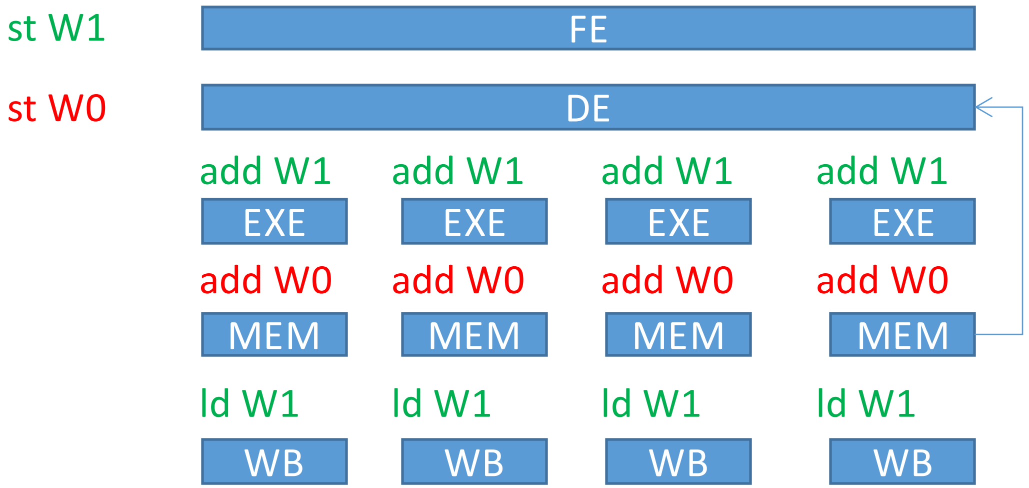
Execution Sequence (cont.)



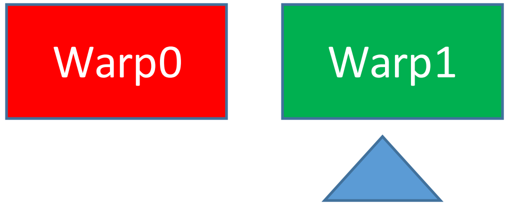

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



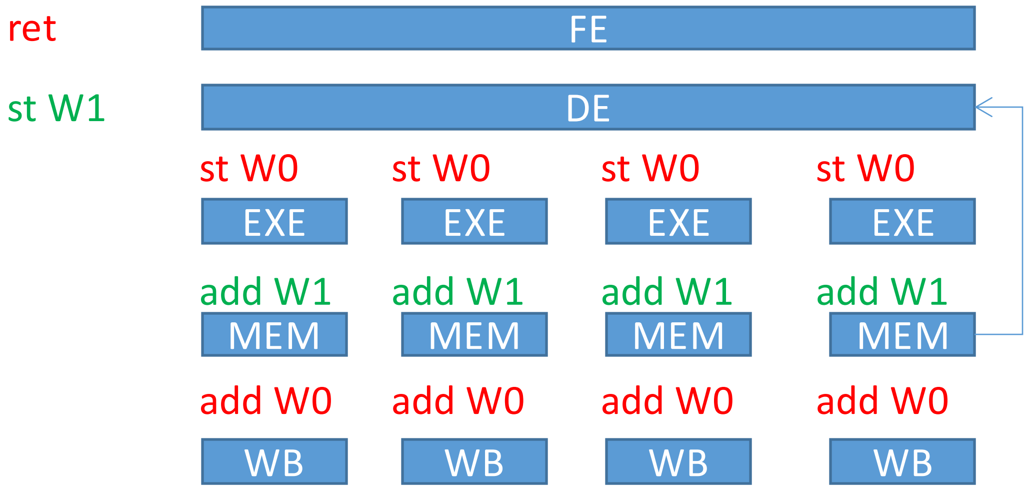
Execution Sequence (cont.)




```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

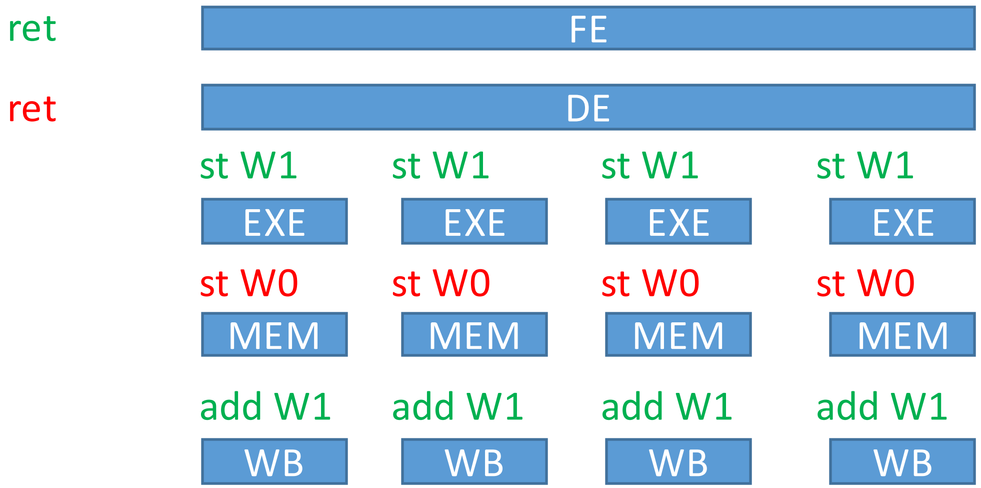


Execution Sequence (cont.)



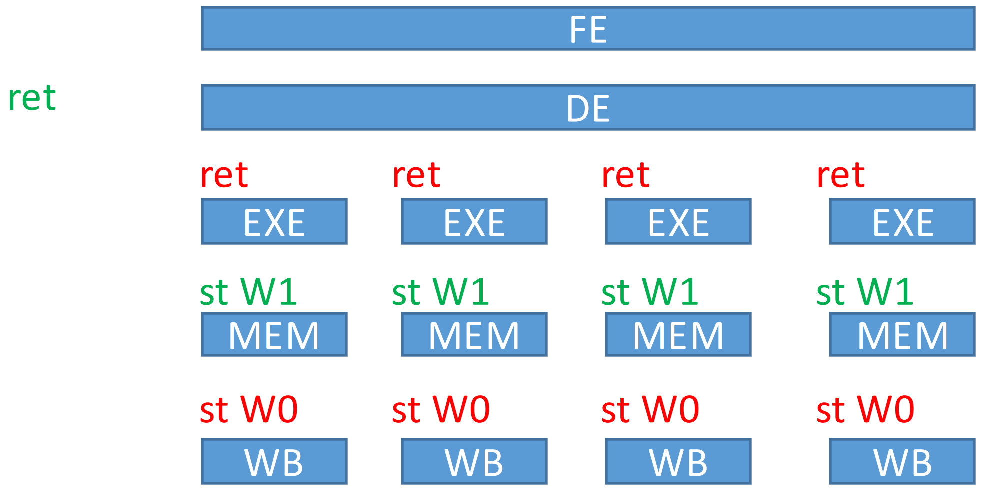

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

Execution Sequence (cont.)

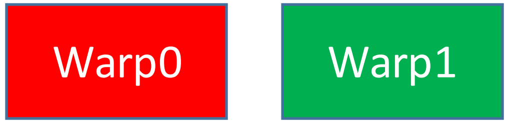



```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

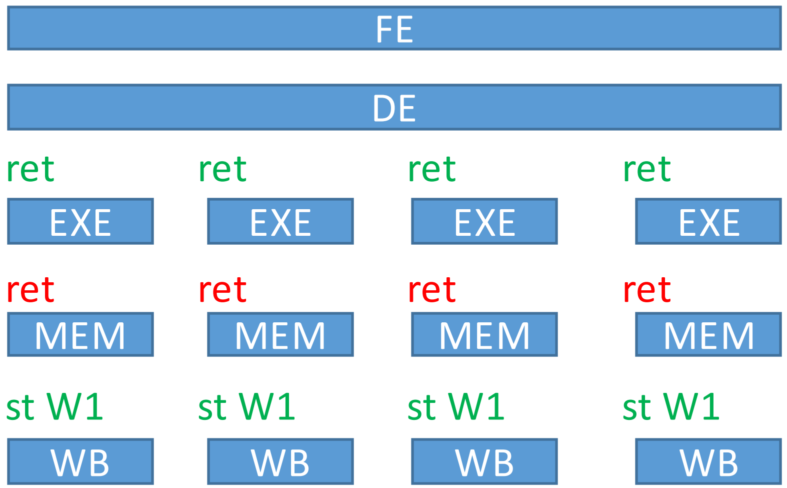
Execution Sequence (cont.)



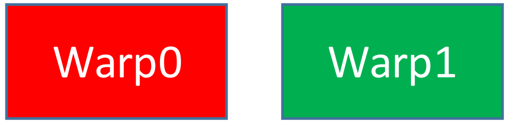

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



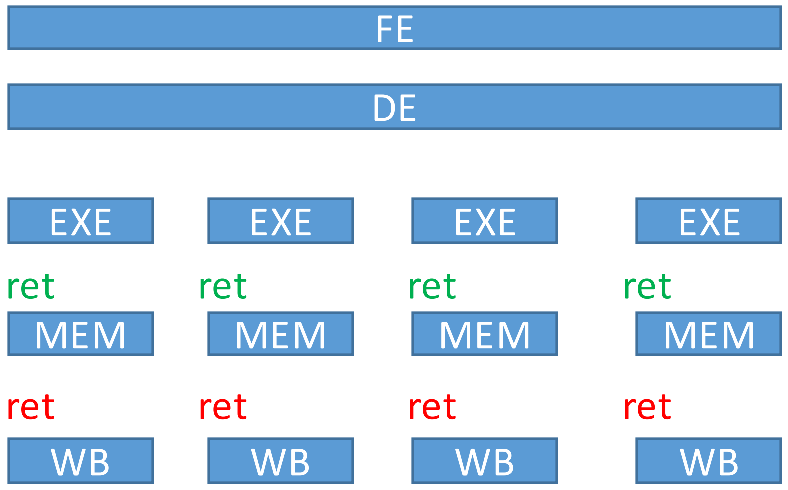
Execution Sequence (cont.)




```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



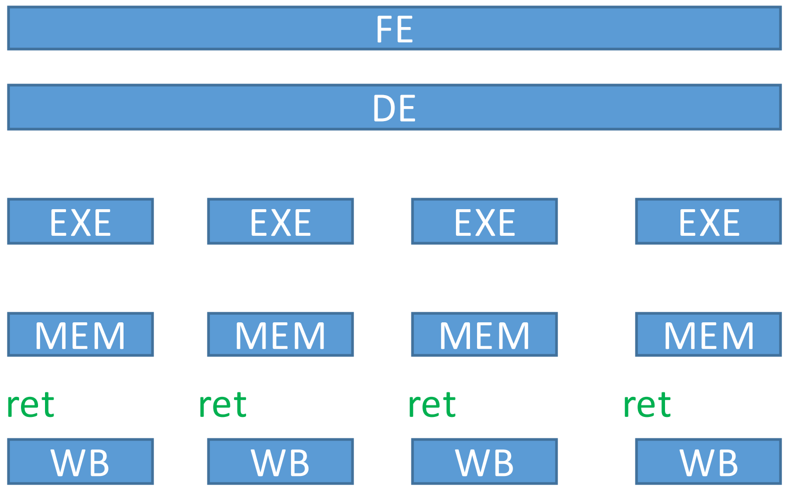
Execution Sequence (cont.)



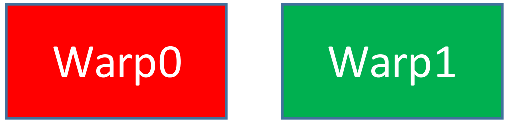

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

Warp0 Warp1

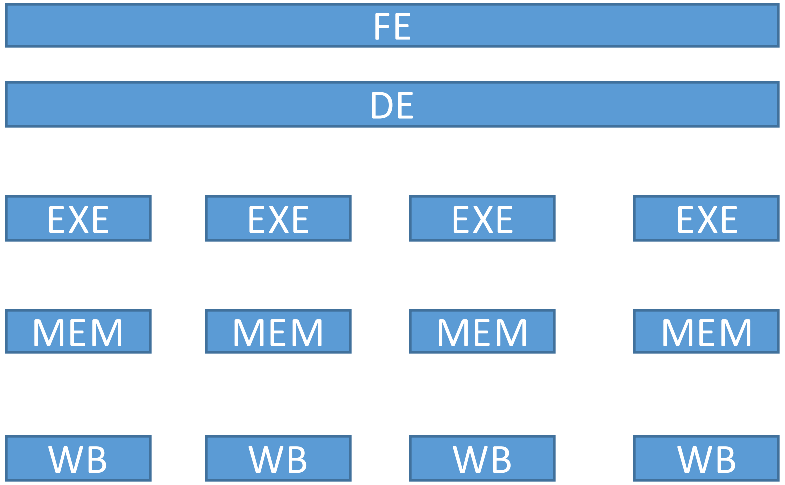
Execution Sequence (cont.)



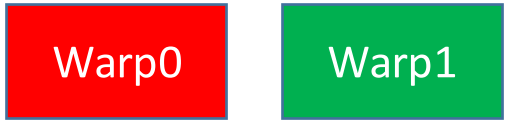

```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```



Execution Sequence (cont.)

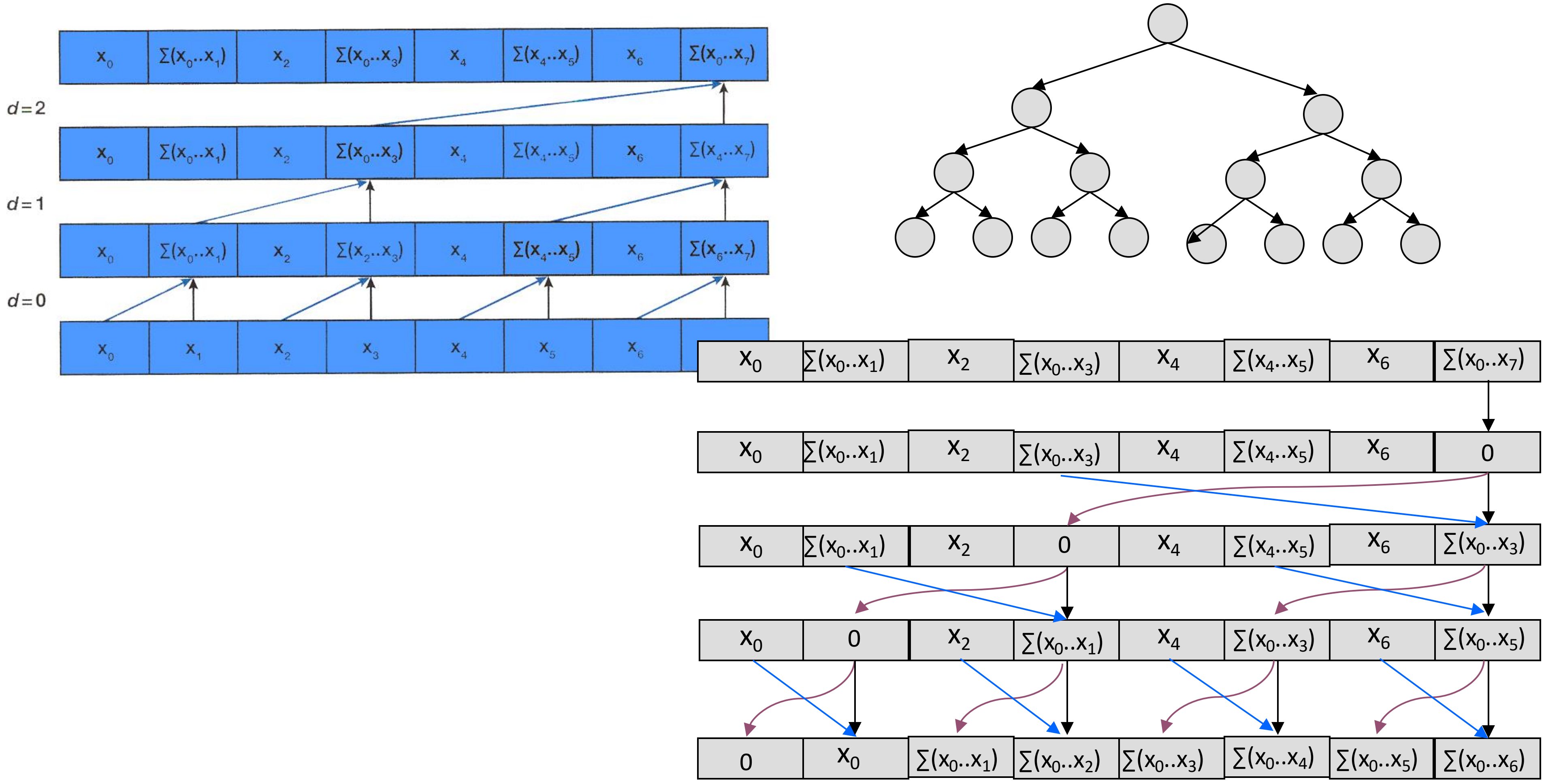



```
setp.lt.s32 %p, %r5, %rd4;
@p bra L1;
bra L2;

L1:
ld.global.f32 %f1, [%r6];
ld.global.f32 %f2, [%r7];
add.f32 %f3, %f1, %f2;

st.global.f32 [%r8], %f3;

L2:
ret;
```

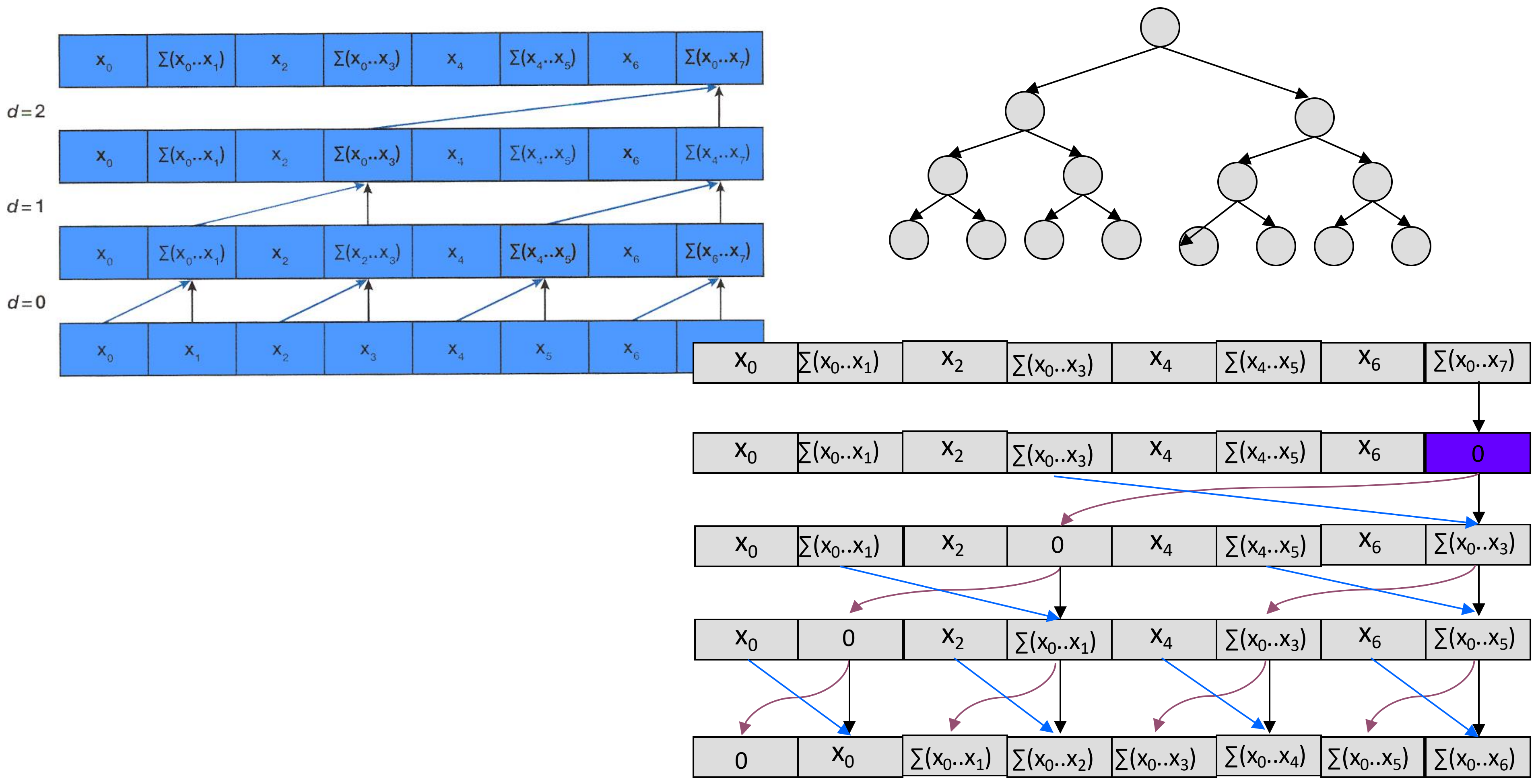


Note: Idealized execution without memory or special function delays

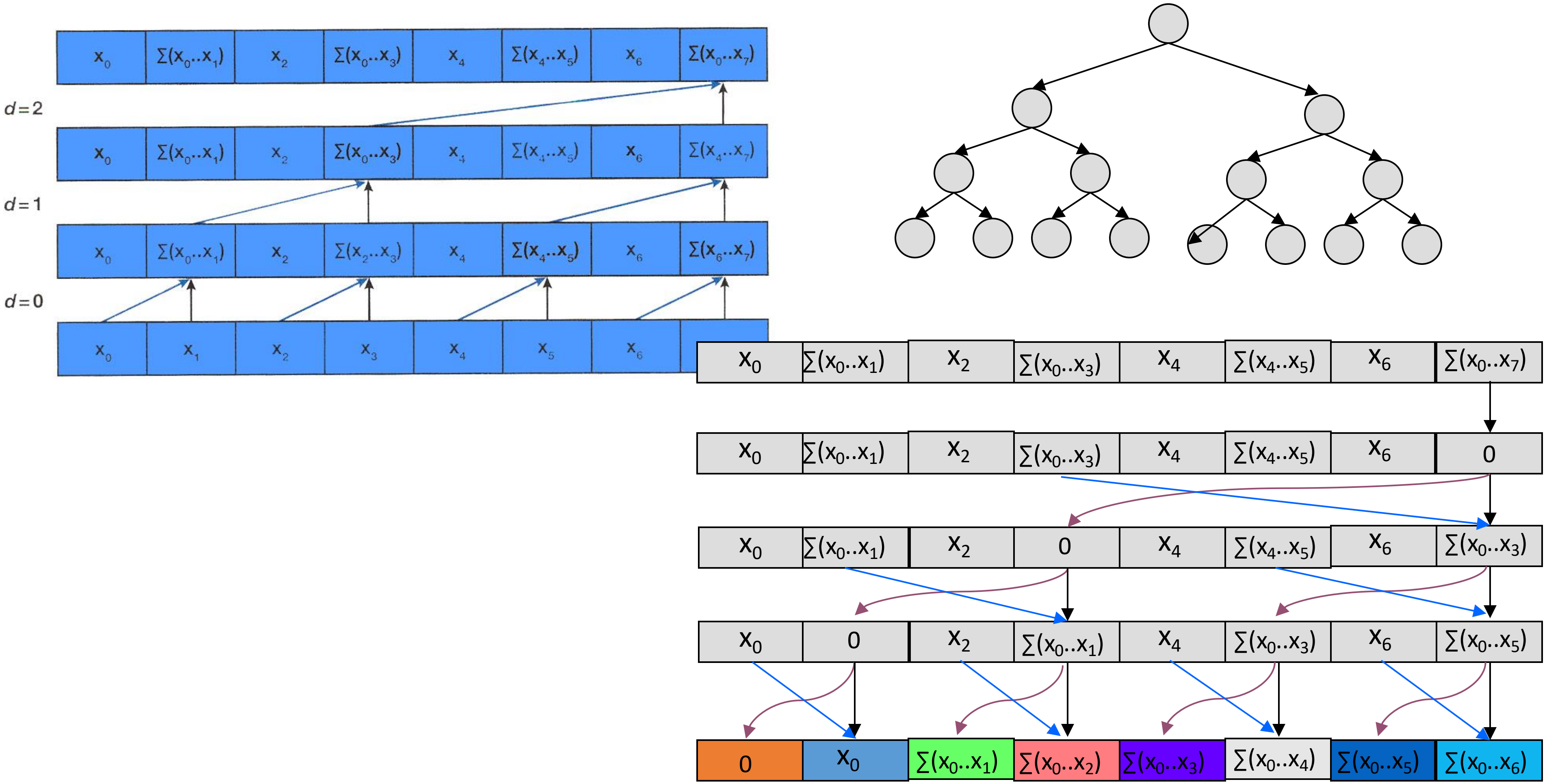
Tree analogy



Tree analogy



Tree analogy



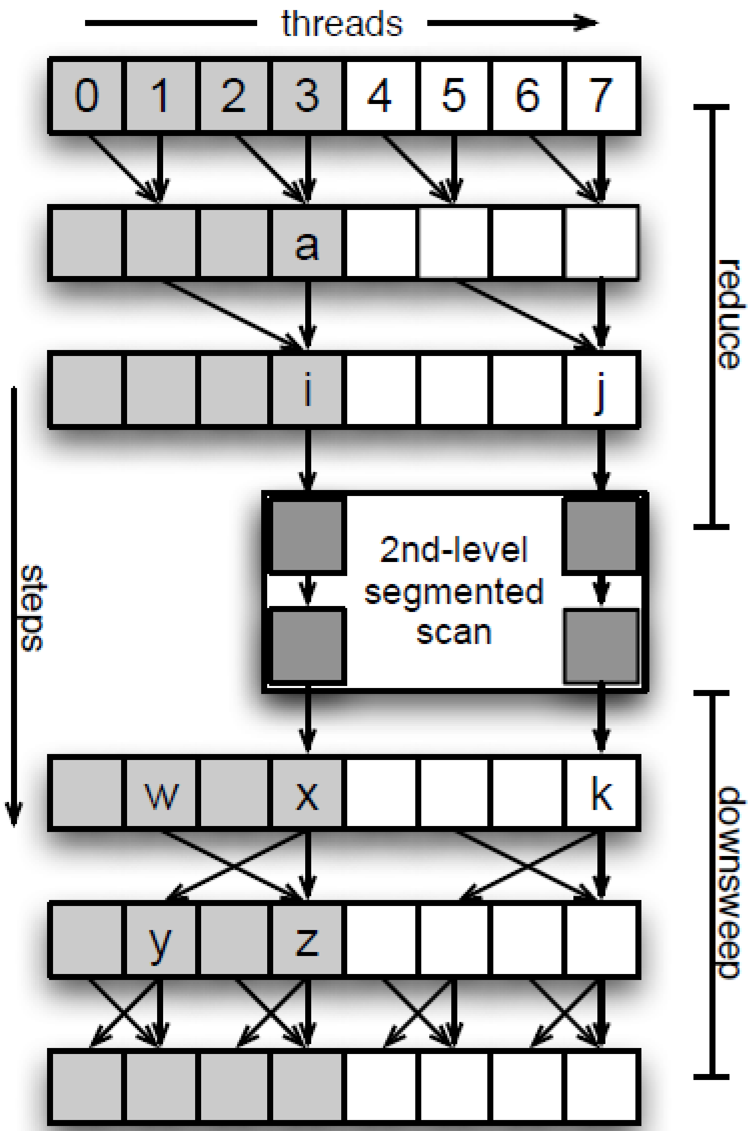
O(n) Segmented Scan

Up-Sweep

for $d = 1$ to $\log_2 n - 1$ **do**
  for all $k = 0$ to $n-1$ by 2^{d+1} in parallel **do**
    if $f[k + 2^{d+1} - 1]$ is not set **then**
      $x[k + 2^{d+1} - 1] \leftarrow x[k + 2^{d} - 1] + x[k + 2^{d+1} - 1]$
    $f[k + 2^{d+1} - 1] \leftarrow f[k + 2^{d} - 1] \mid f[k + 2^{d+1} - 1]$

- Down-Sweep

$x[n-1] \leftarrow 0$
for $d = \log_2 n - 1$ down to 0 **do**
  for all $k = 0$ to $n-1$ by 2^{d+1} in parallel **do**
    $t \leftarrow x[k+2^d-1]$
    $x[k+2^d-1] \leftarrow x[k+2^{d+1}-1]$
    if $f_i[k+2^d]$ is set **then**
      $x[k+2^{d+1}-1] \leftarrow 0$
    else if $f[k+2^d-1]$ is set **then**
      $x[k+2^{d+1}-1] \leftarrow t$
    else
      $x[k+2^{d+1}-1] \leftarrow t + x[k+2^{d+1}-1]$
    Unset flag $f[k+2^d-1]$



Features of segmented scan

- 3 times slower than unsegmented scan
- Useful for building broad variety of applications which are not possible with unsegmented scan.

Primitives built on scan

- Enumerate
    - enumerate([t f f t f t t]) = [0 1 1 1 2 2 3]
    - Exclusive scan of input vector
- Distribute (copy)
    - distribute([a b c][d e]) = [a a a][d d]
    - Inclusive scan of input vector
- Split and split-and-segment

Split divides the input vector into two pieces, with all the elements marked false on the left side of the output vector and all the elements marked true on the right.

Applications

- Quicksort
- Sparse Matrix-Vector Multiply
- Tridiagonal Matrix Solvers and Fluid Simulation
- Radix Sort
- Stream Compaction
- Summed-Area Tables

Quicksort

```
[5 3 7 4 6]     # initial input
[5 5 5 5 5]     # distribute pivot across segment
[f f t f t]     # input > pivot?
[5 3 4][7 6]    # split-and-segment
[5 5 5][7 7]    # distribute pivot across segment
[t f f][t f]    # input >= pivot?
[3 4 5][6 7]    # split-and-segment, done!
```

Sparse Matrix-Vector Multiplication

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix} += \begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}$$

$$\begin{aligned} \text{value} &= [a,b,c,d,e,f] \\ \text{index} &= [0,2,0,1,2,2] \\ \text{rowPtr} &= [0,2,5] \end{aligned}$$

$$\begin{aligned} \text{product} &= [x_0a, x_2b, x_0c, x_1d, x_2e, x_2f] && (1) \\ &= [[x_0a, x_2b][x_0c, x_1d, x_2e][x_2f]] && (2) \\ &= [[x_0a + x_2b, x_2b] \\ & \quad [x_0c + x_1d + x_2e, x_1d + x_2e, x_2e][x_2f]] && (3) \\ y &= y + [[x_0a + x_2b, x_0c + x_1d + x_2e, x_2f] && (4) \end{aligned}$$

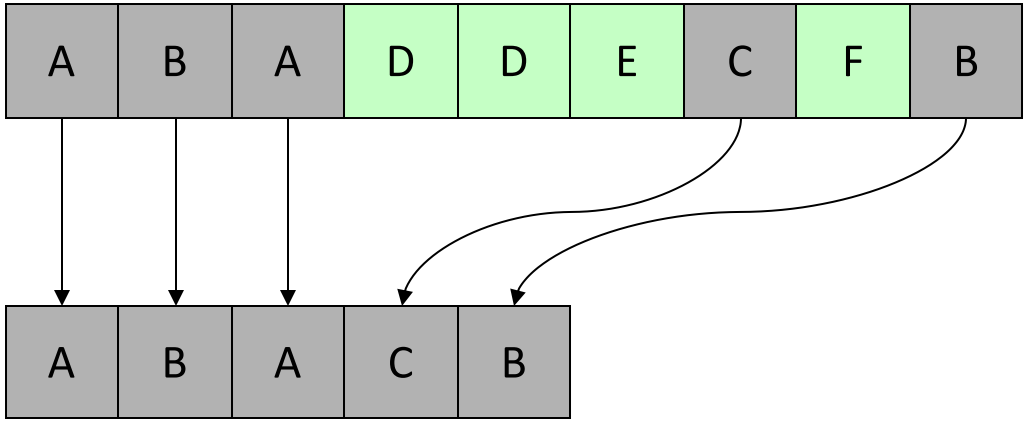
1. The first kernel runs over all entries. For each entry, it sets the corresponding `flag` to 0 and performs a multiplication on each entry: `product = x[index] * value`.
2. The next kernel runs over all rows and sets the head flag to 1 for each `rowPtr` in `flag` through a scatter. This creates one segment per row.
3. We then perform a backward segmented inclusive sum scan on the e elements in `product` with head flags in `flag`.
4. To finish, we run our final kernel over all rows, adding the value in `y` to the gathered value from `products[idx]`.

Stream Compaction

Definition:

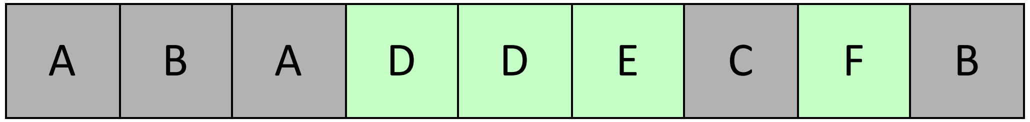
- Extracts the ‘interest’ elements from an array of elements and places them continuously in a new array

- Uses:
  - Collision Detection
  - Sparse Matrix Compression



Stream Compaction

Stream Compaction



Input: We want to preserve the gray elements

Stream Compaction

A	B	A	D	D	E	C	F	B
1	1	1	0	0	0	1	0	1

Input: We want to preserve the gray elements

Set a ‘1’ in each gray input

Stream Compaction

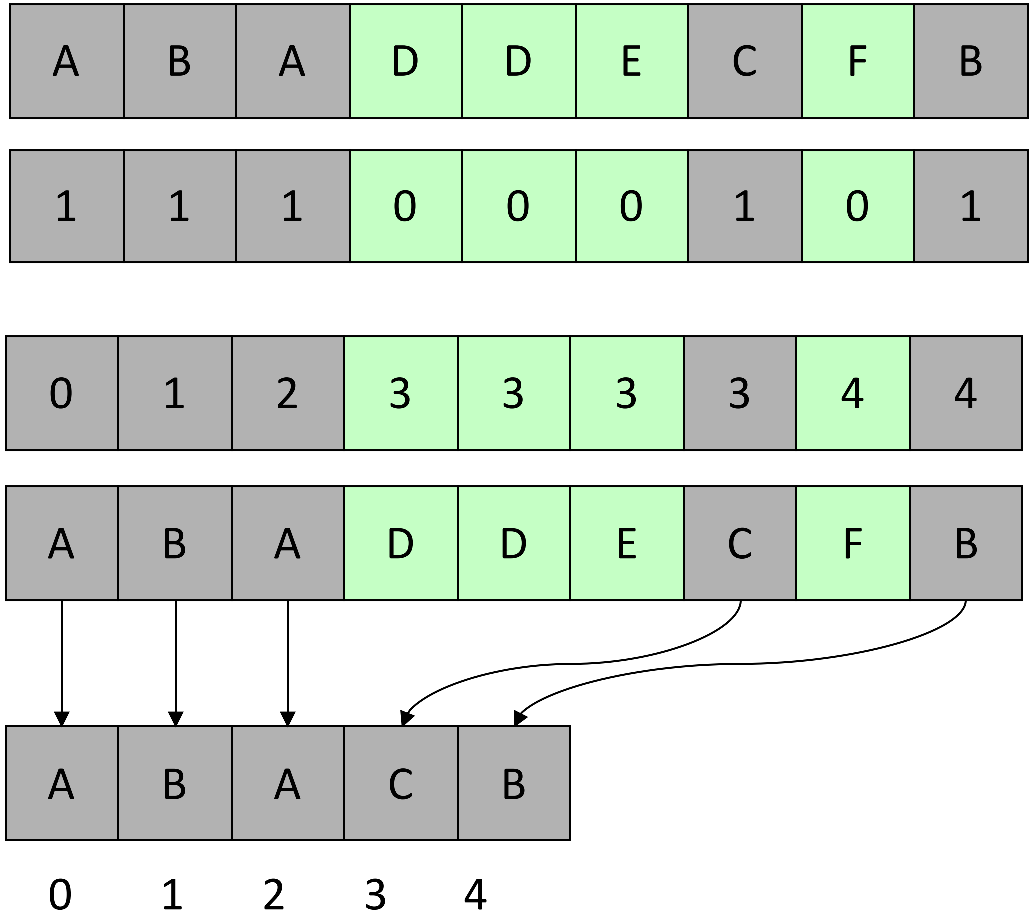
A	B	A	D	D	E	C	F	B
1	1	1	0	0	0	1	0	1
0	1	2	3	3	3	3	4	4

Input: We want to preserve the gray elements

Set a ‘1’ in each gray input

Scan

Stream Compaction



Input: We want to preserve the gray elements

Set a ‘1’ in each gray input

Scan

Scatter gray inputs to output using scan result as scatter address

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys
0	1	1	2	3	3	3	3	f = Scan the 1s

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys
0	1	1	2	3	3	3	3	f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys
0	1	1	2	3	3	3	3	f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

0-0+4 = 4	1-1+4 = 4	2-1+4 = 5	3-2+4 = 5	4-3+4 = 5	5-3+4 = 6	6-3+4 = 7	7-3+4 = 8	t = index – f + Total Falses

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys
0	1	1	2	3	3	3	3	f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

0-0+4 = 4	1-1+4 = 4	2-1+4 = 5	3-2+4 = 5	4-3+4 = 5	5-3+4 = 6	6-3+4 = 7	7-3+4 = 8	t = index – f + Total Falses
0	4	1	2	5	6	7	3	d = b ? t : f

Radix Sort Using Scan

100	111	010	110	011	101	001	000	Input Array
0	1	0	0	1	1	1	0	b = least significant bit
1	0	1	1	0	0	0	1	e = Insert a 1 for all false sort keys
0	1	1	2	3	3	3	3	f = Scan the 1s

Total Falses = e[n-1] + f[n-1]

0-0+4 = 4	1-1+4 = 4	2-1+4 = 5	3-2+4 = 5	4-3+4 = 5	5-3+4 = 6	6-3+4 = 7	7-3+4 = 8	t = index – f + Total Falses
0	4	1	2	5	6	7	3	d = b ? t : f
100	111	010	110	011	101	001	000	Scatter input using d as scatter address
100	010	110	000	111	011	101	001	

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