

Parallel Architectures

Parallel Algorithms

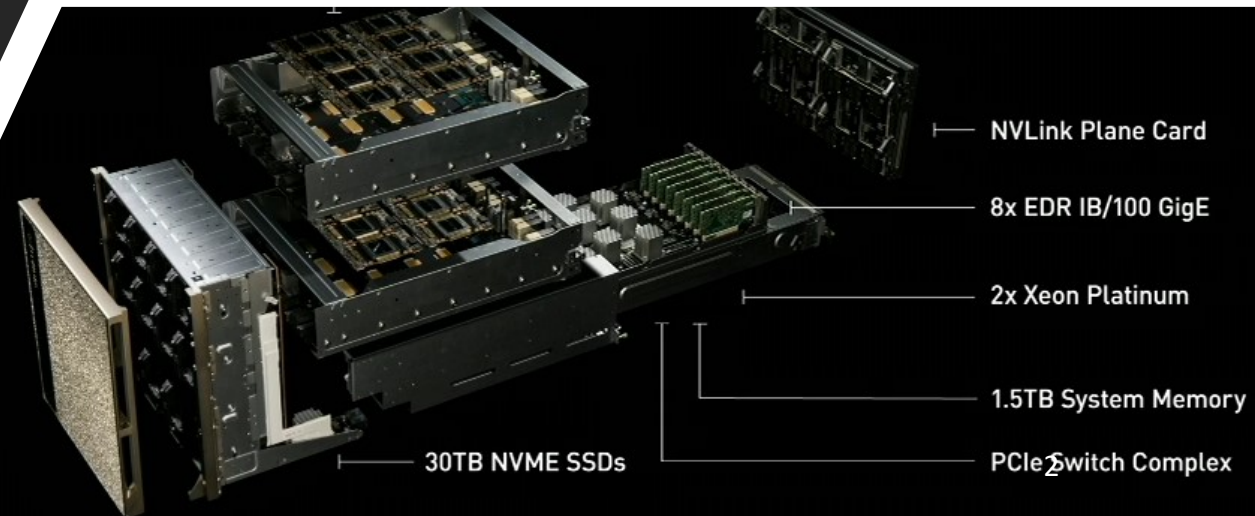
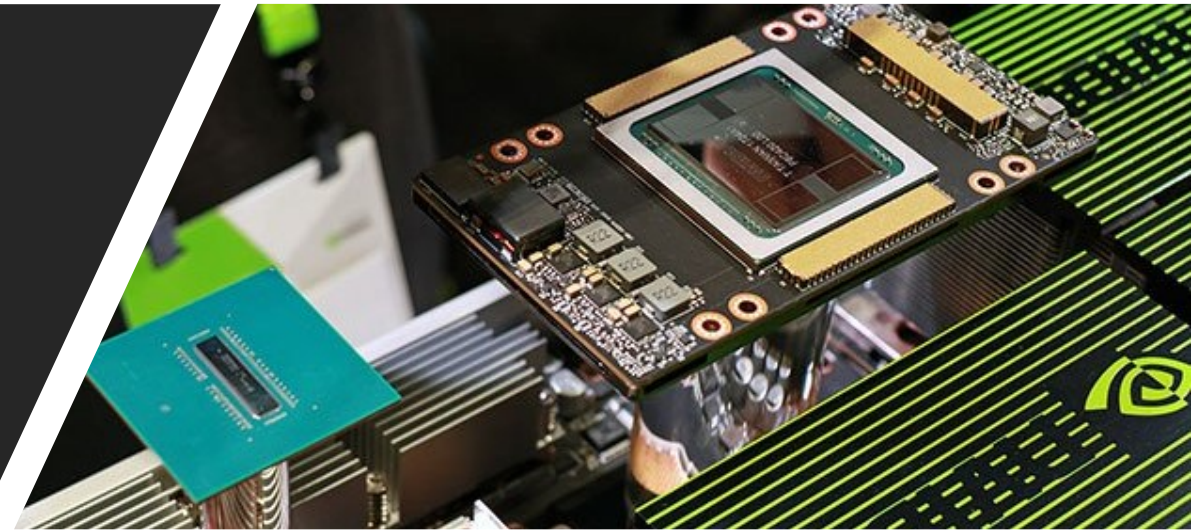
CUDA

Chris Rossbach

cs378h

Outline for Today

- Questions?
- Administrivia
 - Eldar-* machines should be available
- Agenda
 - Parallel Algorithms
 - CUDA
- Acknowledgements:
http://developer.download.nvidia.com/compute/developertrainingmaterials/presentations/cuda_language/Introduction_to_CUDA_C.pptx

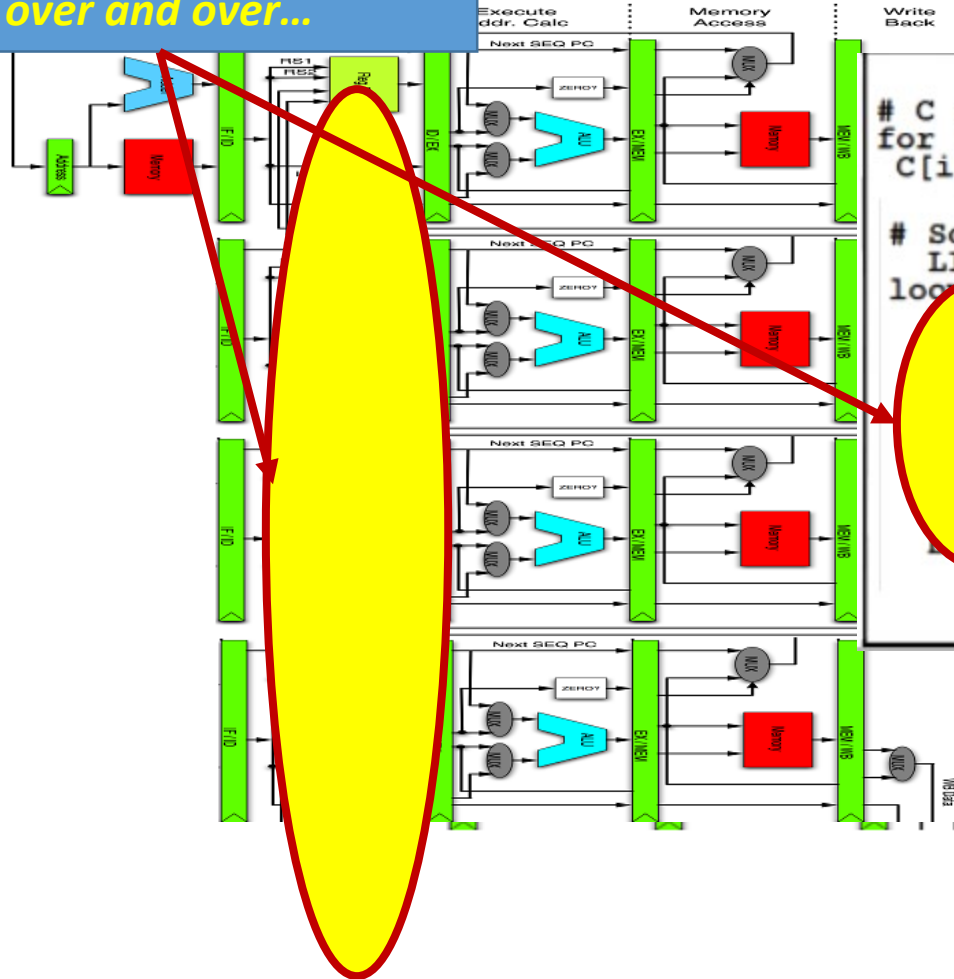


Faux Quiz Questions

- What is a reduction? A prefix sum? Why are they hard to parallelize and what basic techniques can be used to parallelize them?
- Define flow dependence, output dependence, and anti-dependence: give an example of each. Why/how do compilers use them to detect loop-independent vs loop-carried dependences?
- What is the difference between a thread-block and a warp?
- How/Why must programmers copy data back and forth to a GPU?
- What is “shared memory” in CUDA? Describe a setting in which it might be useful.
- CUDA kernels have implicit barrier synchronization. Why is `__syncthreads()` necessary in light of this fact?
- How might one implement locks on a GPU?
- What ordering guarantees does a GPU provide across different hardware threads’ access to a single memory location? To two disjoint locations?
- When is it safe for one GPU thread to wait (e.g. by spinning) for another?

Review: what is a vector processor

Dont decode same instruction over and over...

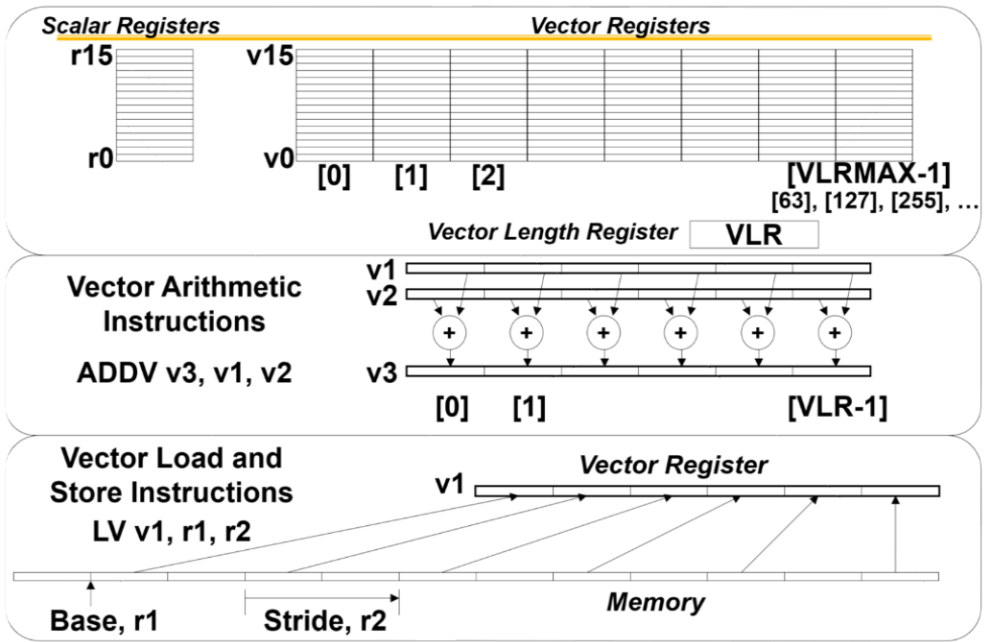


```
# C code
for (i=0; i<64; i++)
    C[i] = A[i] + B[i];

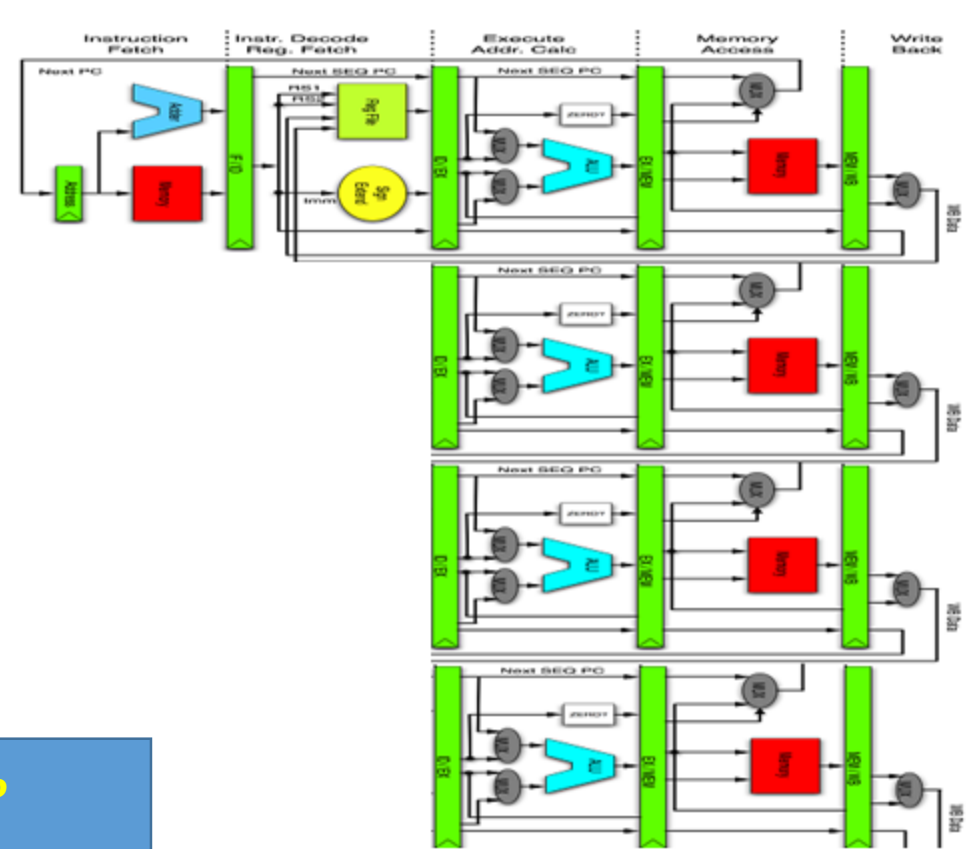
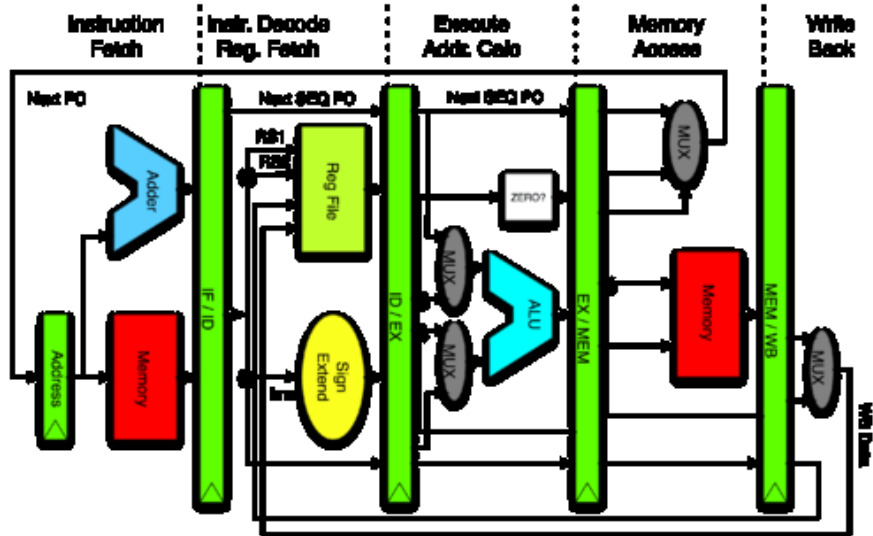
# Scalar Code
LI    R4, 64
loop:
    L.D F0, 0(R1)
    L.D F2, 0(R2)
    A.D F4, F2, F0
    S.D F4, 0(R3)
    DADDIU R1, 8
    DADDIU R2, 8
    DADDIU R3, 8
    DSUBIU R4, 1
    BNEZ R4, loop
```

Imp
• Ir
• S
• M
• Multiple different operands in parallel

<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i]; # Scalar Code LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre># Vector Code LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>
--	---



When does vector processing help?

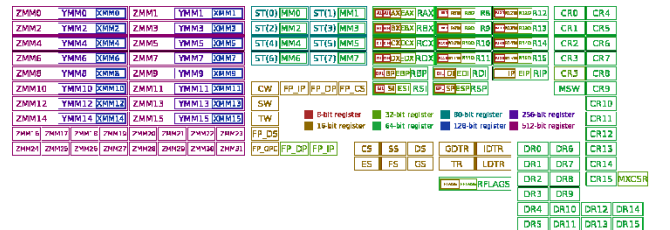
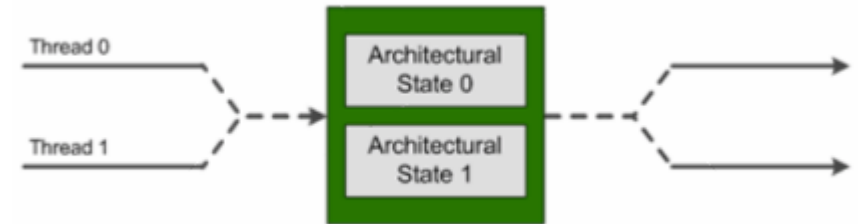
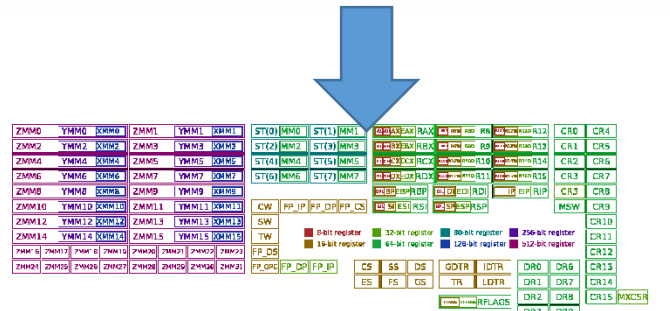
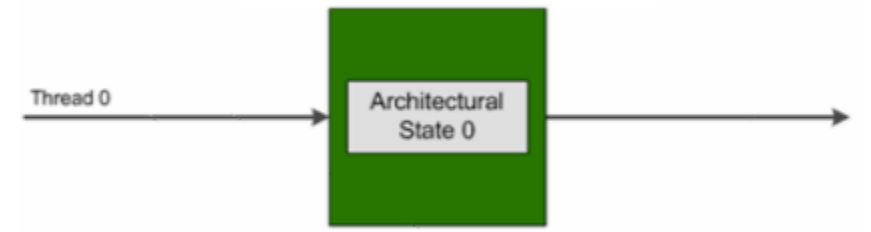
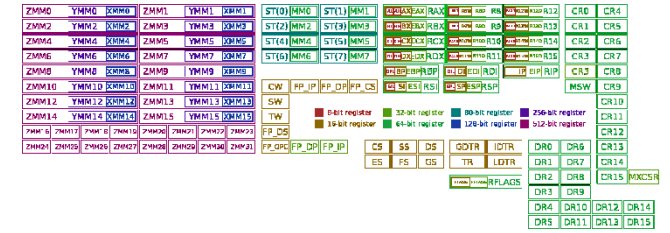


*What are the potential bottlenecks here?
When can it improve throughput?*

Only helps if memory can keep the pipeline busy!

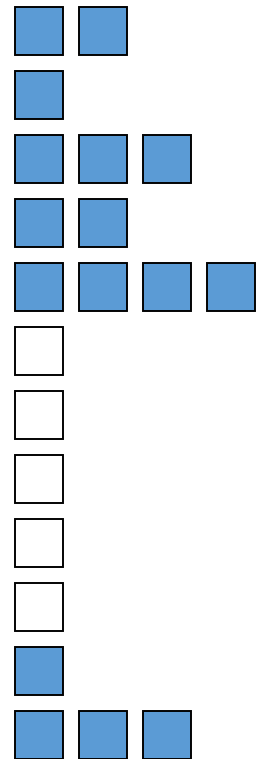
Hardware multi-threading

- Address memory bottleneck
- Share exec unit across
 - Instruction streams
 - Switch on stalls
- Looks like multiple cores to the OS
- Three variants:
 - Coarse
 - Fine-grain
 - Simultaneous

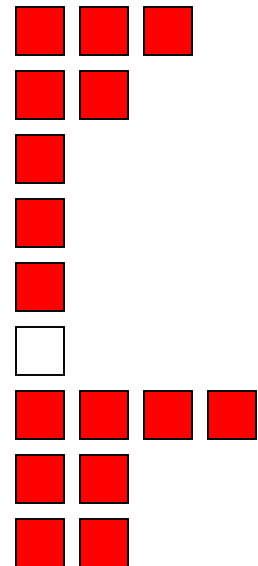


Running example

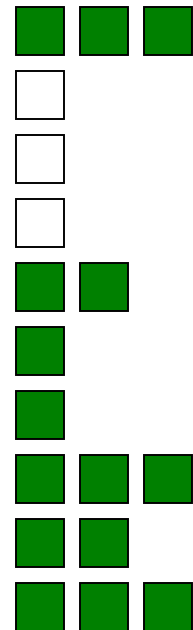
Thread A



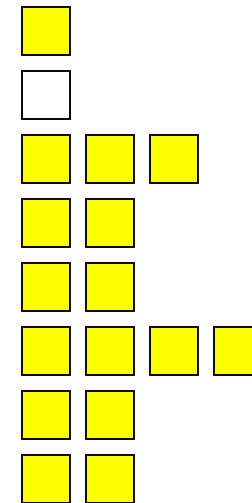
Thread B



Thread C



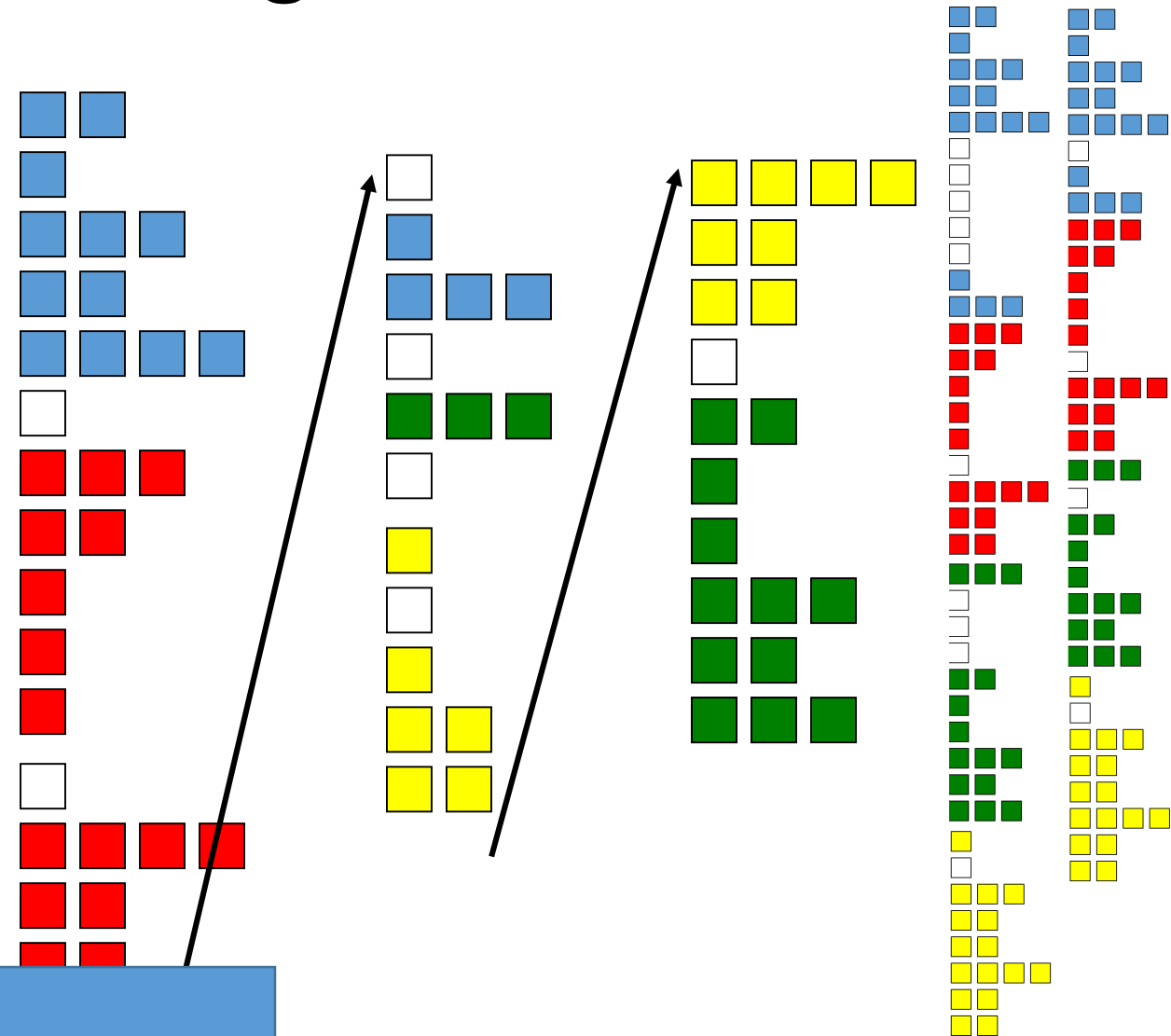
Thread D



- Colors → pipeline full
- White → stall

Coarse-grained multithreading

- Single thread runs until a costly stall
 - E.g. 2nd level cache miss
- Another thread starts during stall
 - Pipeline fill time requires several cycles!
- Hardware support required
 - PC and register file for each thread
 - Looks like another physical CPU to OS/software

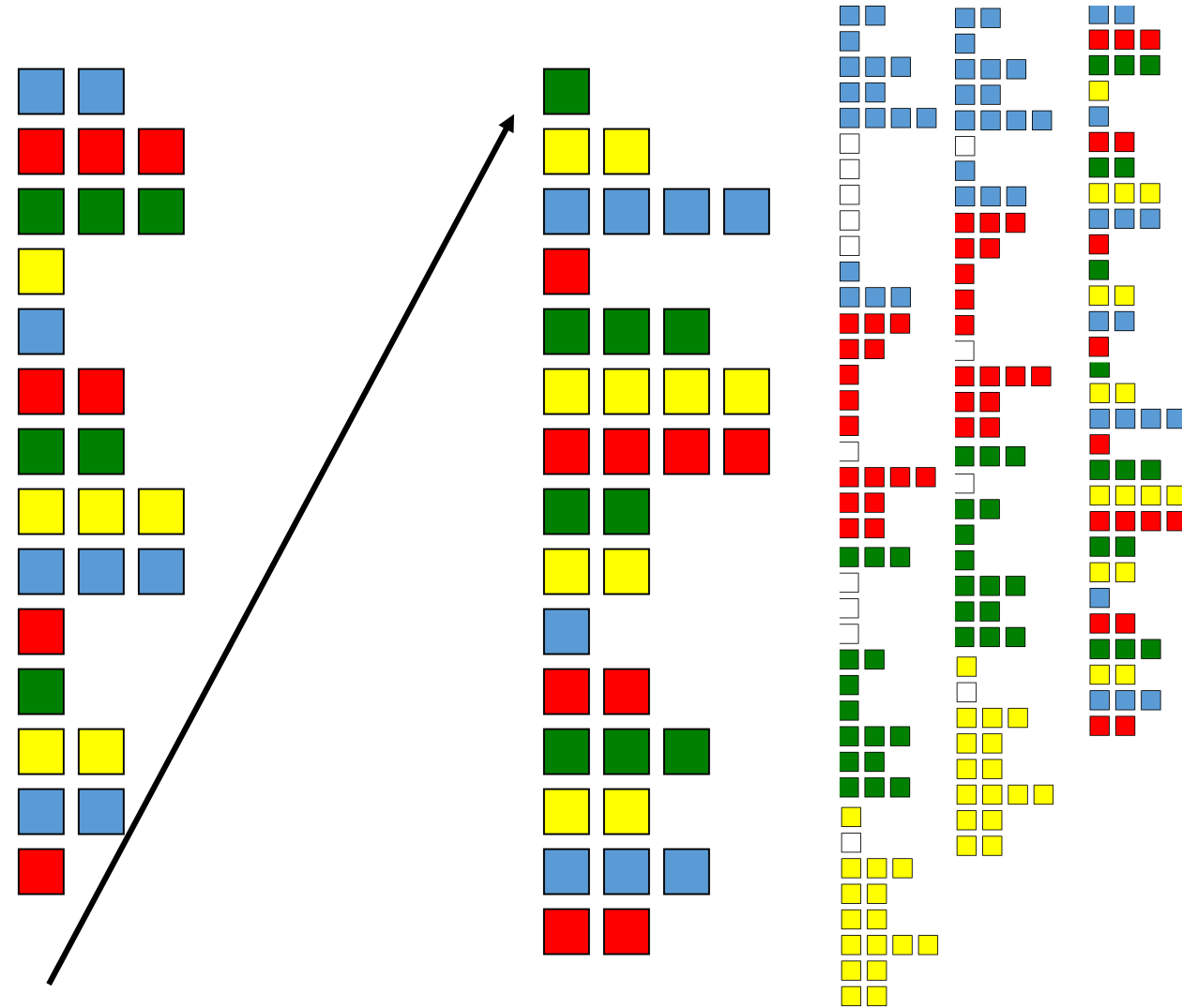


Pros? Cons?

Fine-grained multithreading

- Threads interleave instructions
 - Round-robin
 - Skip stalled threads
- Hardware support required
 - Separate PC and register file per thread
 - Hardware to control alternating pattern
- Naturally hides delays
 - Data hazards, Cache misses
 - Pipeline runs with rare stalls

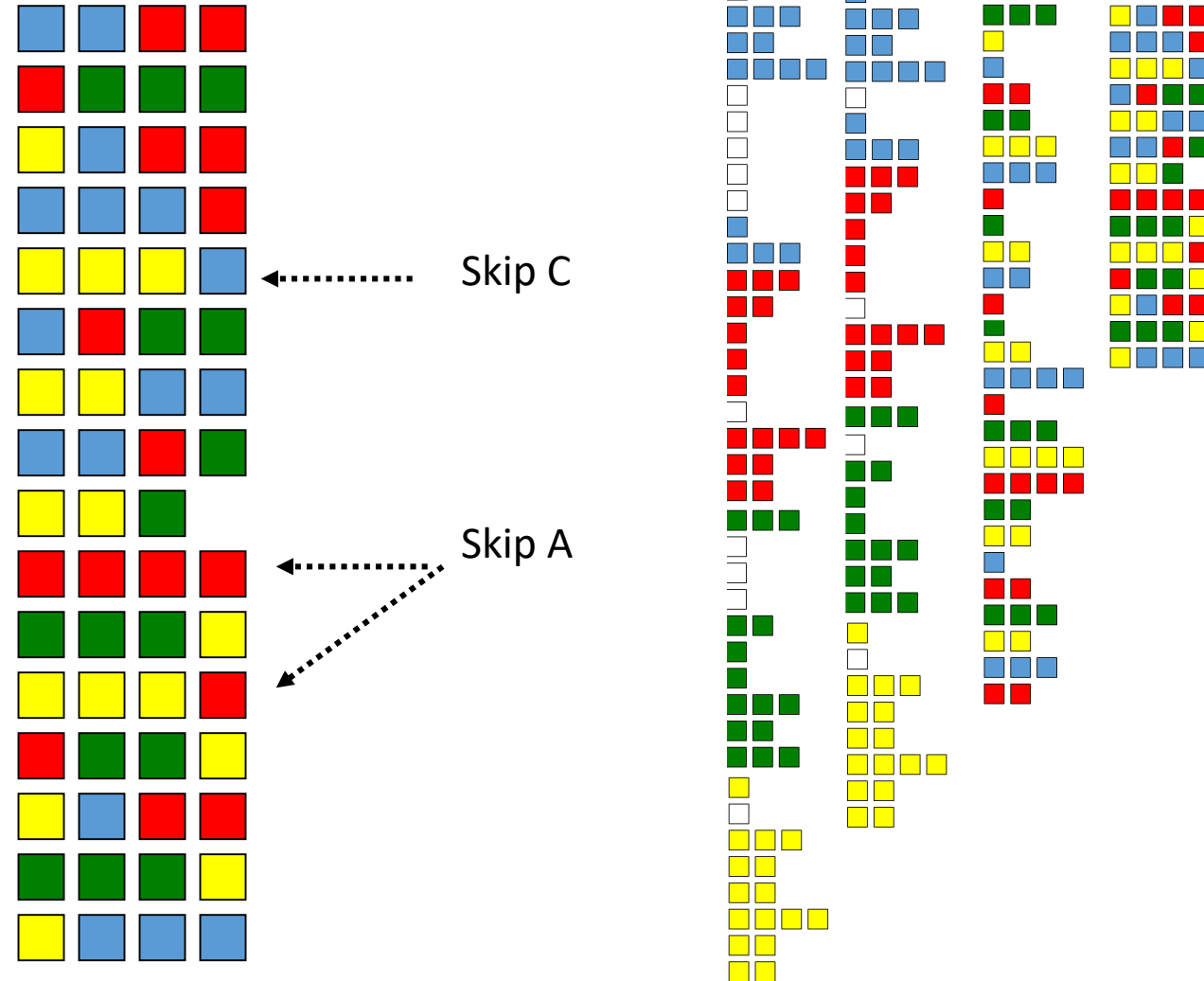
Pros? Cons?



Simultaneous Multithreading (SMT)

- Instructions from multiple threads issued on same cycle
 - Uses register renaming
 - dynamic scheduling facility of multi-issue architecture
- Hardware support:
 - Register files, PCs per thread
 - Temporary result registers pre commit
 - Support to sort out which threads get results from which instructions

Pros? Cons?



Why Vector and Multithreading Background?

GPU:

- A very wide vector machine
- Massively multi-threaded to hide memory latency
- *Originally designed for graphics pipelines...*

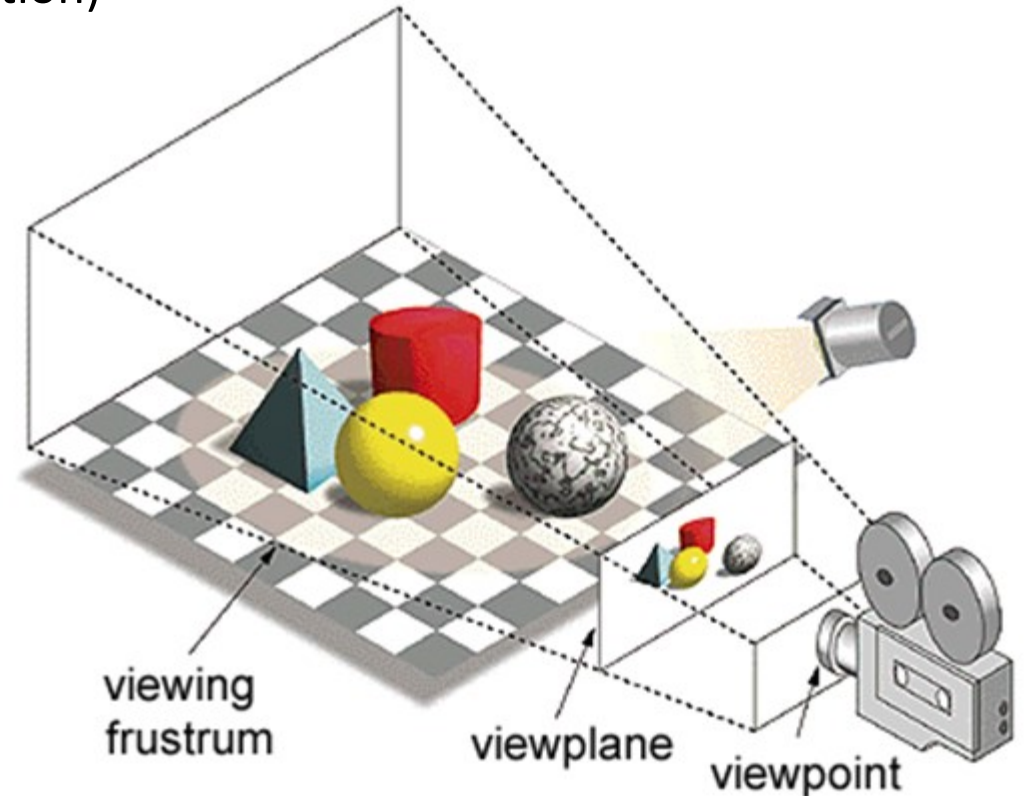
Graphics \approx Rendering

Inputs

- 3D world model(objects, materials)
 - Geometry modeled w triangle meshes, surface normals
 - GPUs subdivide triangles into “fragments” (rasterization)
 - Materials modeled with “textures”
 - Texture coordinates, sampling “map” textures \rightarrow geometry
- Light locations and properties
 - Attempt to model surface/light interactions with modeled objects/materials
- View point

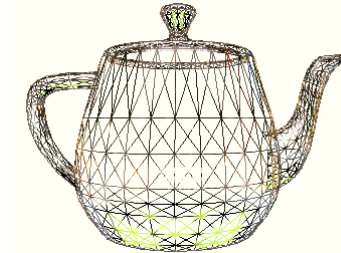
Output

- 2D projection seen from the view-point



Grossly over-simplified rendering algorithm

```
foreach(vertex v in model)
    map  $v_{\text{model}} \rightarrow v_{\text{view}}$ 
fragment[] frags = {};
foreach triangle t ( $v_0, v_1, v_2$ )
    frags.add(rasterize(t));
foreach fragment f in frags
    choose_color(f);
display(visible_fragments(frags));
```



Algorithm \rightarrow Graphics Pipeline

```
foreach(vertex v in model)
```

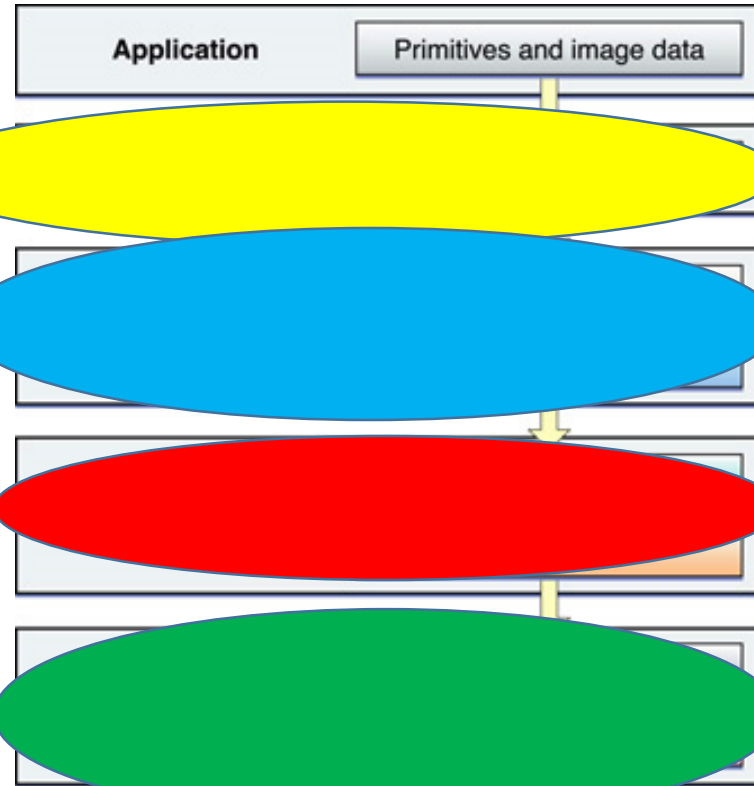


```
fragment[] frags = {};
```

```
foreach triangle t (v0, v1, v2)
```



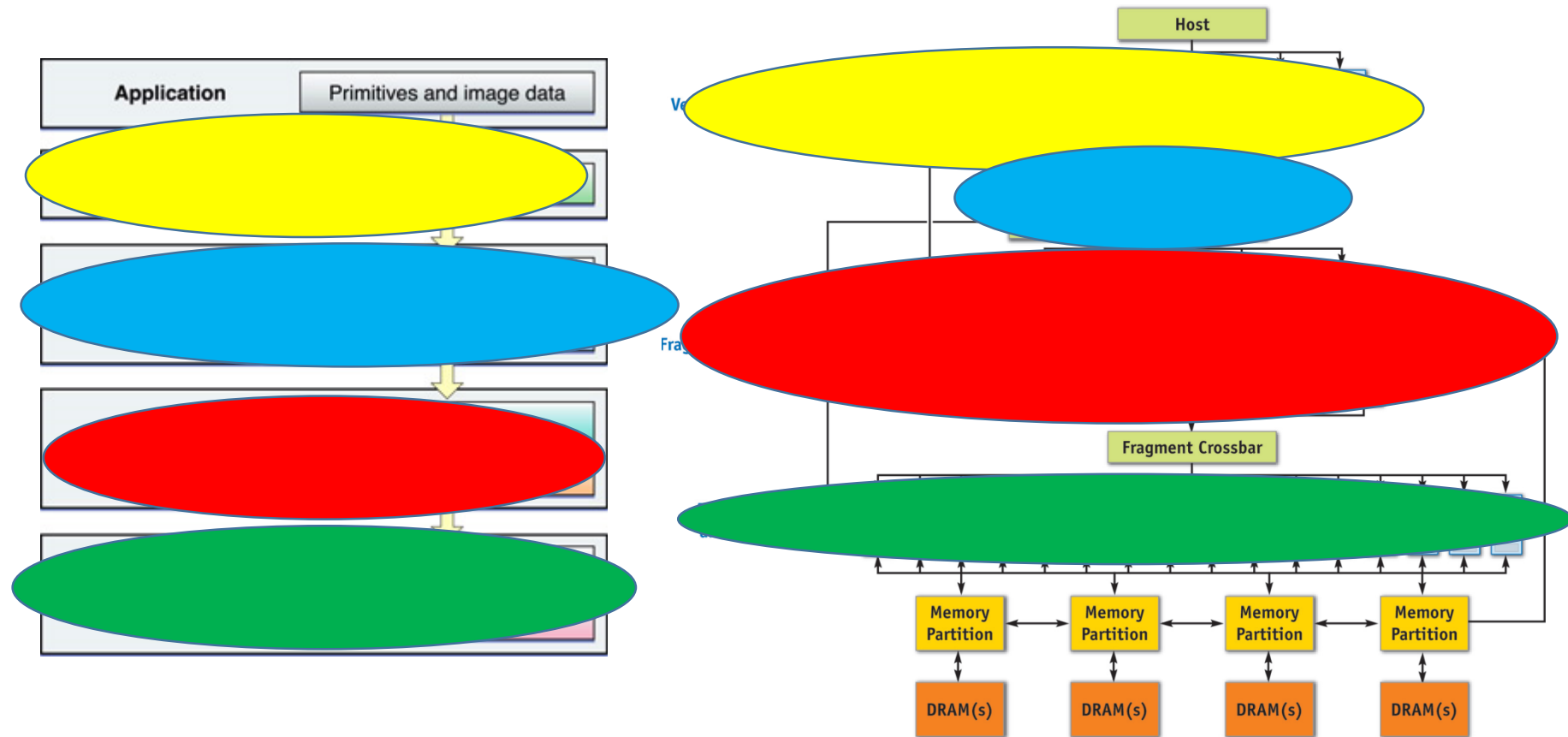
```
foreach fragment f in frags
```



OpenGL pipeline

To first order, DirectX looks the same!

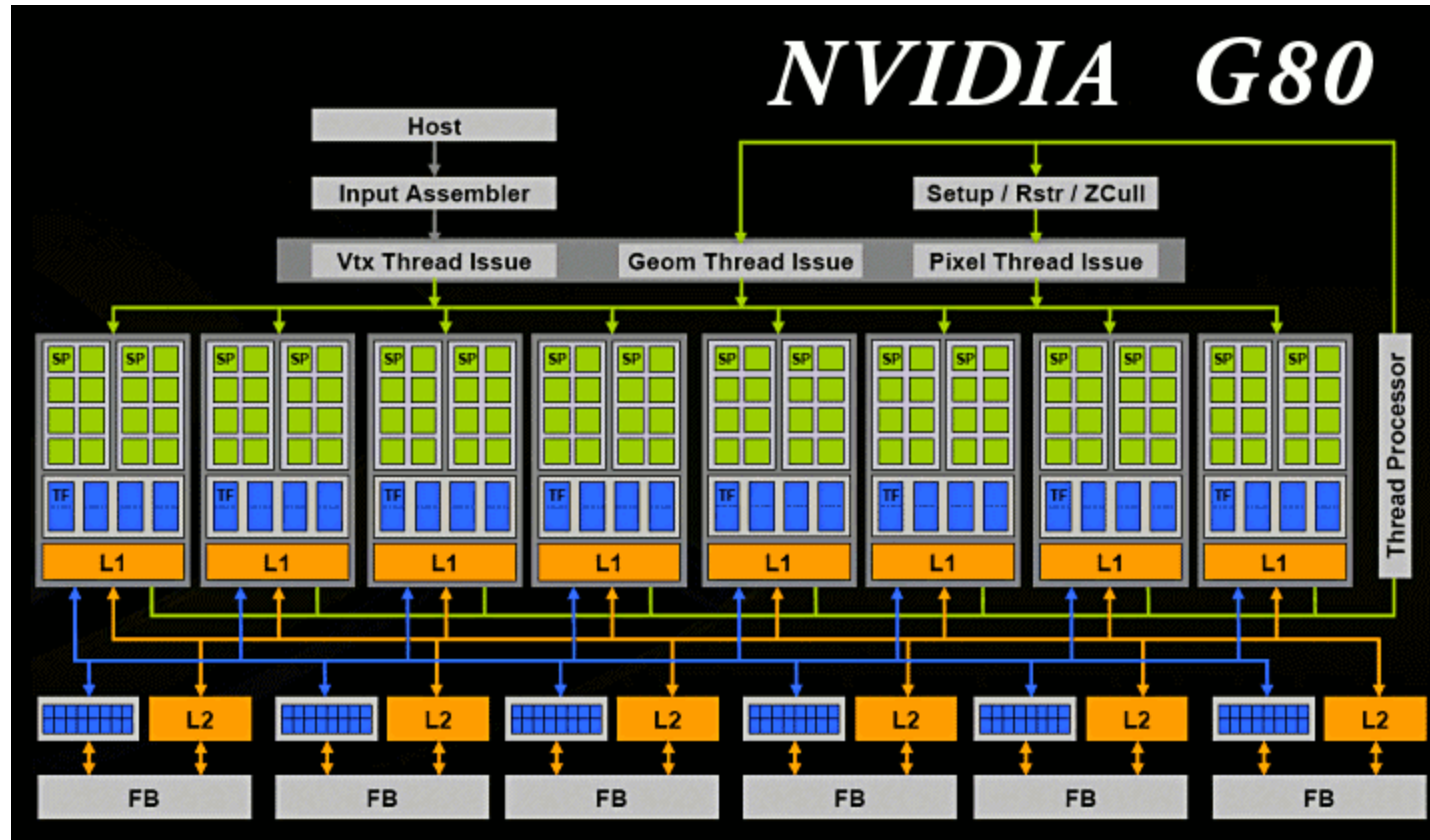
Graphics pipeline → GPU architecture



Limited “programmability” of shaders:
Minimal/no control flow
Maximum instruction count

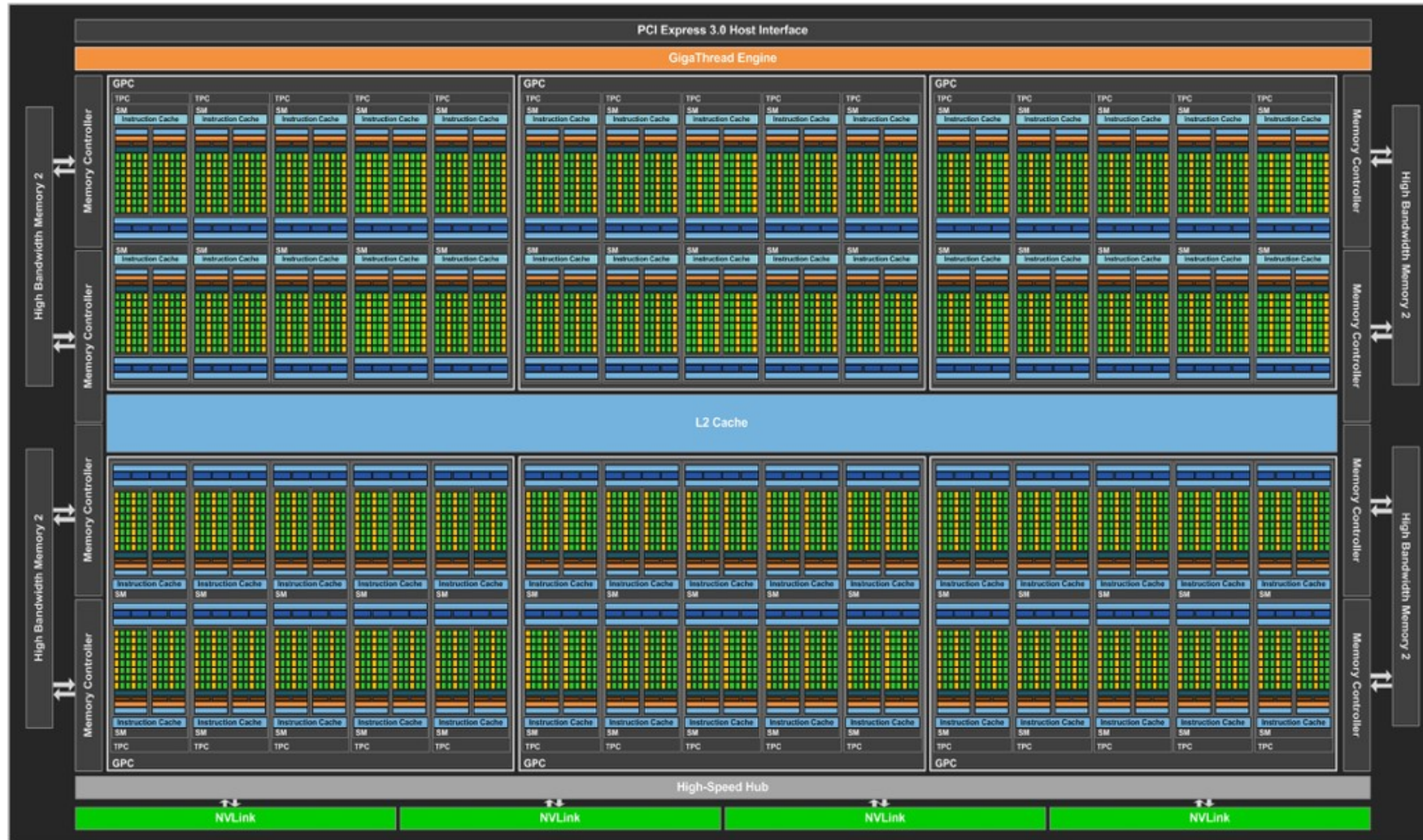
GeForce 6 series

Late Modernity: unified shaders



Mapping to Graphics pipeline no longer apparent
Processing elements no longer specialized to a particular role
Model supports *real* control flow, larger instr count

Mostly Modern: Pascal



Cross-generational GPU observations

GPUs designed for parallelism in graphics pipeline:

- Data
 - Per-vertex
 - Per-fragment
 - Per-pixel
- Task
 - Vertex processing
 - Fragment processing
 - Rasterization
 - Hidden-surface elimination
- MLP
 - HW multi-threading for hiding memory latency

- Simple cores
- Single instruction stream

Even as GPU architectures become more general, certain assumptions persist:

1. Data parallelism is *trivially* exposed
2. All problems look like painting a box with colored dots

But what if my problem isn't painting a box?!?!

OR
ing

Programming Model

- ***GPUs are I/O devices, managed by user-code***
- “kernels” == “shader programs”
- 1000s of HW-scheduled threads per kernel
- Threads grouped into independent blocks.
 - Threads in a block can synchronize (barrier)
 - This is the **only** synchronization
- “Grid” == “launch” == “invocation” of a kernel
 - a group of blocks (or warps)

***Need codes that are 1000s-X
parallel...***

Parallel Algorithms

- Sequential algorithms often do not permit easy parallelization
 - Does not mean there work has no parallelism
 - A different approach can yield parallelism
 - but often changes the algorithm
 - Parallelizing != just adding locks to a sequential algorithm
- Parallel Patterns
 - Map
 - Scatter, Gather
 - Reduction
 - Scan
 - Search, Sort

If you can express your algorithm using these patterns, an apparently fundamentally sequential algorithm can be made parallel

Map

- Inputs
 - Array A
 - Function $f(x)$
- $\text{map}(A, f) \rightarrow$ apply $f(x)$ on all elements in A
- Parallelism trivially exposed
 - $f(x)$ can be applied in parallel to all elements, in principle

```
for(i=0; i<numPoints; i++) {  
    labels[i] = findNearestCenter(points[i]);  
}
```



```
map(points, findNearestCenter)
```

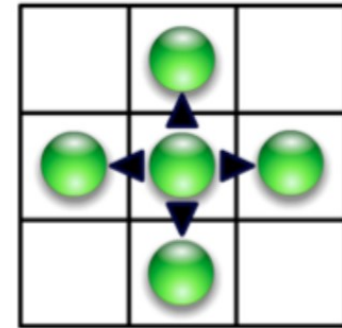
Scatter and Gather

- Gather:
 - Read multiple items to single /packed location
- Scatter:
 - Write single/packed data item to multiple locations
- Inputs: x, y, indeces, N

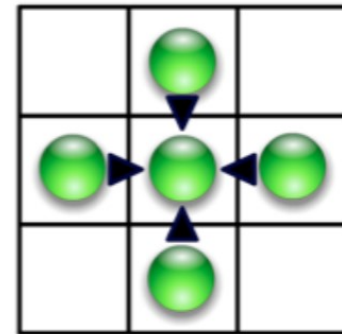
```
for (i=0; i<N; ++i)  
x[i] = y[idx[i]];      gather(x, y, idx)
```

```
for (i=0; i<N; ++i)  
y[idx[i]] = x[i];      scatter(x, y, idx)
```

Why is this useful on a box-drawing machine?



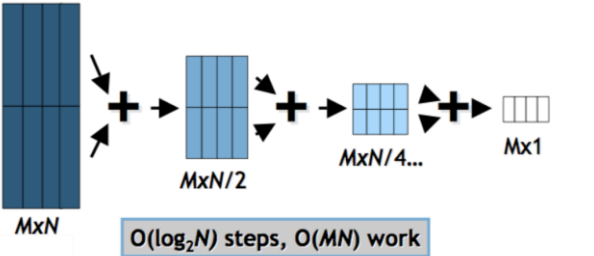
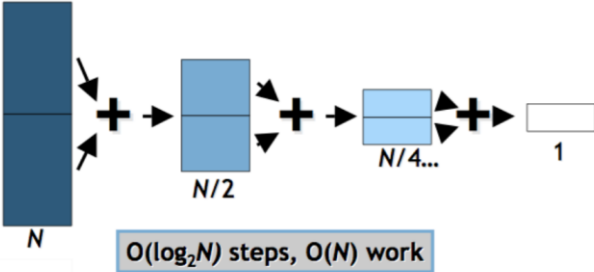
Scatter



Gather

Reduce

- Input
 - Associative operator **op**
 - Ordered set $s = [a, b, c, \dots z]$
- $\text{Reduce}(\text{op}, s)$ returns $a \text{ op } b \text{ op } c \dots \text{ op } z$



```
for(i=0; i<N; ++i) {  
    accum += (point[i]*point[i])  
}
```

→ accum = reduce(*, point)

Why must op be associative?

Scan (prefix sum)

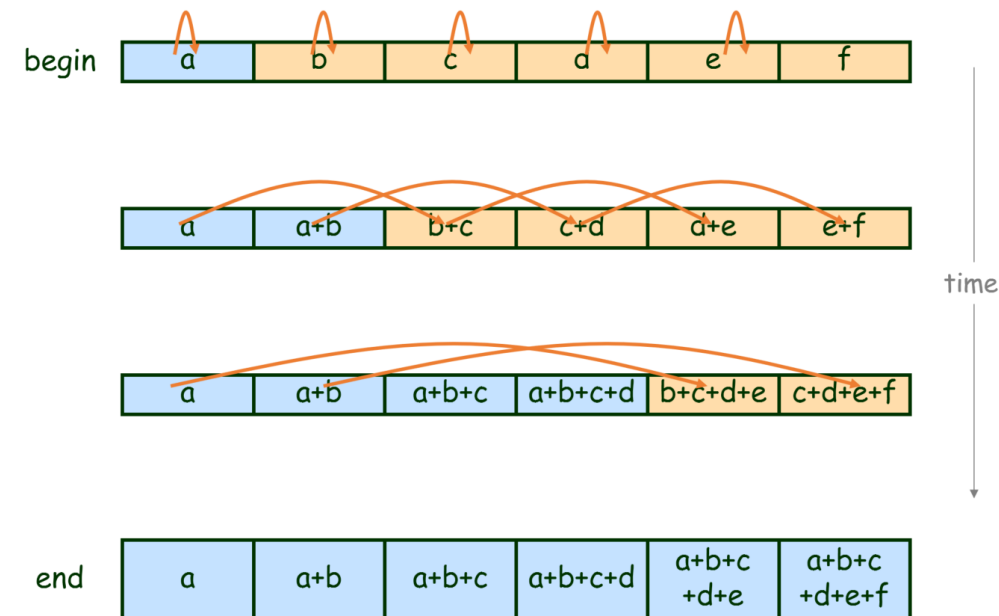
- Input

- Associative operator **op**
- Ordered set $s = [a, b, c, \dots z]$
- Identity I

- $\text{scan}(\text{op}, s) = [I, a, (a \text{ op } b), (a \text{ op } b \text{ op } c) \dots]$

- Scan is the workhorse of parallel algorithms:

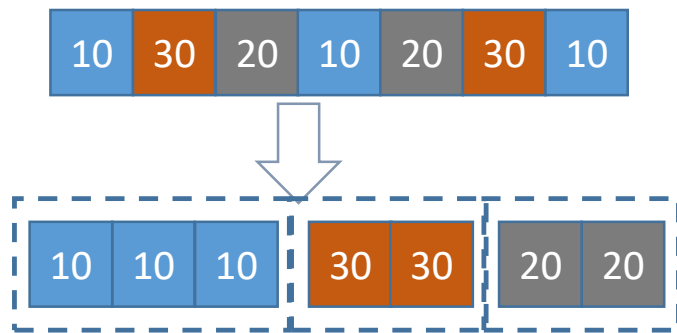
- Sort, histograms, sparse matrix, string compare, ...





GroupBy

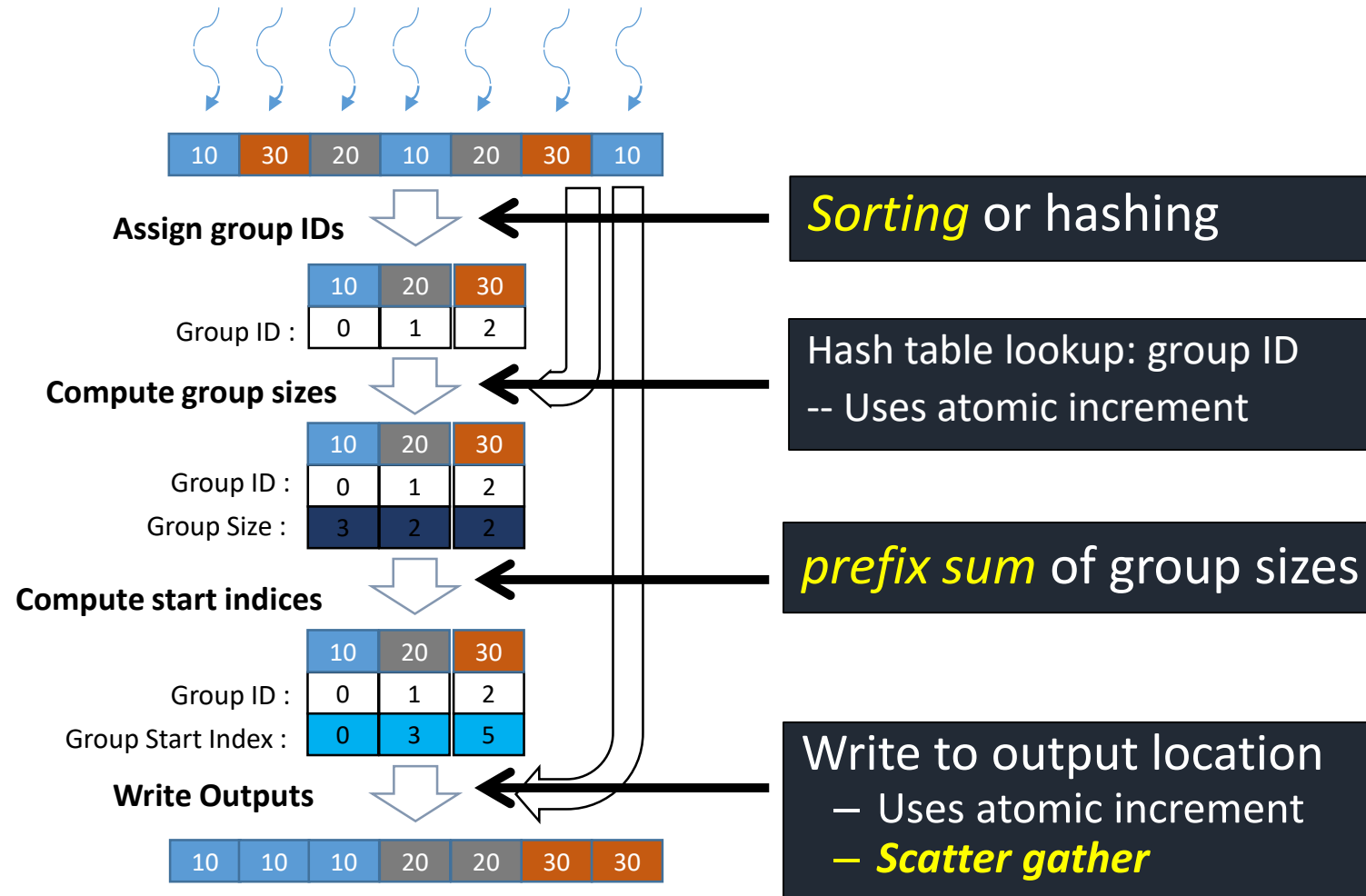
- Group a collection by key
- Lambda function maps elements \rightarrow key

```
var res = ints.GroupBy(x => x);
```



```
foreach(T elem in PF(ints))  
{  
    key    = KeyLambda(elem);  
    group = GetGroup(key)   
    group.Add(elem);   
}
```

GroupBy using parallel primitives



Sort

- OK, let's build a parallel sort

Summary

Re-expressing apparently sequential algorithms as combinations of parallel patterns is a common technique when targeting GPUs

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

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()

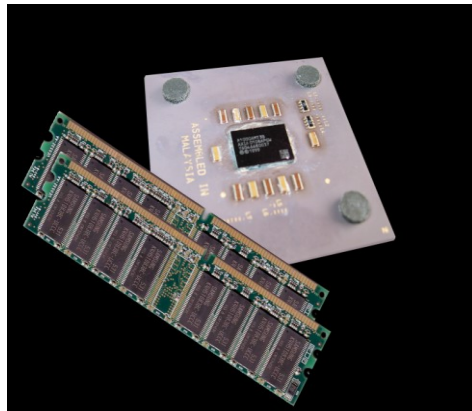
Asynchronous operation

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 gid = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

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

    // Store the result
    out[gid] = 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);

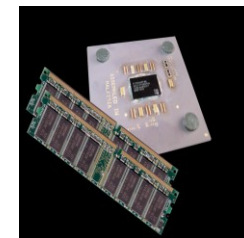
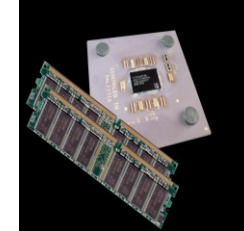
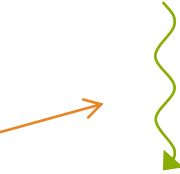
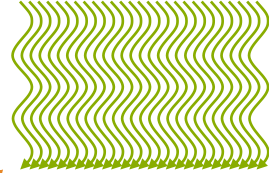
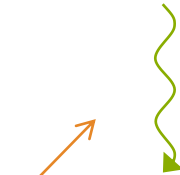
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

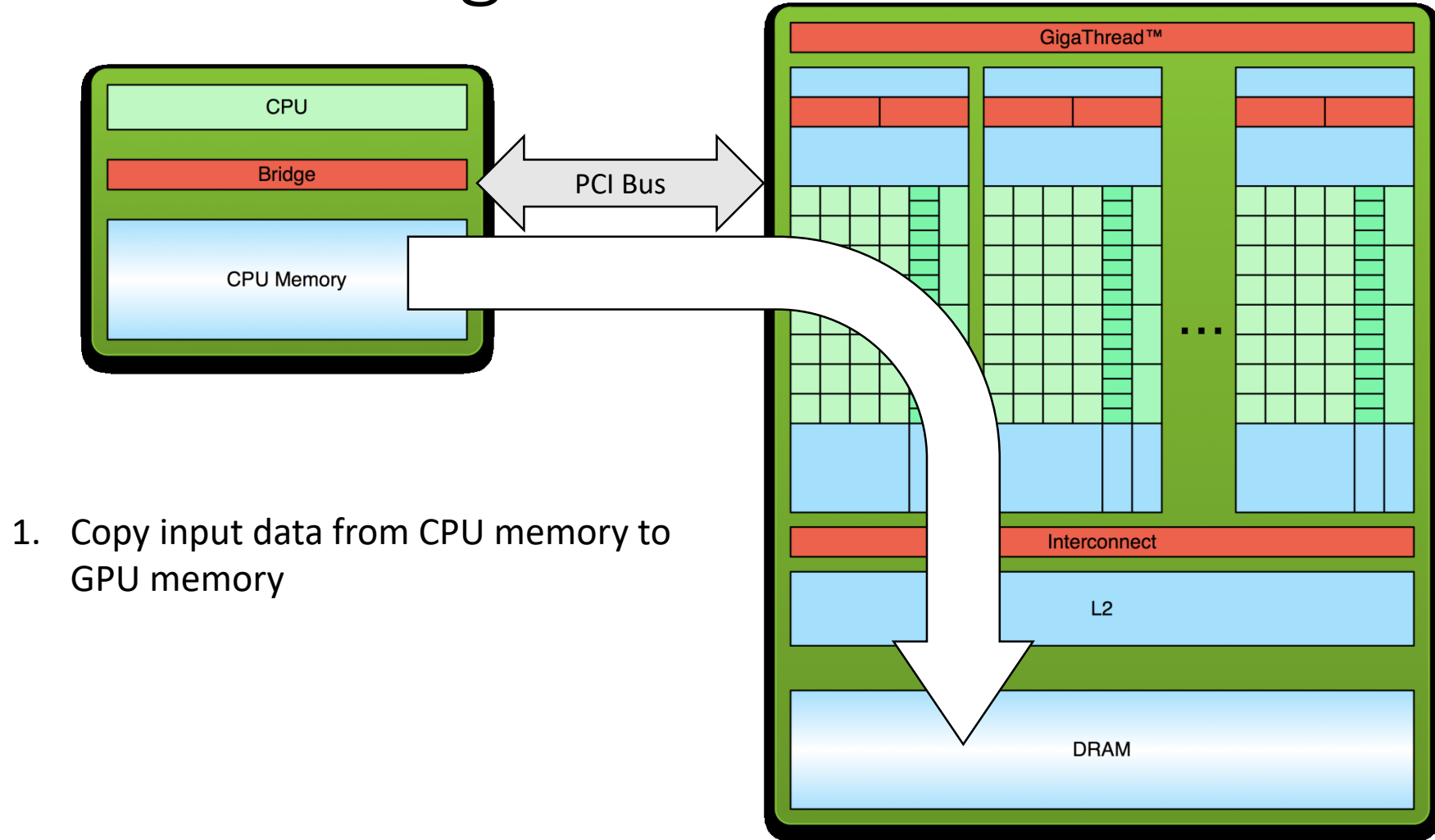
serial code

parallel code

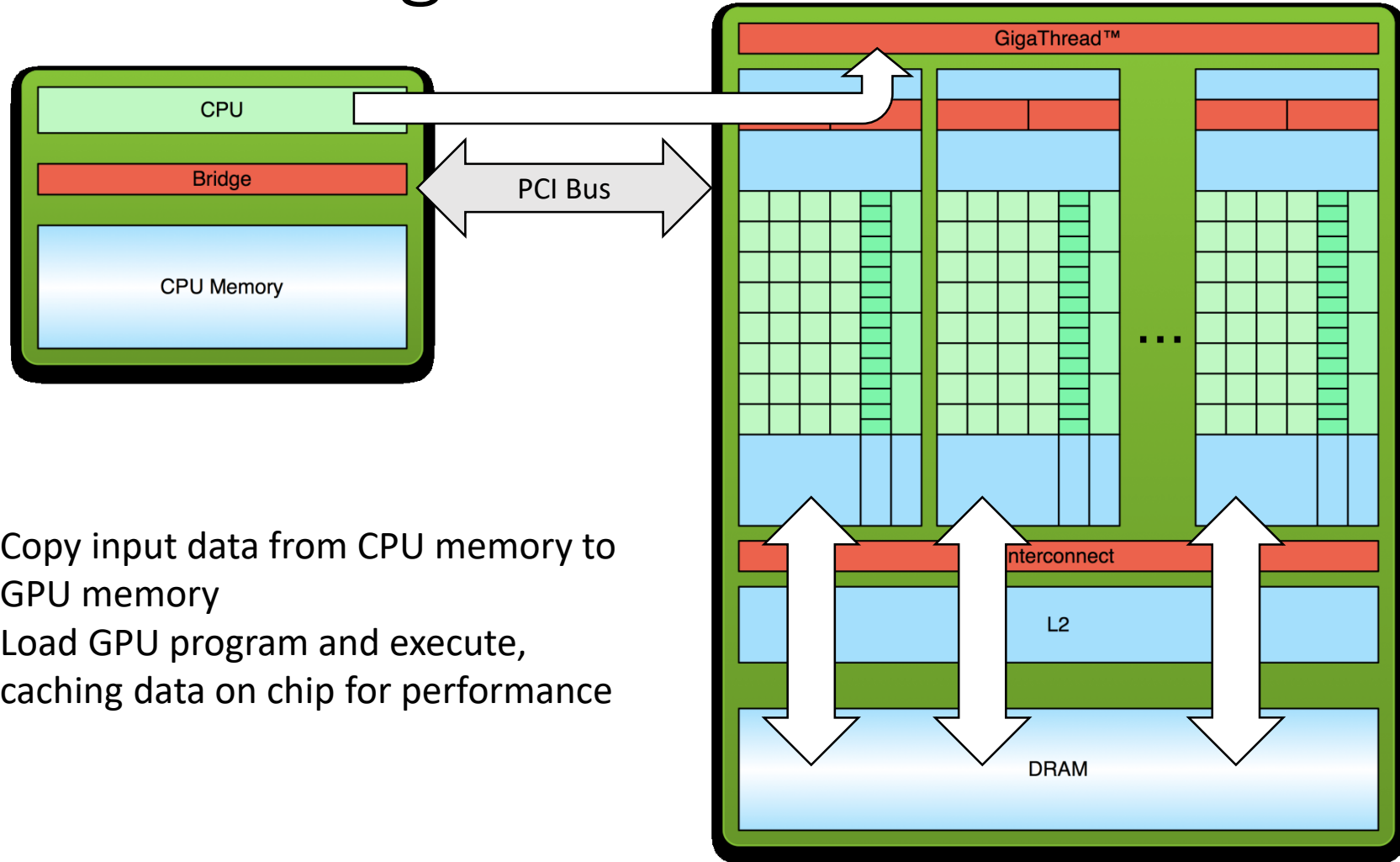
serial code



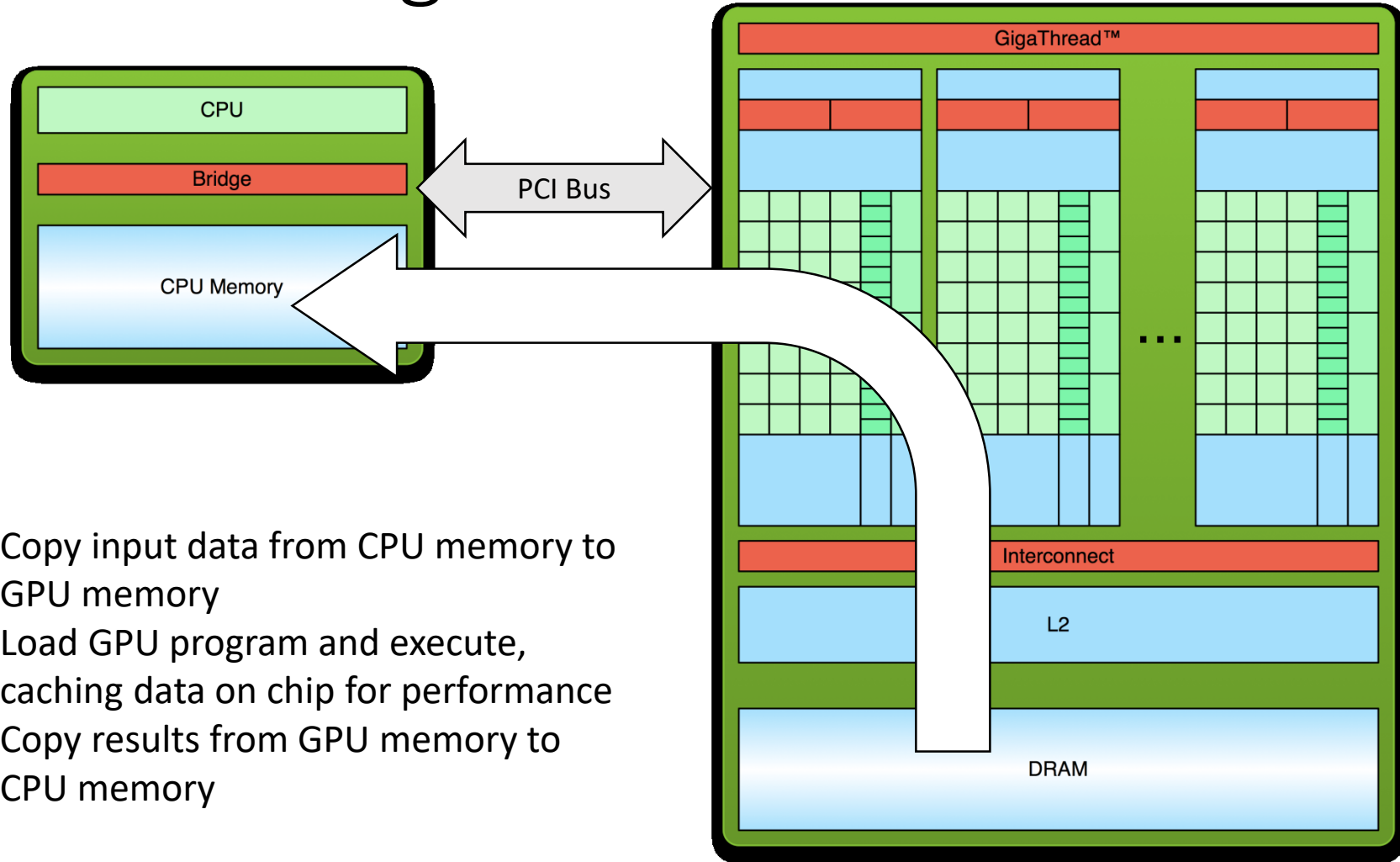
Simple Processing Flow



Simple Processing Flow



Simple Processing Flow



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;  
    }  
}
```

- Two new syntactic elements...

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>>>();
```

- 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;  
}
```

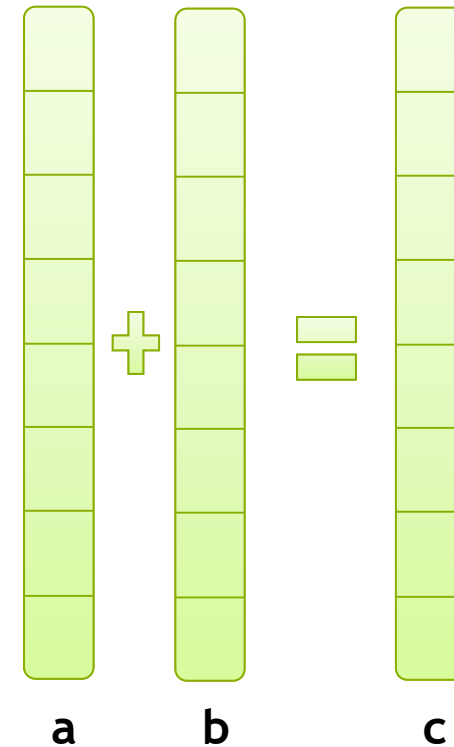
- `mykernel()` does nothing, somewhat anticlimactic!

Output:

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

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;  
}
```

- 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;  
}
```

- `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

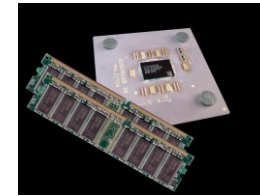
May be passed to/from host code

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



- 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  
    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);

// 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

Moving to 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
- 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];  
}
```

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

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
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
    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);

// 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;
}
```


Review

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- `__global__` declares device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function
- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch `N` copies of `add()` with `add<<<N,1>>>(...)` ;
 - Use `blockIdx.x` to access block index

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*:

```
__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;
}
```

COMBINING THREADS AND BLOCKS

CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

__syncthreads()

Asynchronous operation

Handling errors

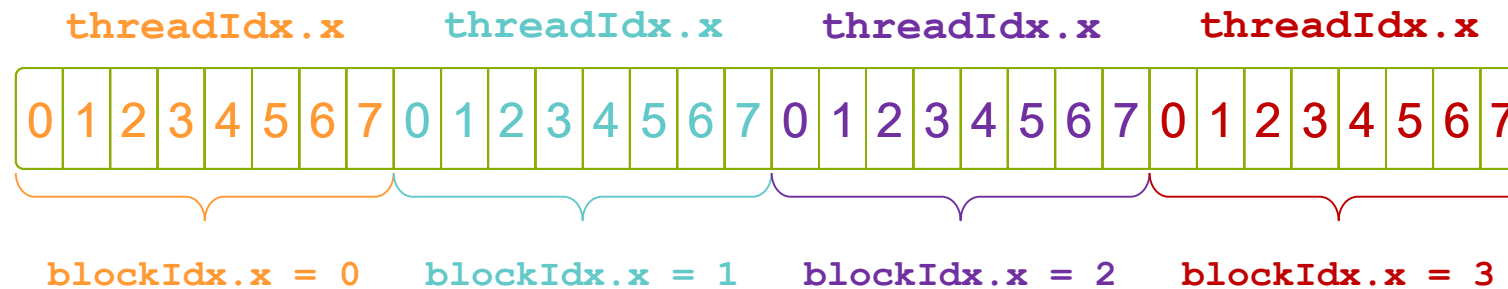
Managing devices

Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Index an array with one elem. per thread (8 threads/block)

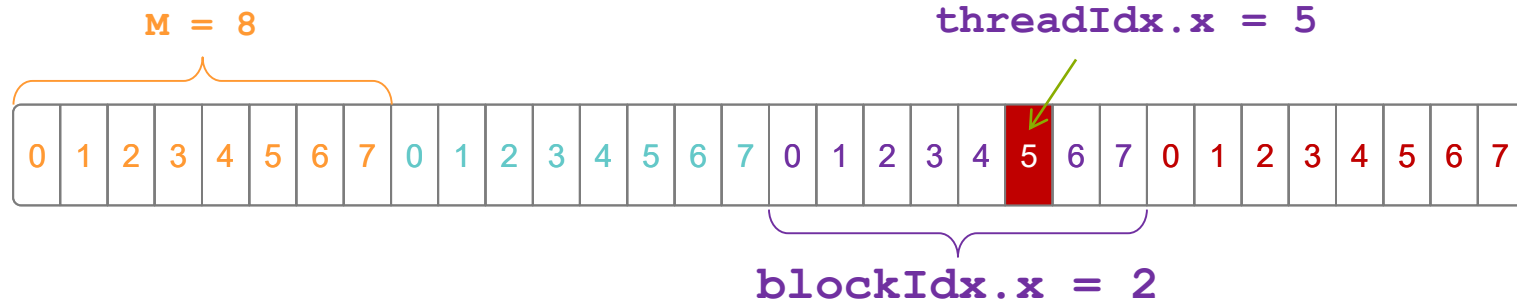
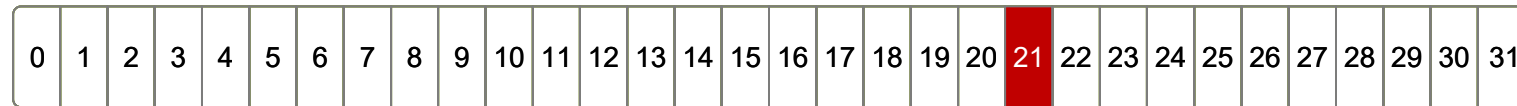


- With M threads/block, unique index per thread is :

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


Indexing Arrays: Example

- Which thread will operate on the red element?



```
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;
```

- 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;
```

```
}
```

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?
- 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

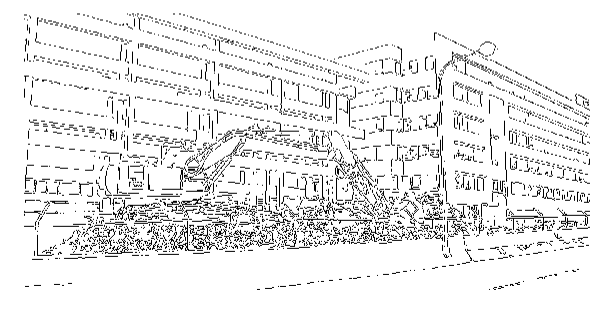
Handling errors

Managing devices

Stencils

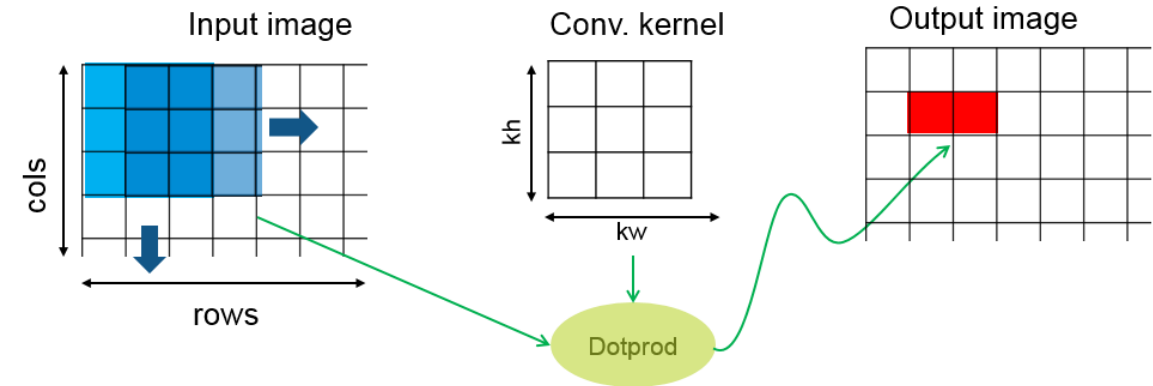
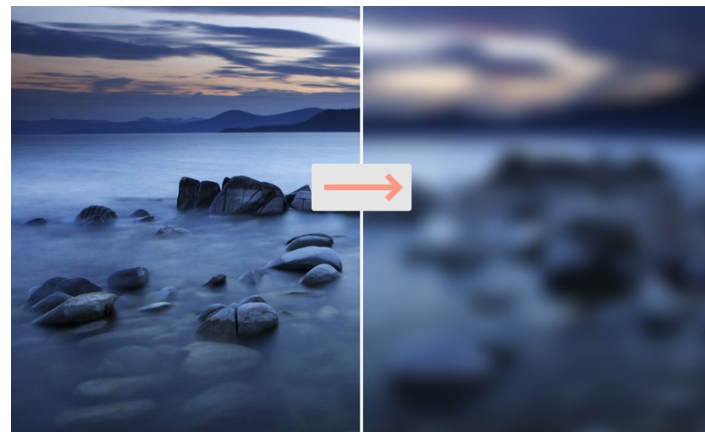
- Each pixel \rightarrow function of neighbors
- Edge detection:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$



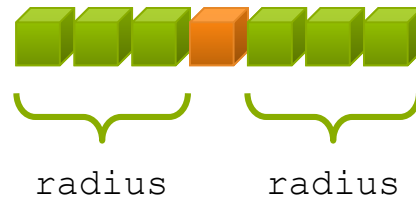
- Blur:

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
- Radius == 3 \rightarrow each output element is sum of 7 input elements:



Implementation within a block

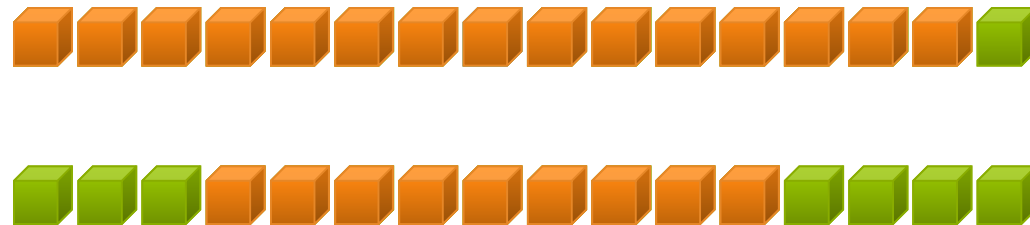
- Each thread: process 1 output element
 - blockDim.x elements per block
- Input elements read many times
 - With radius 3, each input element is read seven times

```
__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;  
}
```



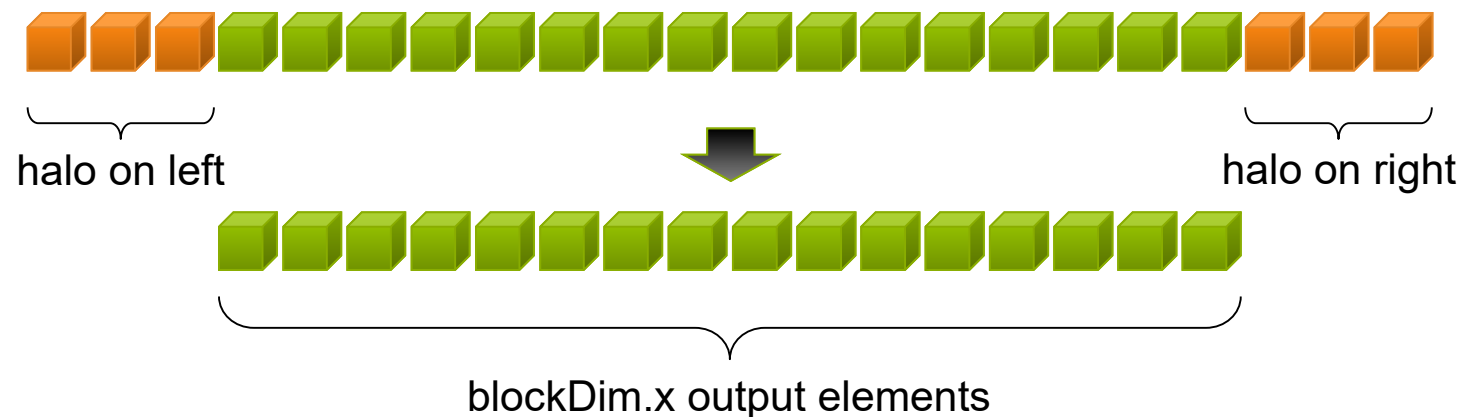
Why is this a problem?

Sharing Data Between Threads

- 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 ($\text{blockDim.x} + 2 * \text{radius}$) 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) {
    __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?

__syncthreads()

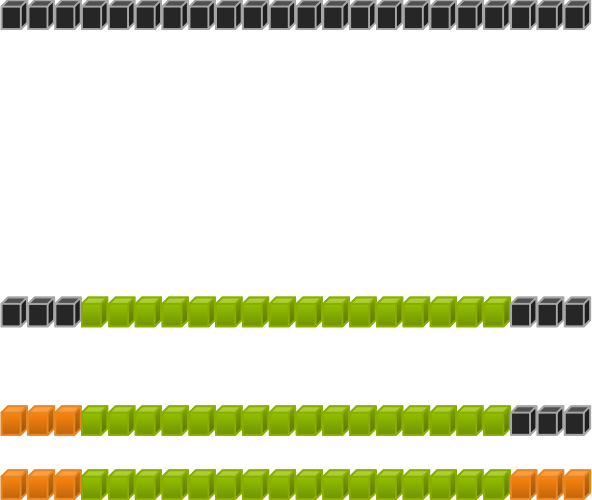
- `void __syncthreads ();`
- Synchronizes all threads within a block
 - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
 - In conditional code, the condition must be uniform across the block

Correct Stencil Kernel

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

```
__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

```
__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 uninterruptible – **atomic**

`atomicAdd()`

`atomicSub()`

`atomicMin()`

`atomicMax()`

`atomicInc()`

`atomicDec()`

`atomicExch()`

`atomicCAS()`

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)

- 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

Questions?