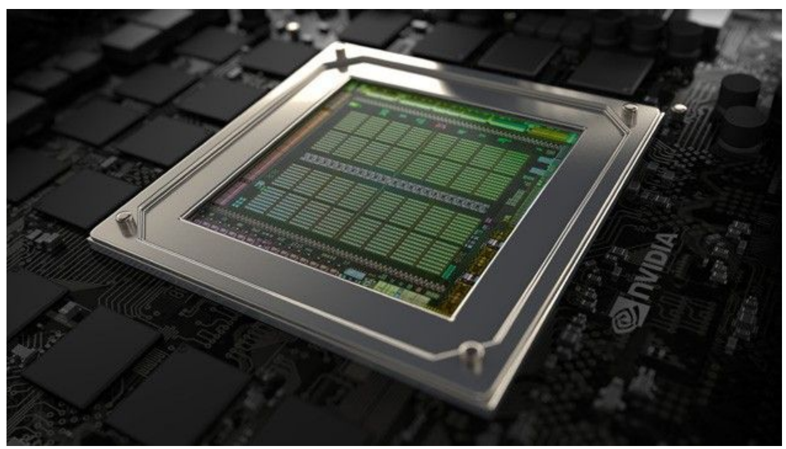
Parallel Computing with GPUs

Wayne Franz
Comp 4510
Nov. 2016

Schedule

Week 1	GPU Fundamentals
1. (Mon)	Introduction to GPUs
2. (Wed)	GPU Architecture
(Fri)	(Remembrance Day)
Week 2	**Solving Problems on the GPU**
3. (Mon)	Programming in CUDA
4. (Wed)	Case Study: Parallel Sum Reduction
5. (Fri)	Case Study (Cont'd)

1. Introduction to GPUs

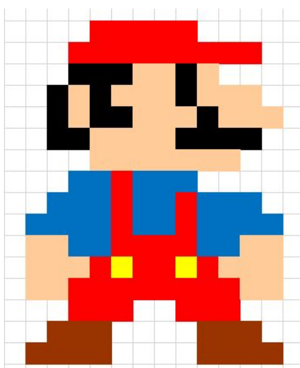


Graphics Rendering

- Standard (Wikipedia) definition:

  "The process of generating an image from a 2D or 3D model."

  - Eg. (2D):



colours →

0xfff fff	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xfff fff	0xfff fff	0xfff fff
0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000	0xff0 000
0x000 000	0x000 000	0x000 000	0xffc 0c0	0xffc 0c0	0x000 000	0xffc 0c0	0xfff fff	0xfff fff

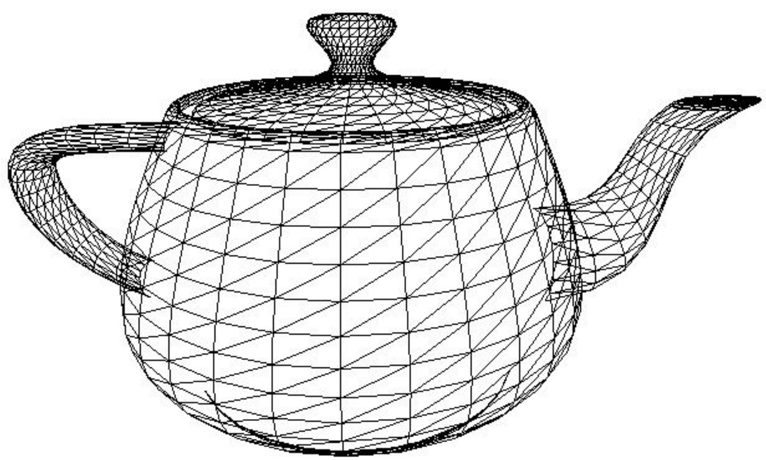
...

Image ← - - - - - - - - - - - - - - - - Model

Rendering

Graphics Rendering

- Eg. (3D)



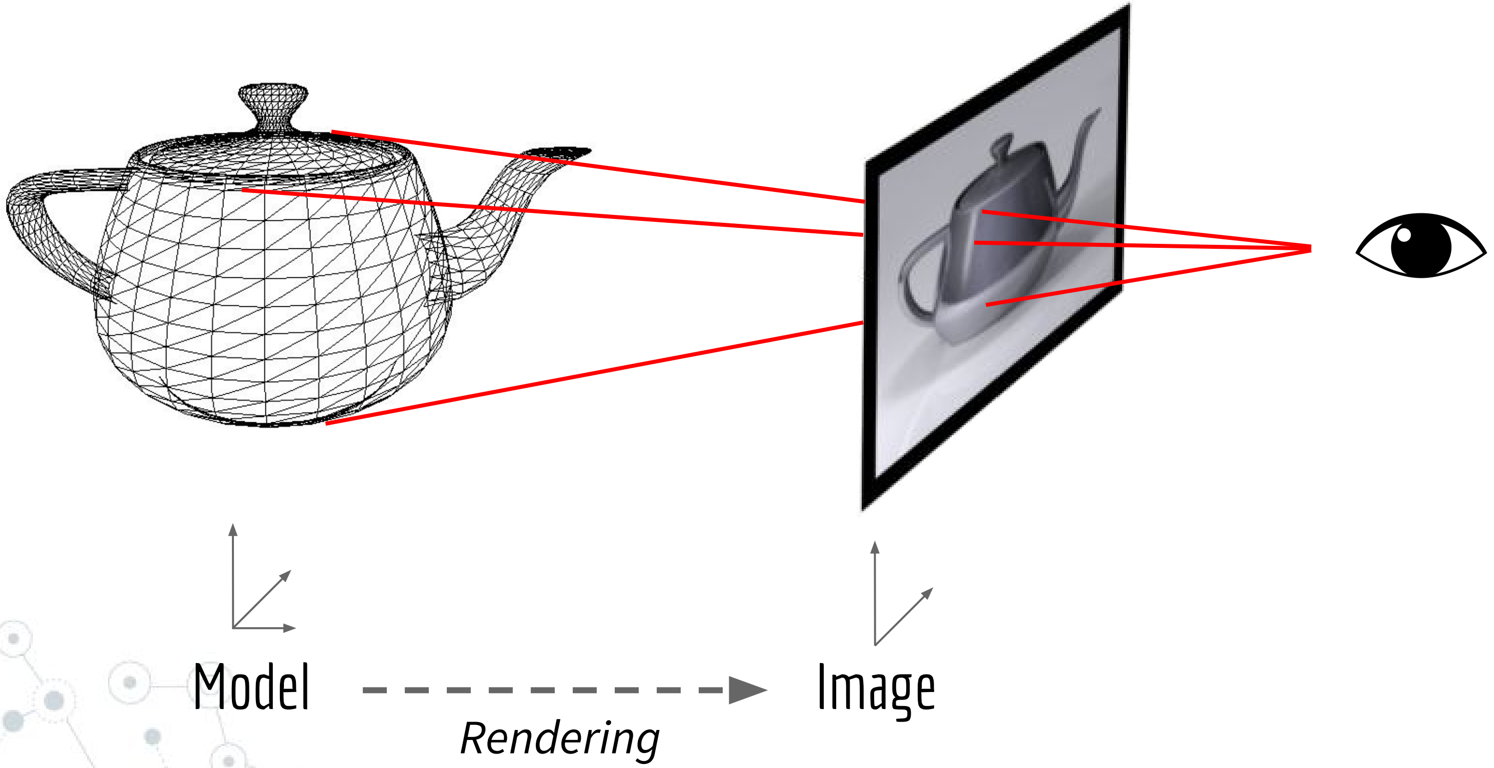
vertices →

(1,4,7)	(2,3,6)	(9,2,5)

Model

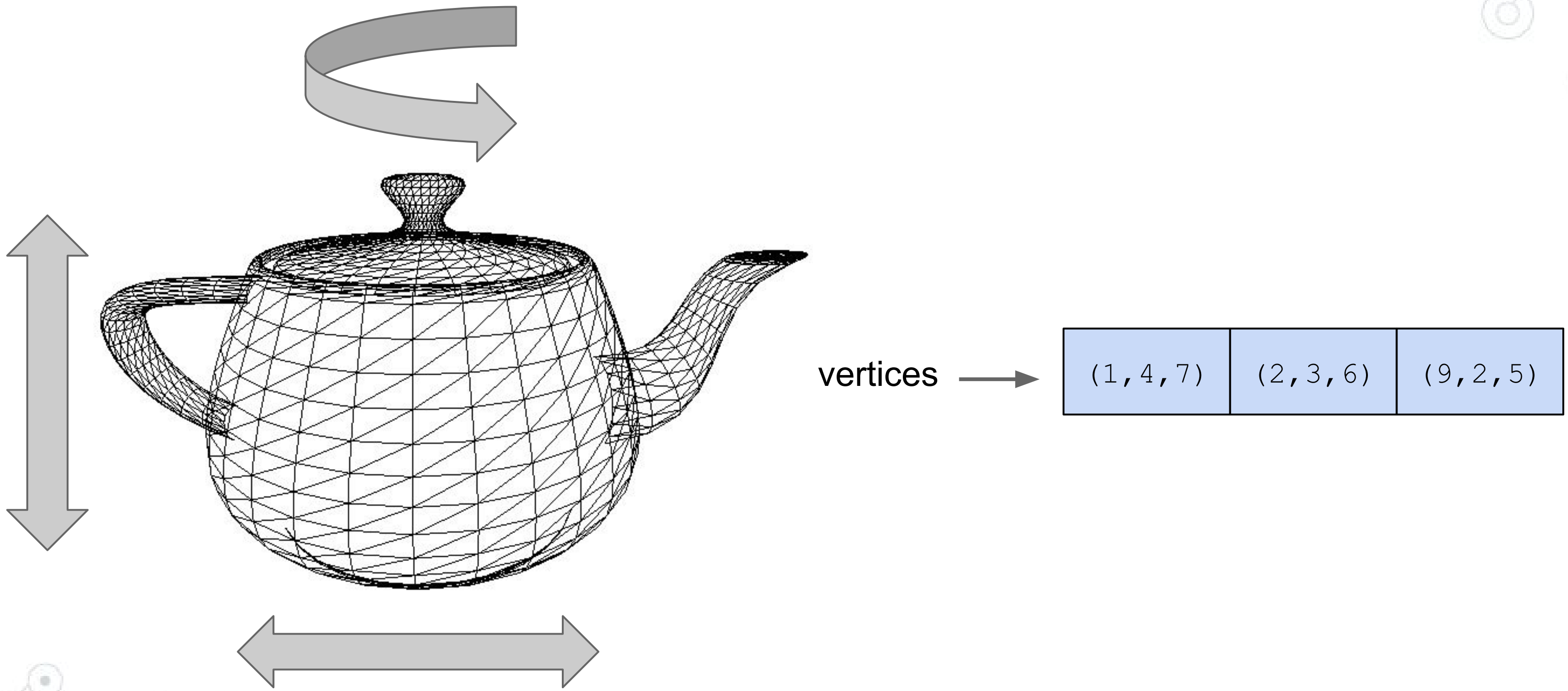
Graphics Rendering

- Projection



Graphics Rendering

- Rotation, translation



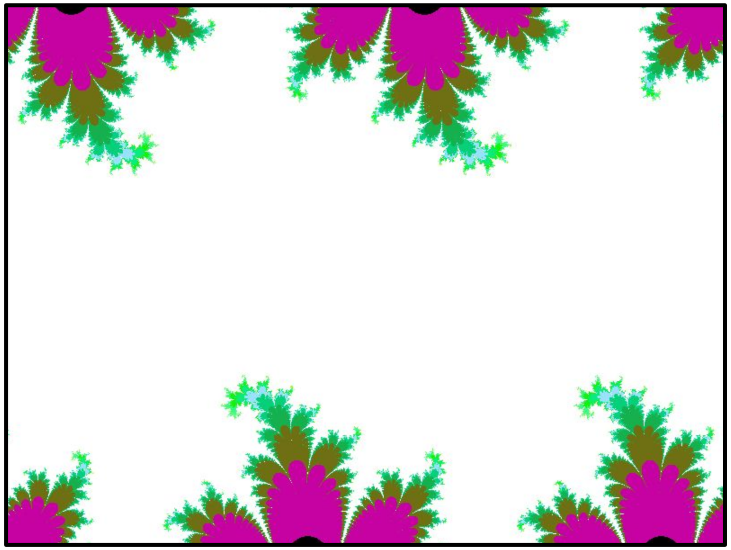
Graphics Rendering

- Historically done on the CPU
  - **Problems:**
    - Computationally intensive
    - CPU also does other things
    - Requires a hard deadline: refresh rate
  - **Solution:**
    - Use an accelerator!

Graphics Rendering

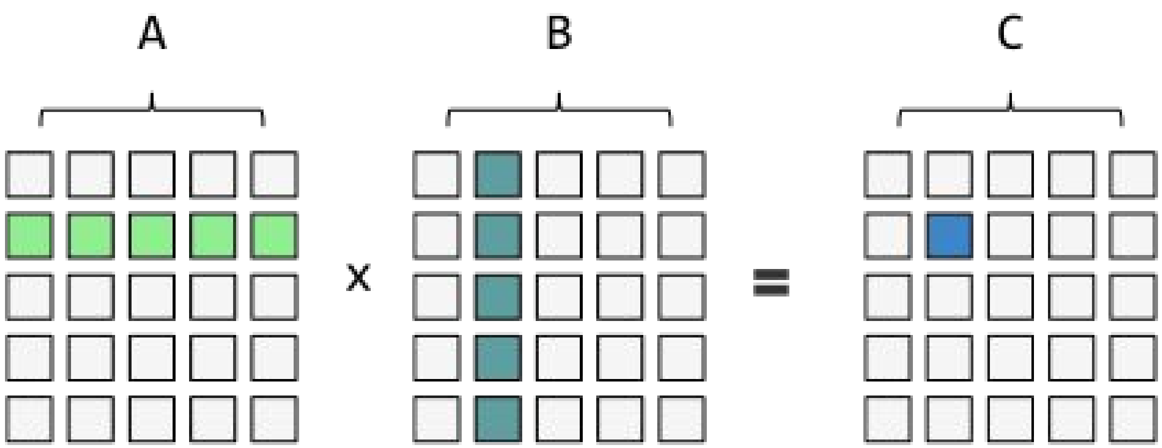
- Graphics rendering tasks:
  - translation
  - rotation
  - projection

→ *data-parallel*



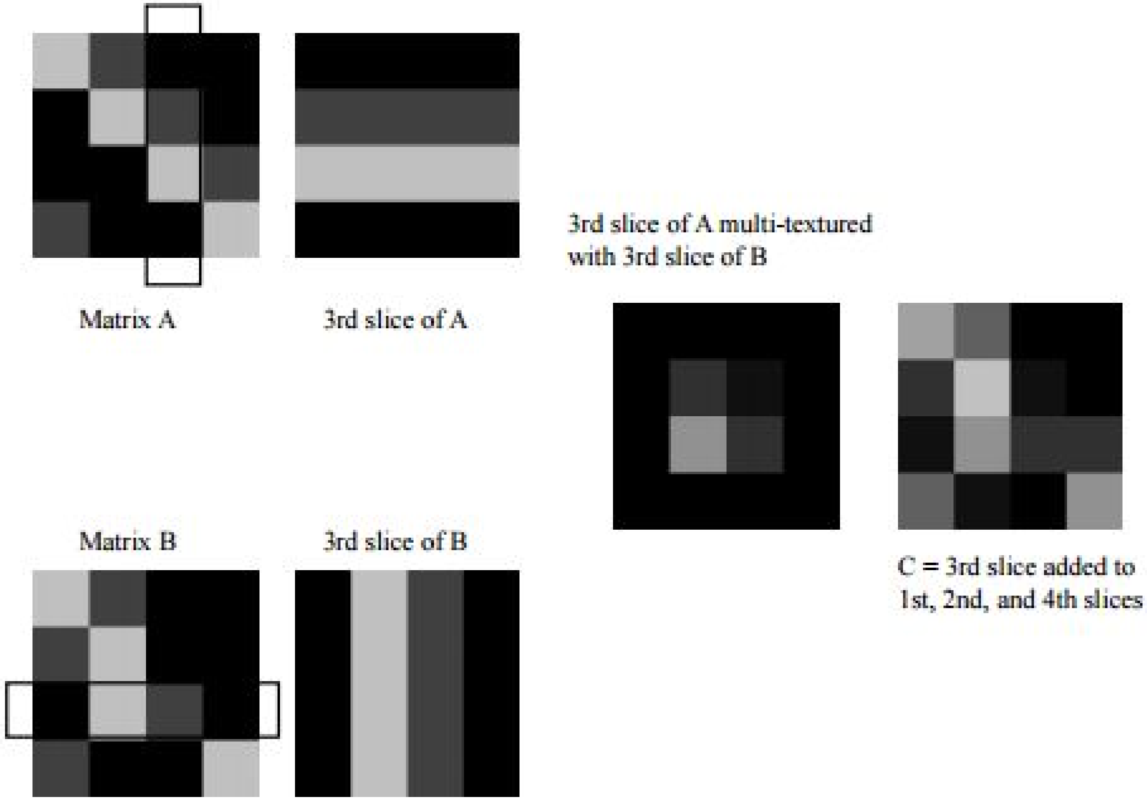
History

- GPUs have evolved to accelerate *data-parallel* computation
- Data-parallelism can be found in areas other than just graphics...



History

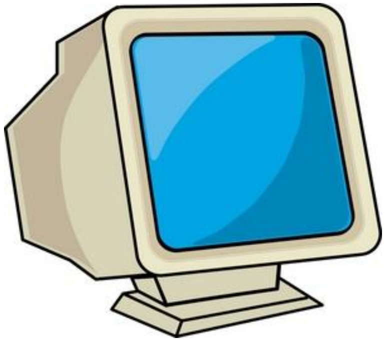
- General Purpose GPU Computing (GPGPU)
  - Larson & McCallister, *Fast Matrix Multiplies using Graphics Hardware*, 2001



History

- Results:
  - Less than stellar…
  - Data transfer time > Compute time

- Take-away:
  - Obtaining any benefit from the GPU requires *programming with the hardware in mind*.



History

“Compute Unified Device Architecture” (CUDA)

- Nvidia, 2006
- C-like language
- Can operate on arrays instead of images / vertices
- Finer-grained control over GPU

Today

- Multiple platforms for GPGPU

NVIDIA CUDA

OpenCL

OpenACC
DIRECTIVES FOR ACCELERATORS

Today

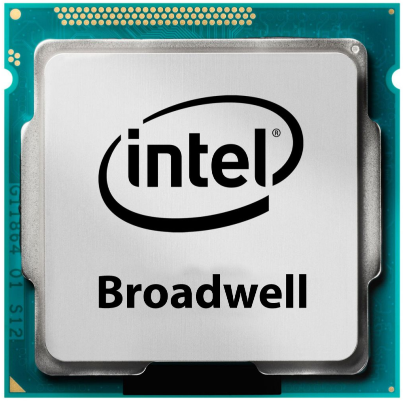
- GPU Architecture increasingly tailored for *general purpose* computation

- Eg. Nvidia P100

Today

Why should we care?

Device	Theoretical Max. Throughput (SP FP)
Intel Xeon (Broadwell) E5-2699 (v4)	~774 GFLOPS
Nvidia P100	~10,609 GFLOPS

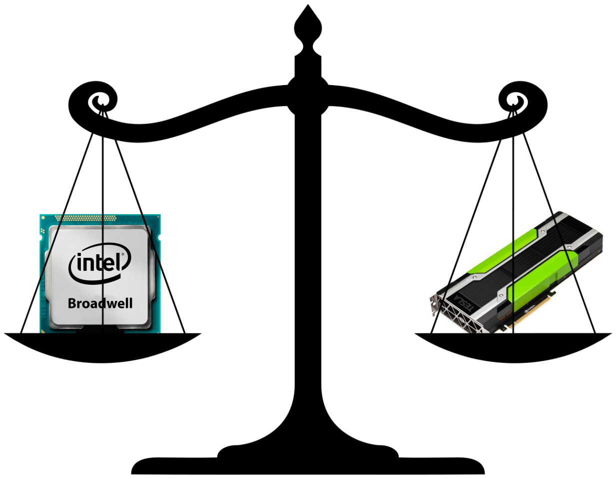


But ...

Trade-offs

- Sacrifices are made for the GPU’s throughput…
  - CPUs and GPUs are designed to do very *different things*.

~774 GFLOPS

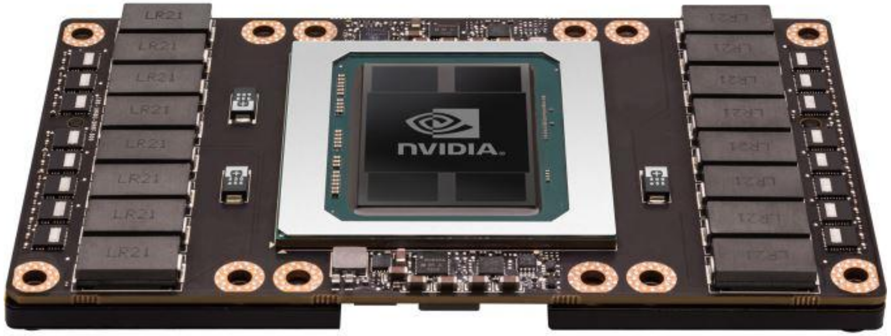


~10,609 GFLOPS

Trade-offs

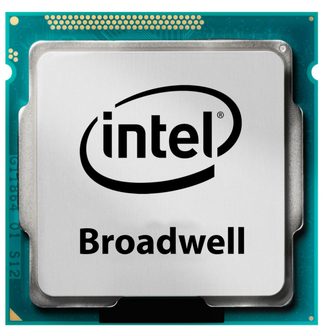
	CPUs	GPUs
Purpose	General purpose computing	Data-parallel (i.e. graphics) computing
Taxonomy	MIMD	SIMD
Strengths	• Multitasking (context switching) • I/O	• Throughput • Power efficiency (per FLOP)
Weaknesses	• Throughput (sort of…) • Memory wall (requires caches)	• Context switching • Branching • I/O

2. GPU Architecture



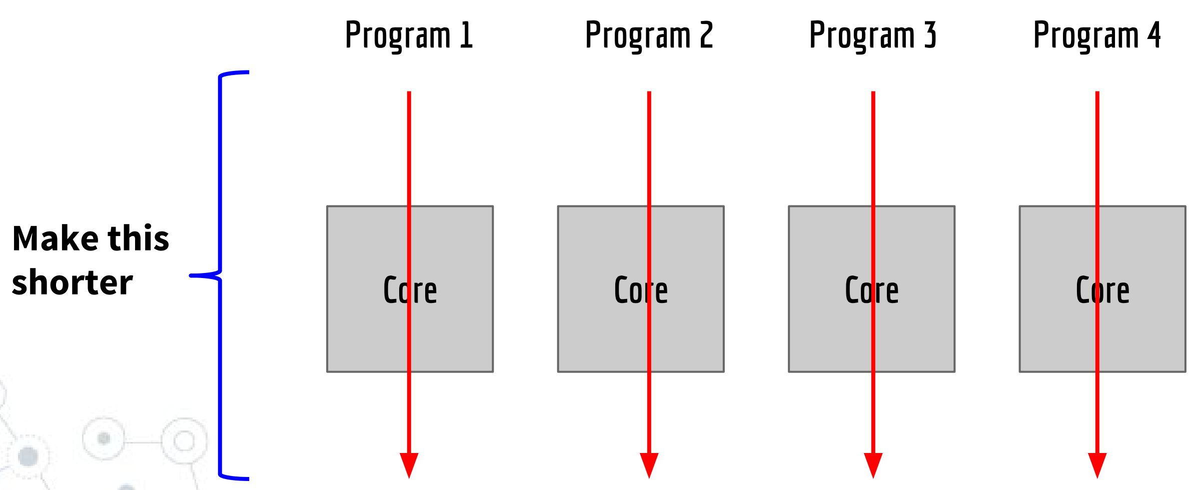
Architectural Synopsis

	Clock Rate	Cores	Memory Bandwidth	Cache levels	Power Requirements
Intel Xeon (Broadwell) E5-2699 (v4)	2.2 - 3.6 GHz	22	76.8 GB/s	L1: 64 KB (per core) L2: 256 KB (per core) L3: 55 MB (per CPU)	145 W
Nvidia P100	1.3 - 1.5 GHz	3584	720 GB/s	L1: 64 KB (per 64 cores) L2: 4096 KB (per GPU)	300 W

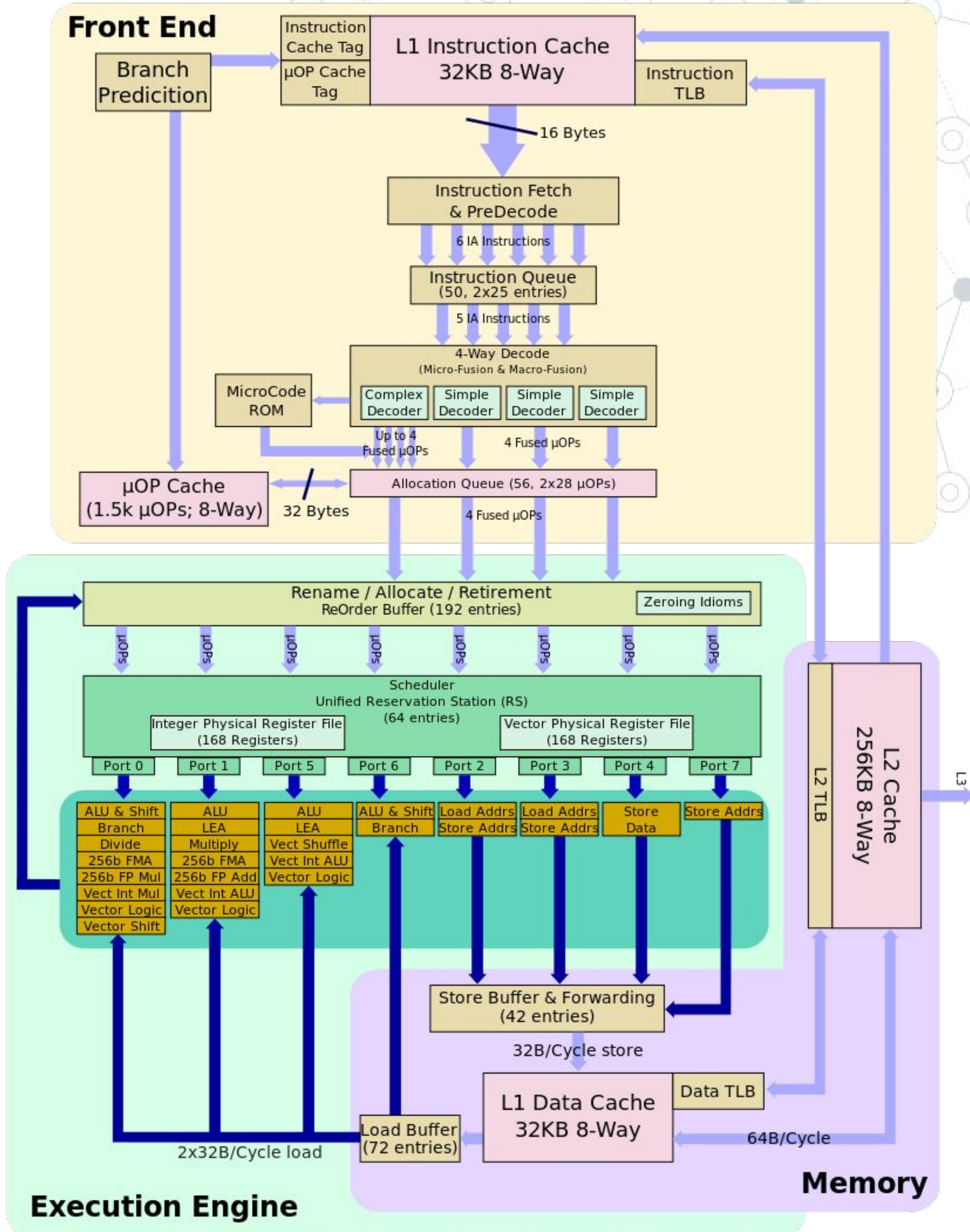


What’s in a core?

- Modern CPU cores
  - Like multiple “independent” sub-processors
  - Tailored for *high-frequency execution of multiple, independent tasks*
- Several large, complex cores:

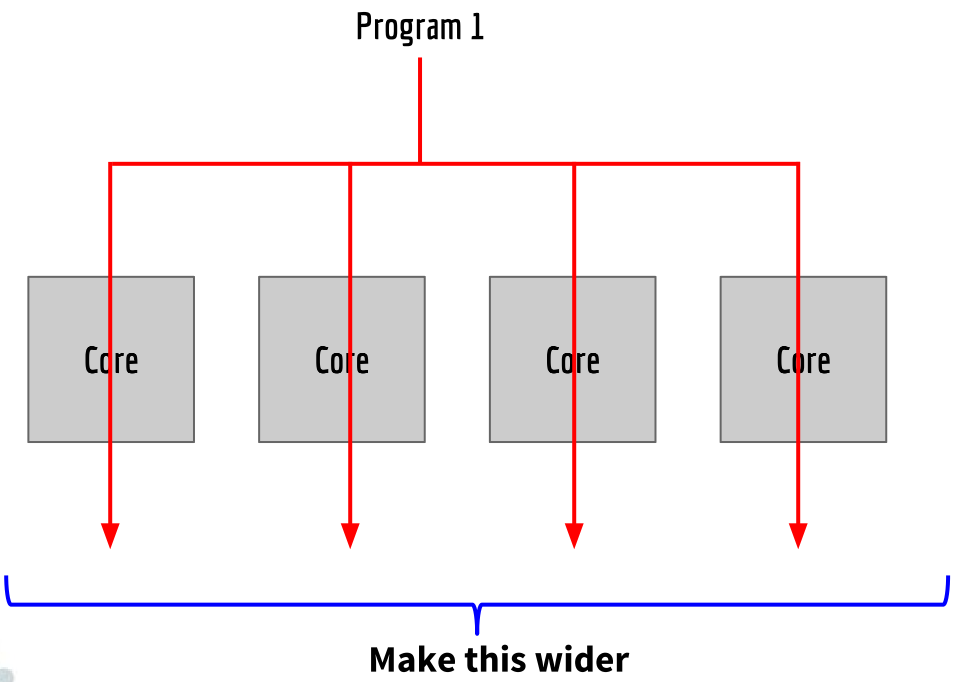


Intel Broadwell

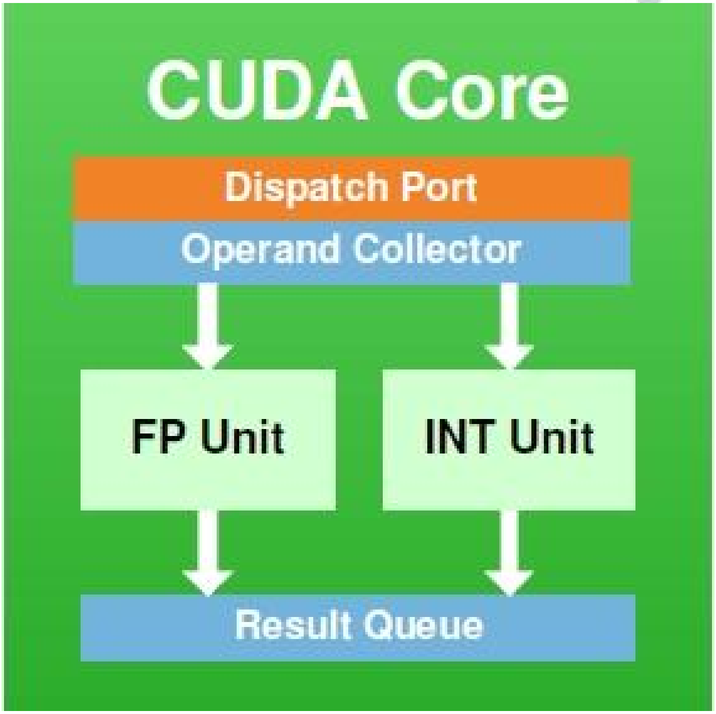


What's in a core?

- GPUs
  - Not much!
  - Tailored for exploiting the *maximum amount of parallelism in a single task*
- Many small, simple cores:



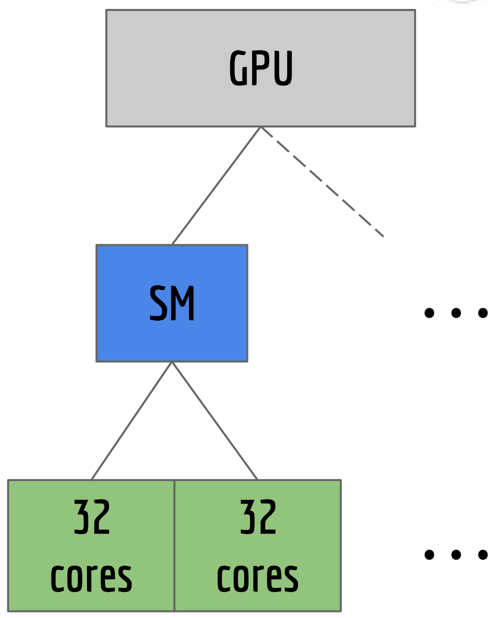
Nvidia P100



CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue

GPU Core organization

- Cores are grouped together
  - Groups of 32
  - Perform same instruction in lockstep
- Streaming Multiprocessors (SMs)
  - Contain 2 groups of 32 cores
  - The "instruction control unit"



Hardware Threads

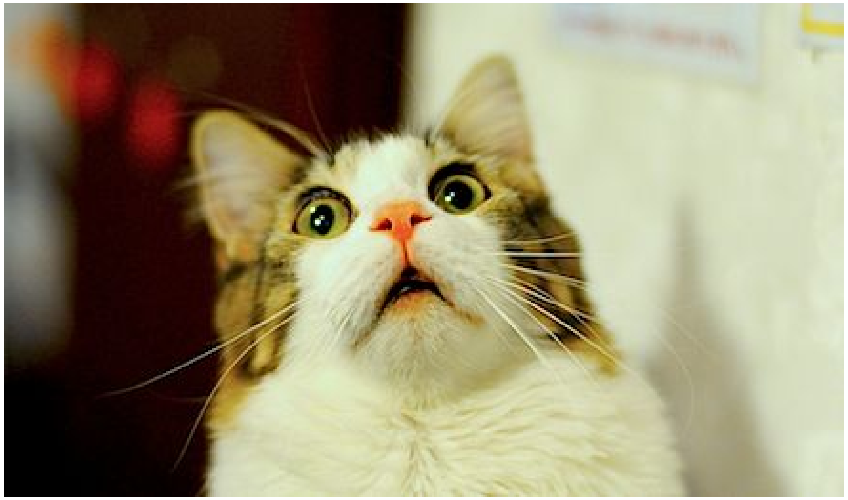
- Hardware threads are run in groups of 32
  - “***Warp***”

- Each SM has 2 x 32 cores
  - Can run 2 warps simultaneously

SM
Instruction Cache
Instruction Buffer
Warp Scheduler
Dispatch Unit
Register File (32,768 x 32-bit)
Core
DP Unit
LD/ST
SFU
Texture / L1 Cache
Tex
64KB Shared Memory

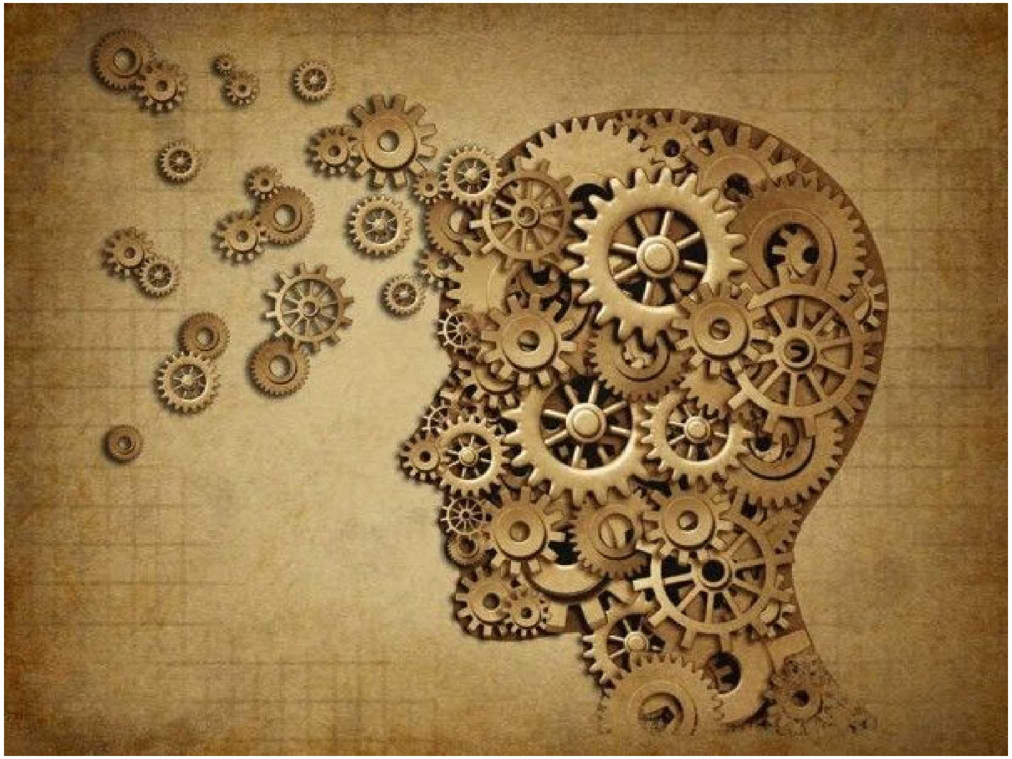
How many SMs?

The P100 has **56** SMs.



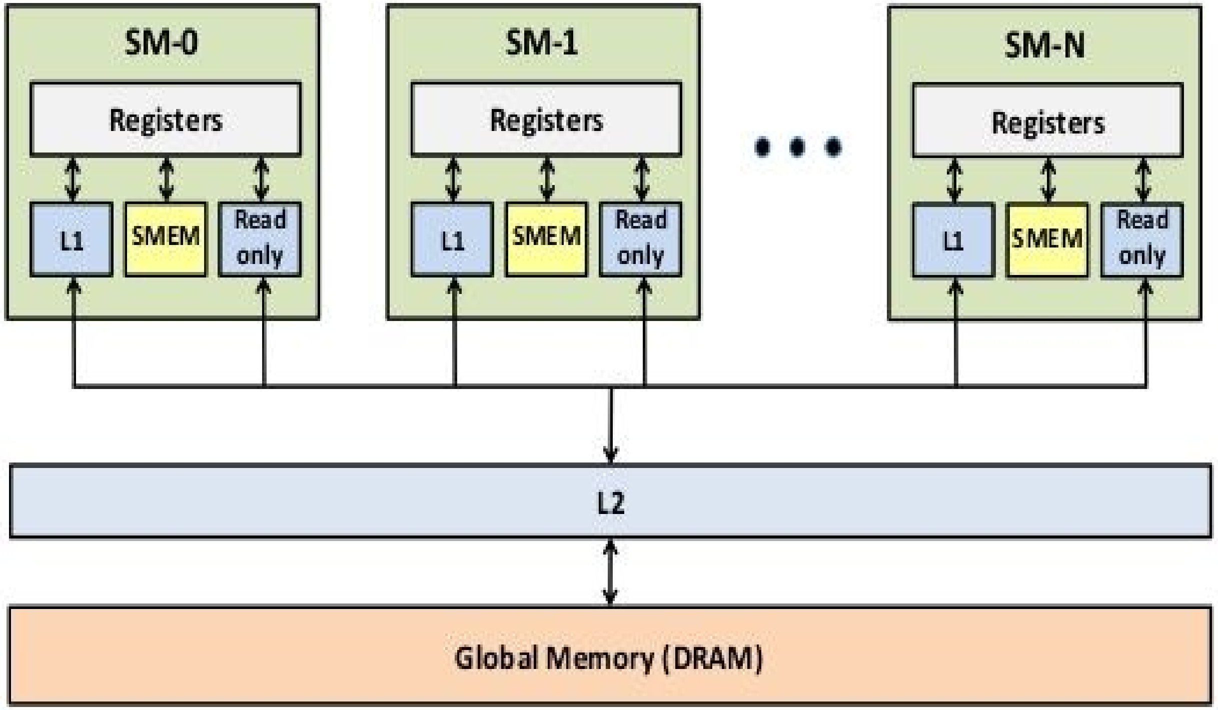

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

Memory System



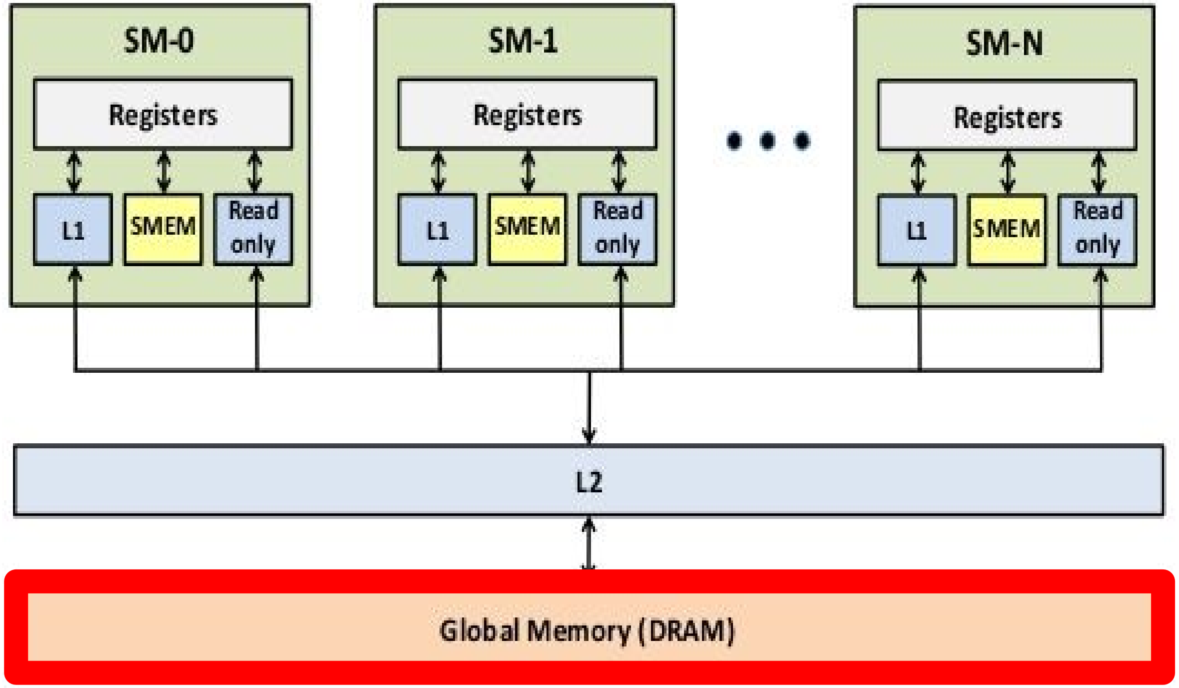
Memory System

- GPUs have several types of memory:
  1. Global
  2. Shared
  3. Constant/Texture
  4. Register



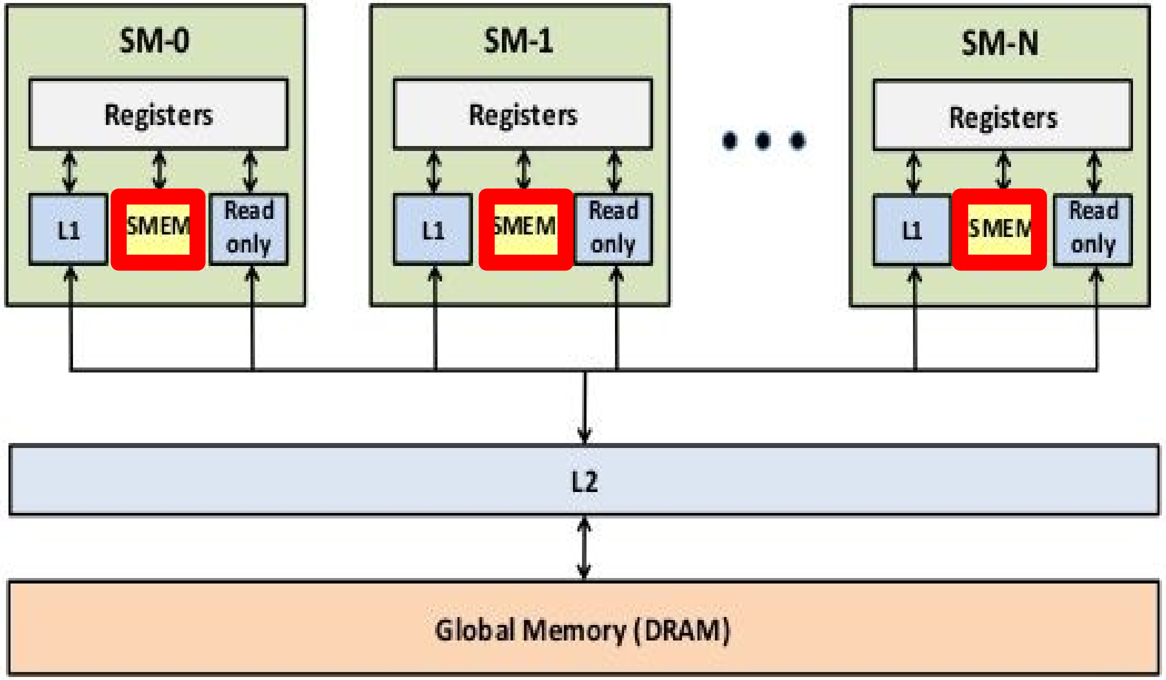
1. Global Memory

Capacity:	16 GB
Cache	L1, L2
Access:	GPU-wide
Latency:	200-400 cycles ● Most instructions take ~20-30



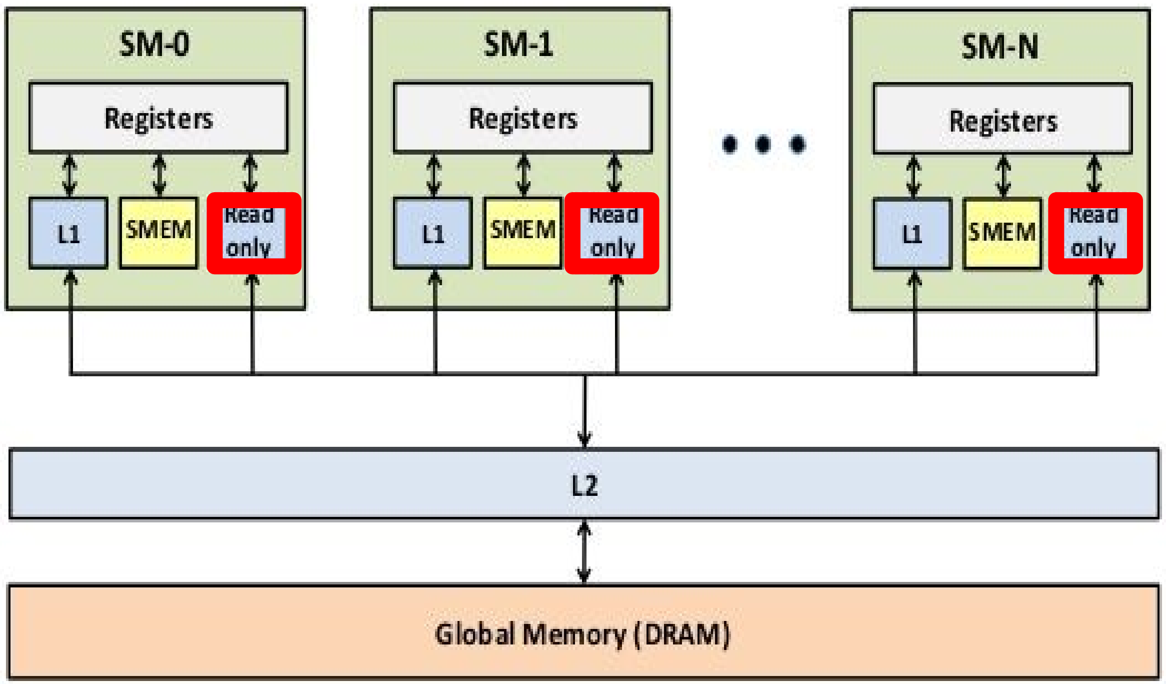
2. Shared Memory

Capacity:	64 KB / SM
Cache	None
Access:	SM-wide
Latency:	1 cycle (!)



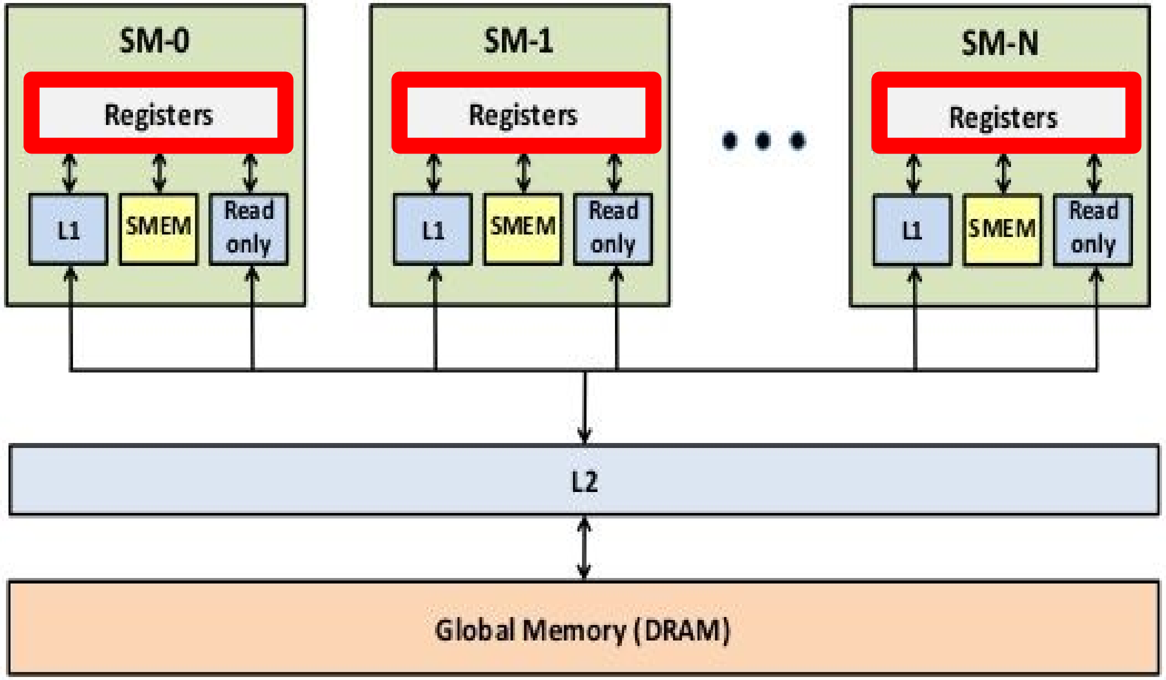
3. Constant Memory

Capacity:	64 KB / SM
Cache	Special cache
Access:	GPU-wide (but cached on each SM)
Latency:	1 cycle (hit) 200 - 400 cycles (miss)



4. Register Memory

Capacity:	256 KB / SM
Cache	None
Access:	Private to each thread
Latency:	1 cycle

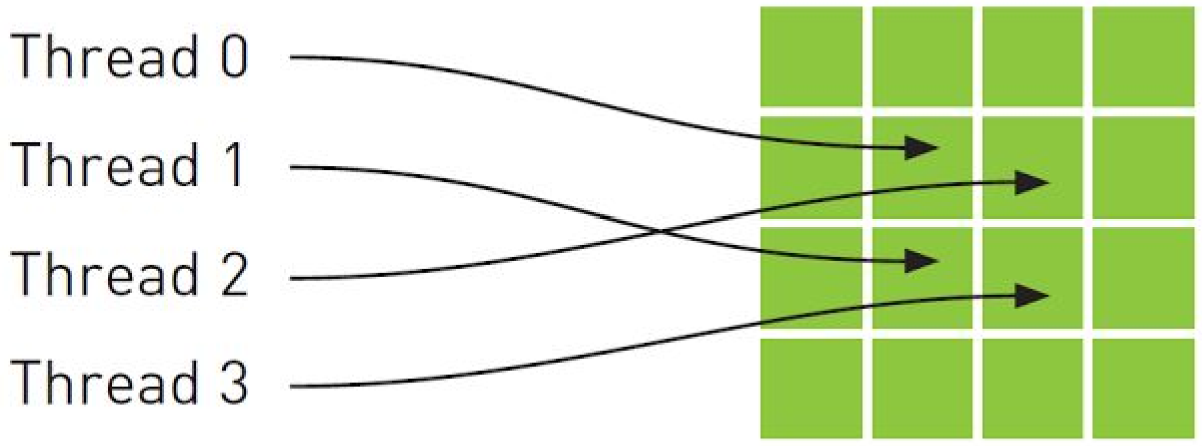


Accessing Memory

- *The way we access memory matters.*

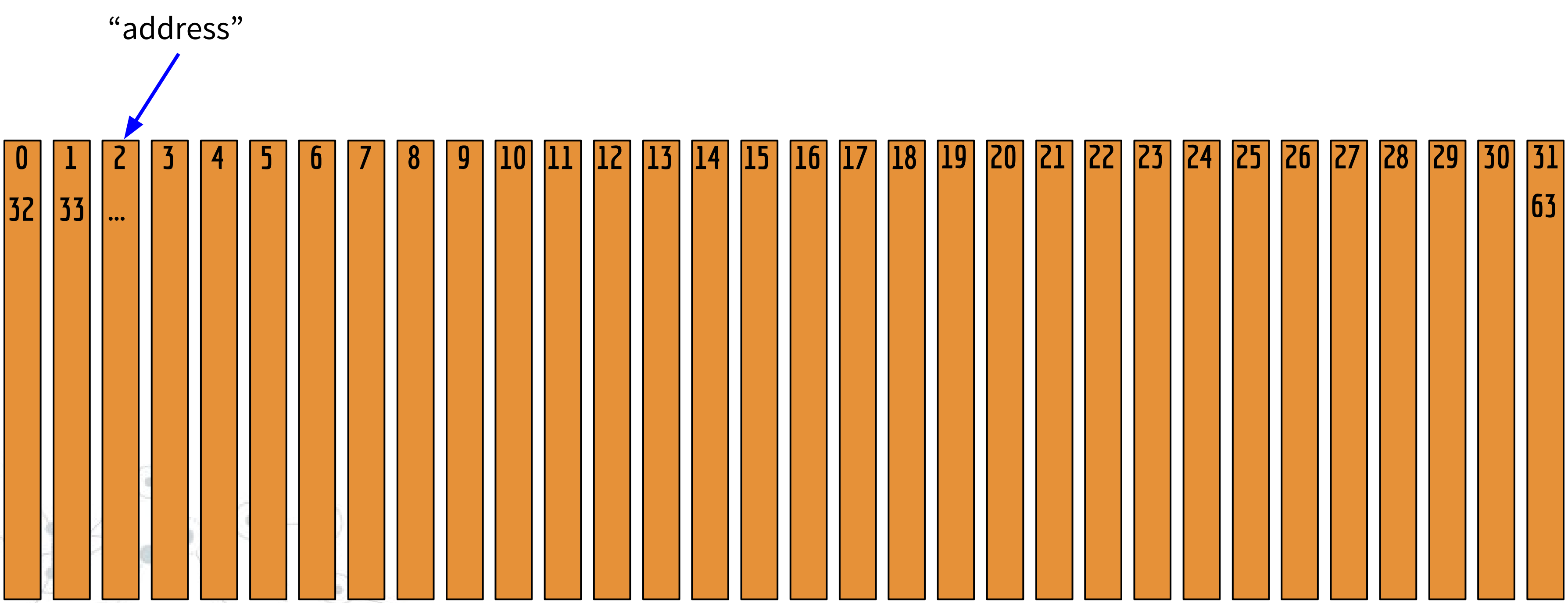
- On the GPU, we often work with arrays
  - If each thread reads one element...
  - ...lots of reads (at once!)

- How close together are these reads?
  - Spatial Locality...



Accessing Shared Memory

- Organized into 32 banks
- A bank can handle only 1 request at a time

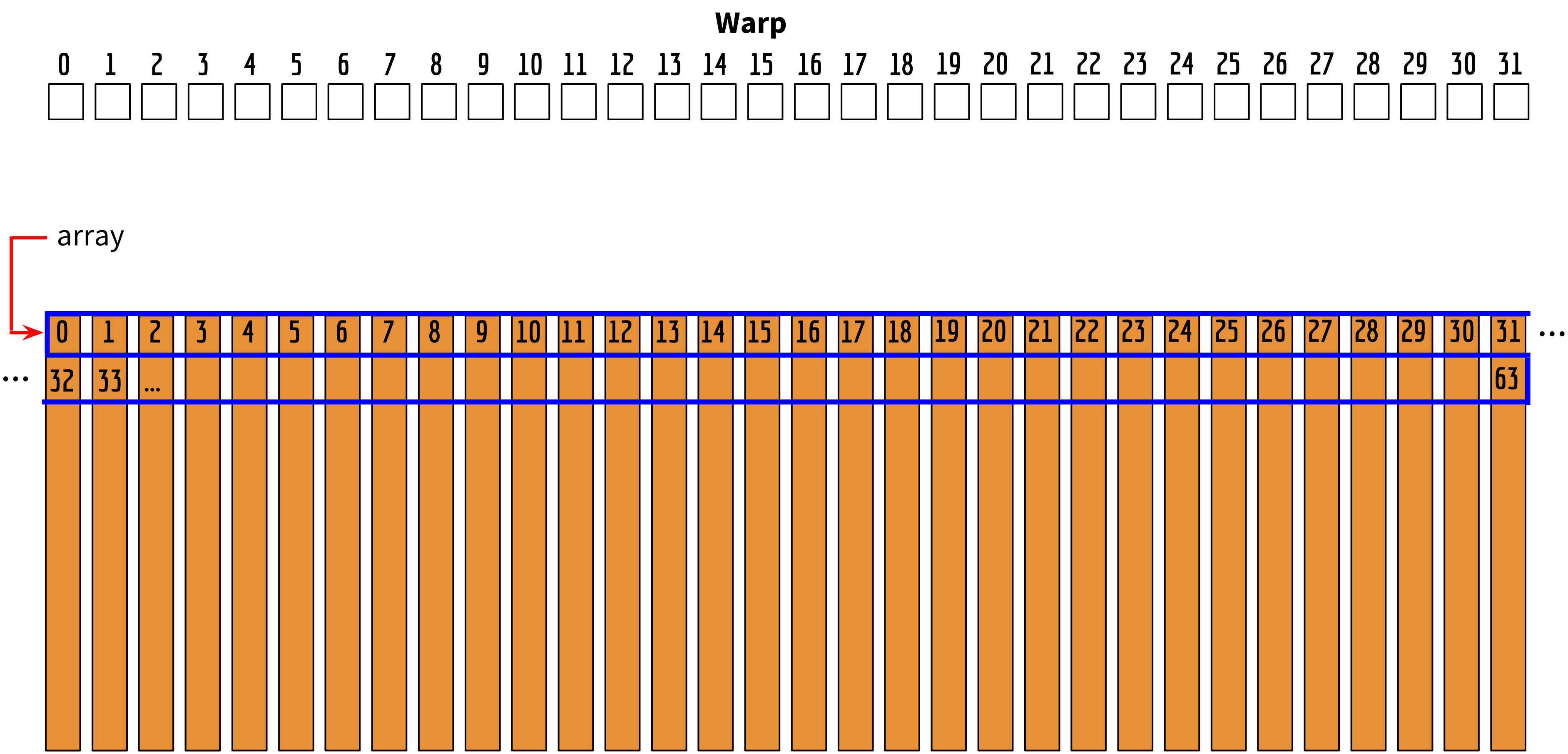


Accessing Shared Memory

- Suppose we have an array:

```
__shared__ float array[64];
```

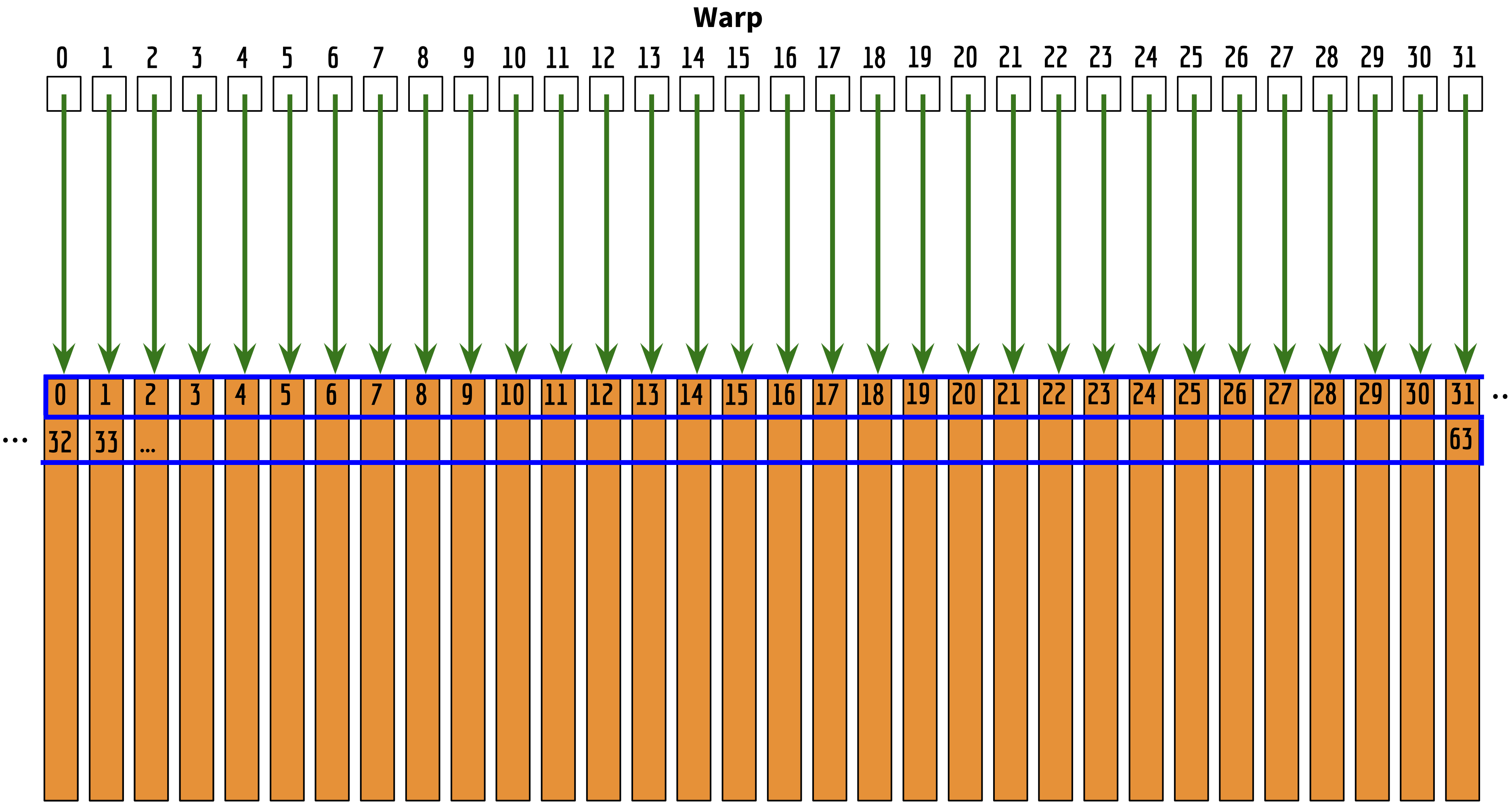
- Being accessed by a warp of threads:



Accessing Shared Memory

1. `float my_val = array[id];`

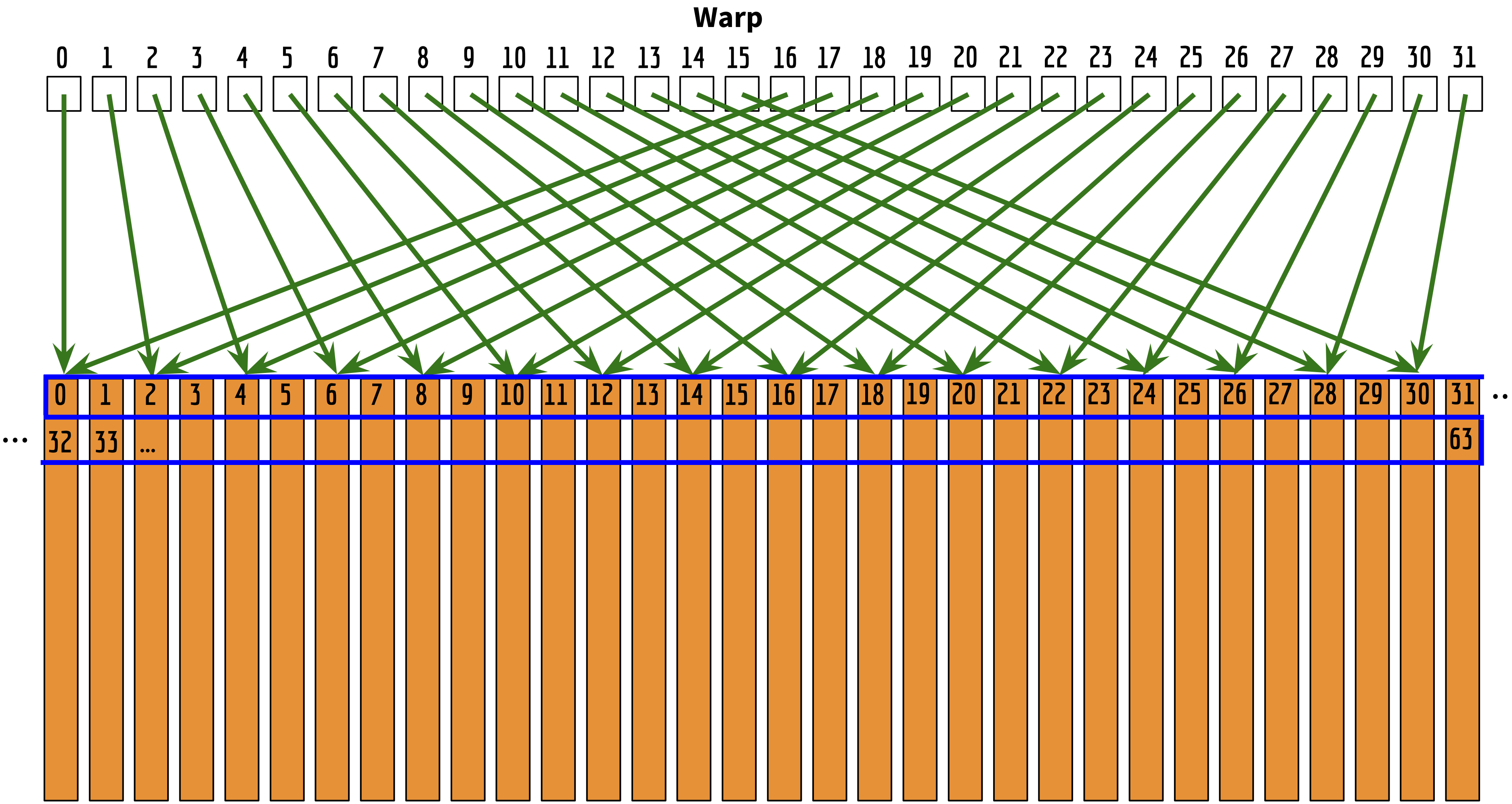
Best case:
32 values in 1 cycle!



Accessing Shared Memory

2. `float my_val = array[id * 2];`

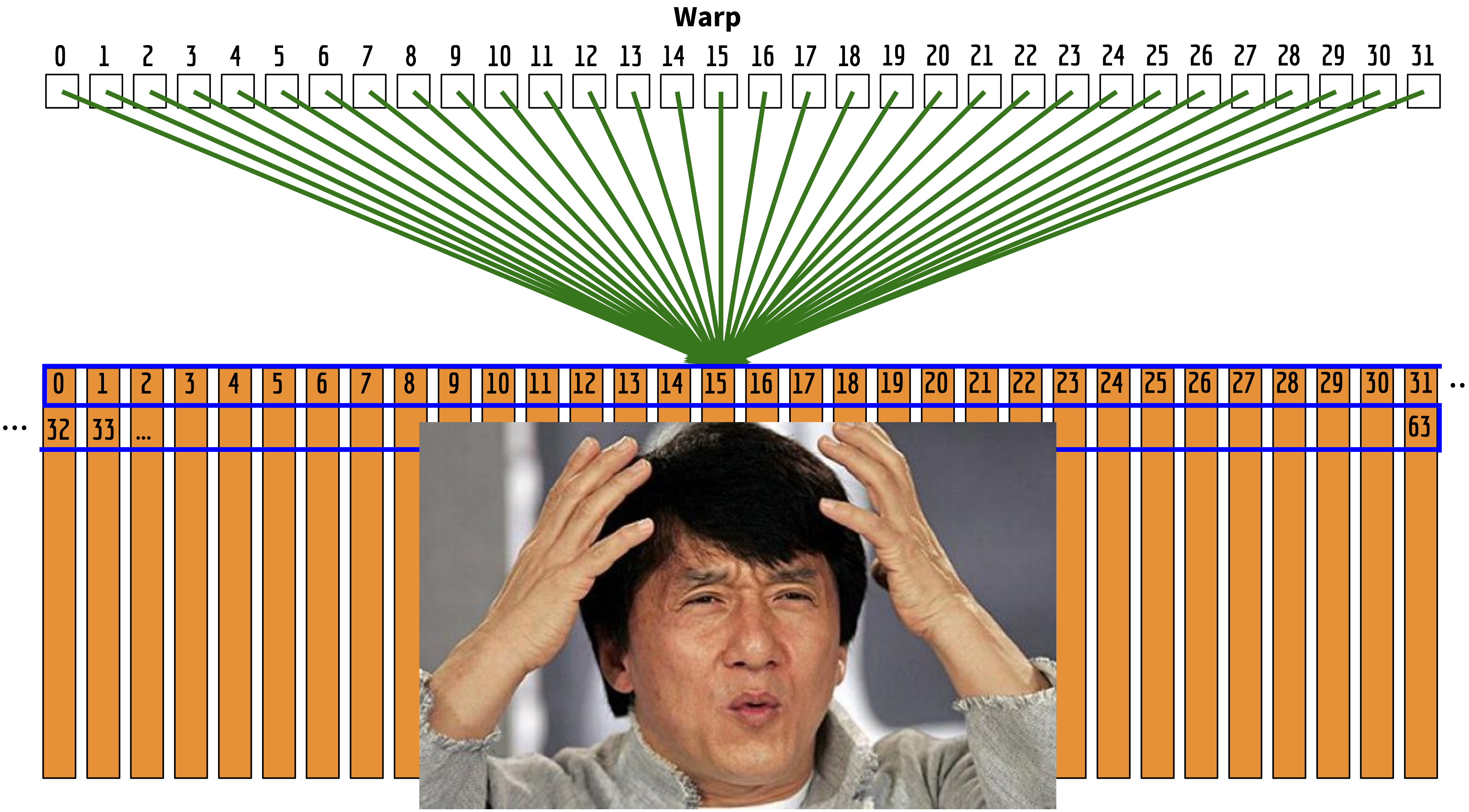
Bank conflict:
16 values on cycle 1
16 values on cycle 2



Accessing Shared Memory

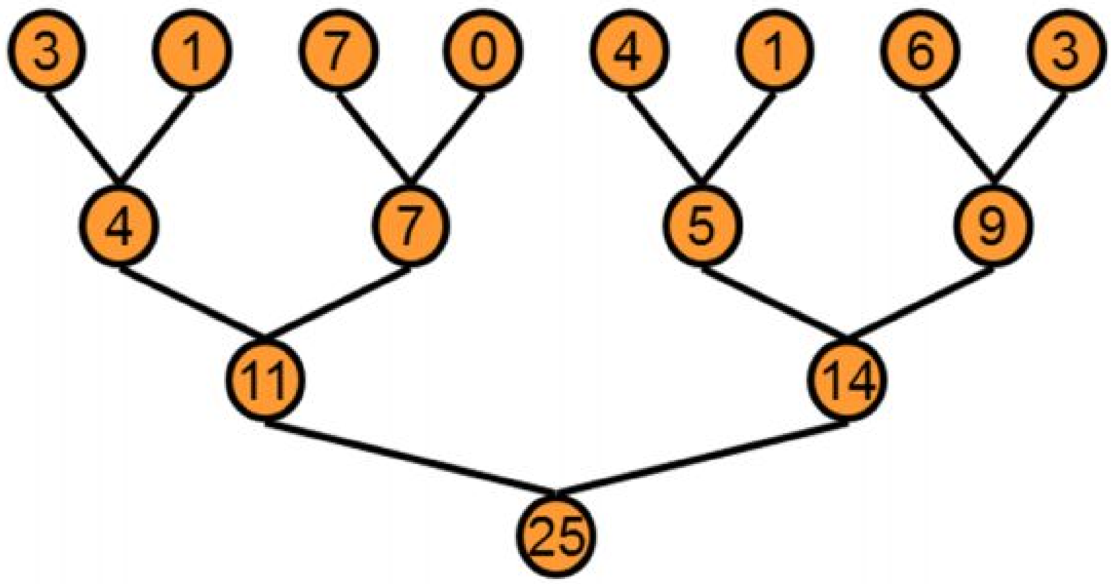
3. `float my_val = array[15];`

Broadcast feature:
32 bytes in 1 cycle.



Why?

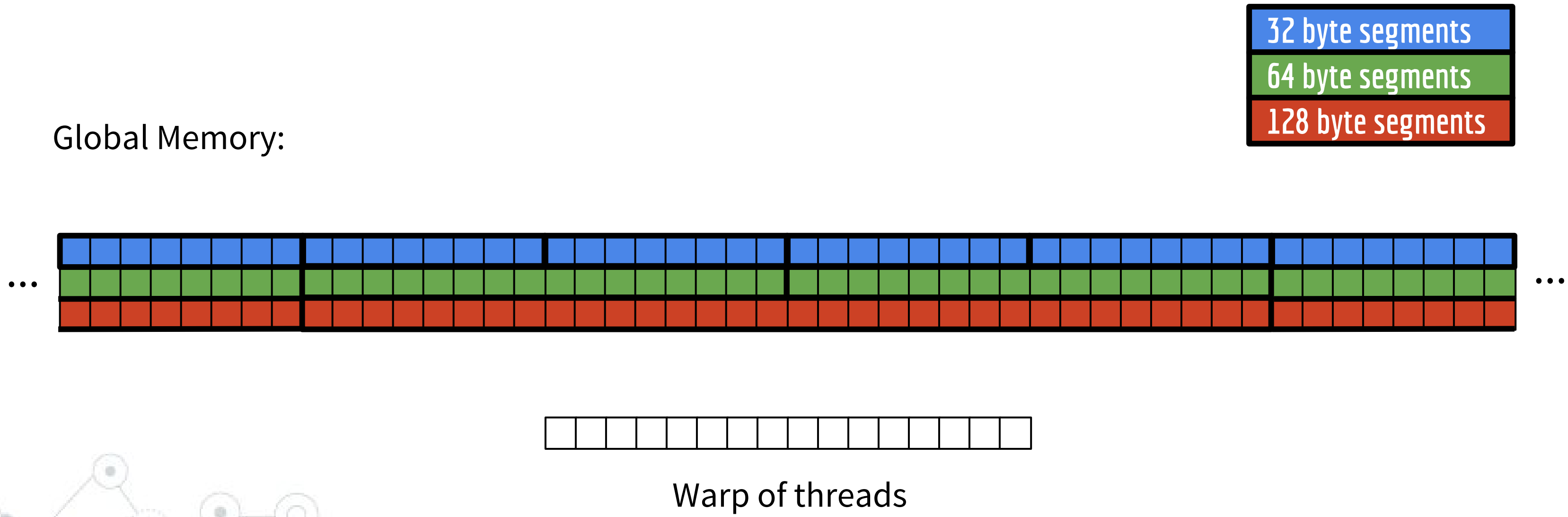
- Consider a parallel sum reduction:



- What if everybody needs the result?

Accessing Global Memory

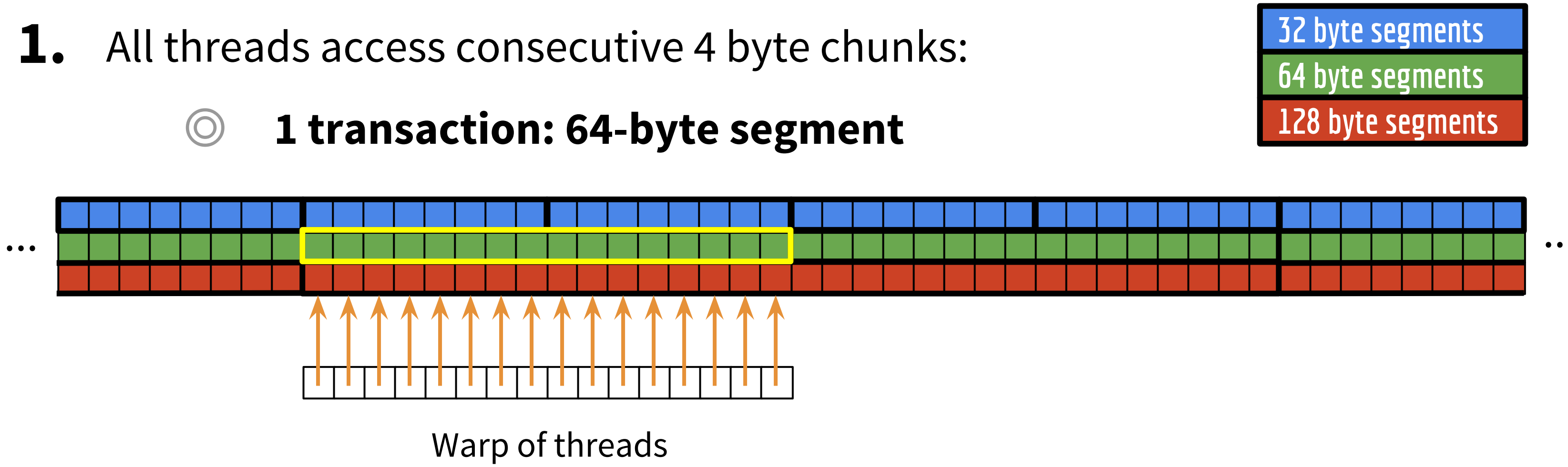
- Global memory is accessed in wide swaths:
  - 32, 64, or 128 byte segments



Accessing Global Memory

1. All threads access consecutive 4 byte chunks:

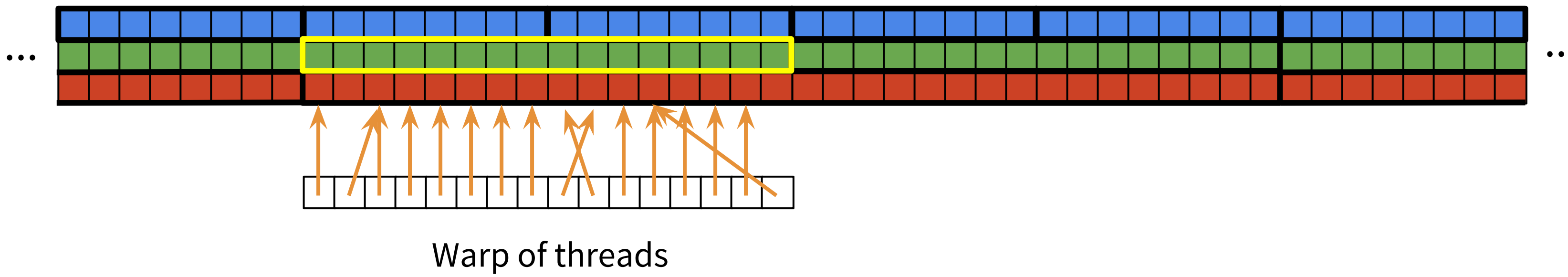
- **1 transaction: 64-byte segment**



Accessing Global Memory

2. All threads access 4 bytes out of order

◎ **1 transaction: 64-byte segment**



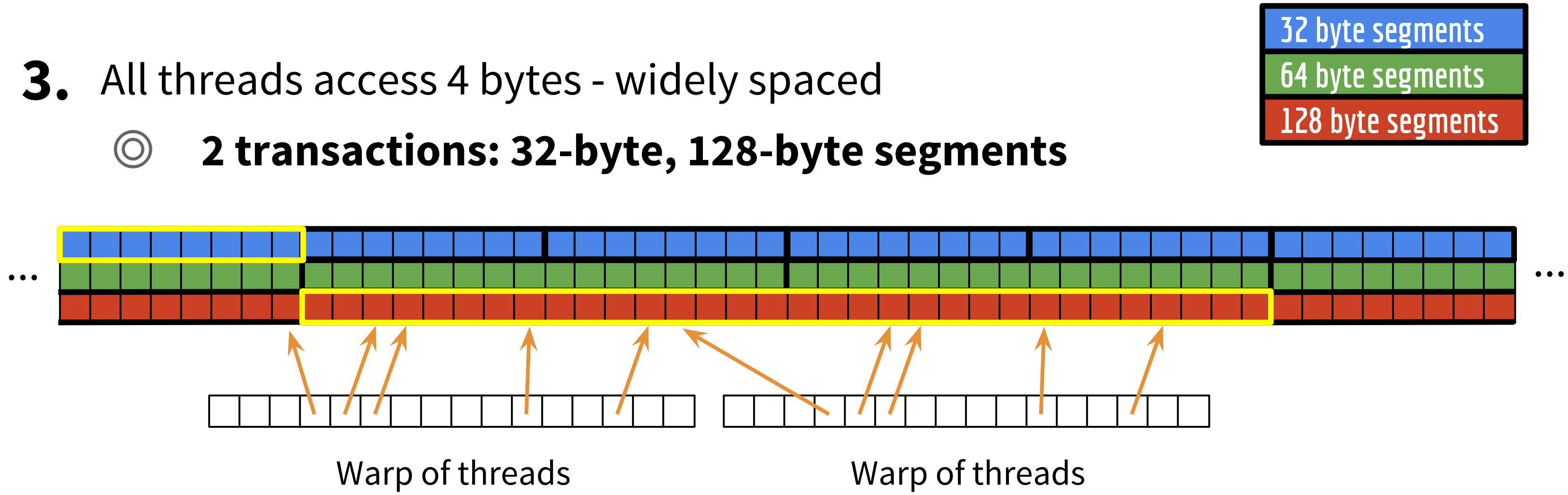
Global memory coalescing

◎ h/w recognizes all threads are within single 64-byte addressable segment; only 1 transaction is performed

Accessing Global Memory

3. All threads access 4 bytes - widely spaced

- **2 transactions: 32-byte, 128-byte segments**



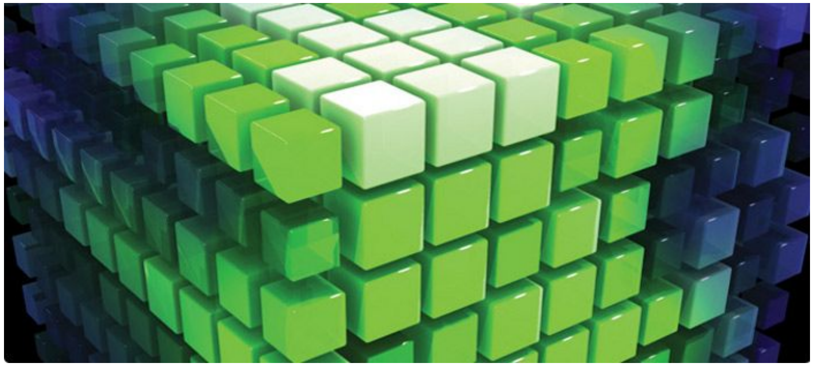
Final Comments

“Premature optimization is the root of all evil.”

-Tony Hoare

1. Get it working
2. Make it fast
   a. Try to arrange your data so that accesses by different threads are close together
   b. Try to partition the work you need to do so that you can give independent tasks to each SM

3. Programming in CUDA

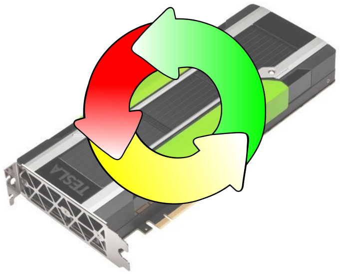


Words you should know:

“Host”

“Device”

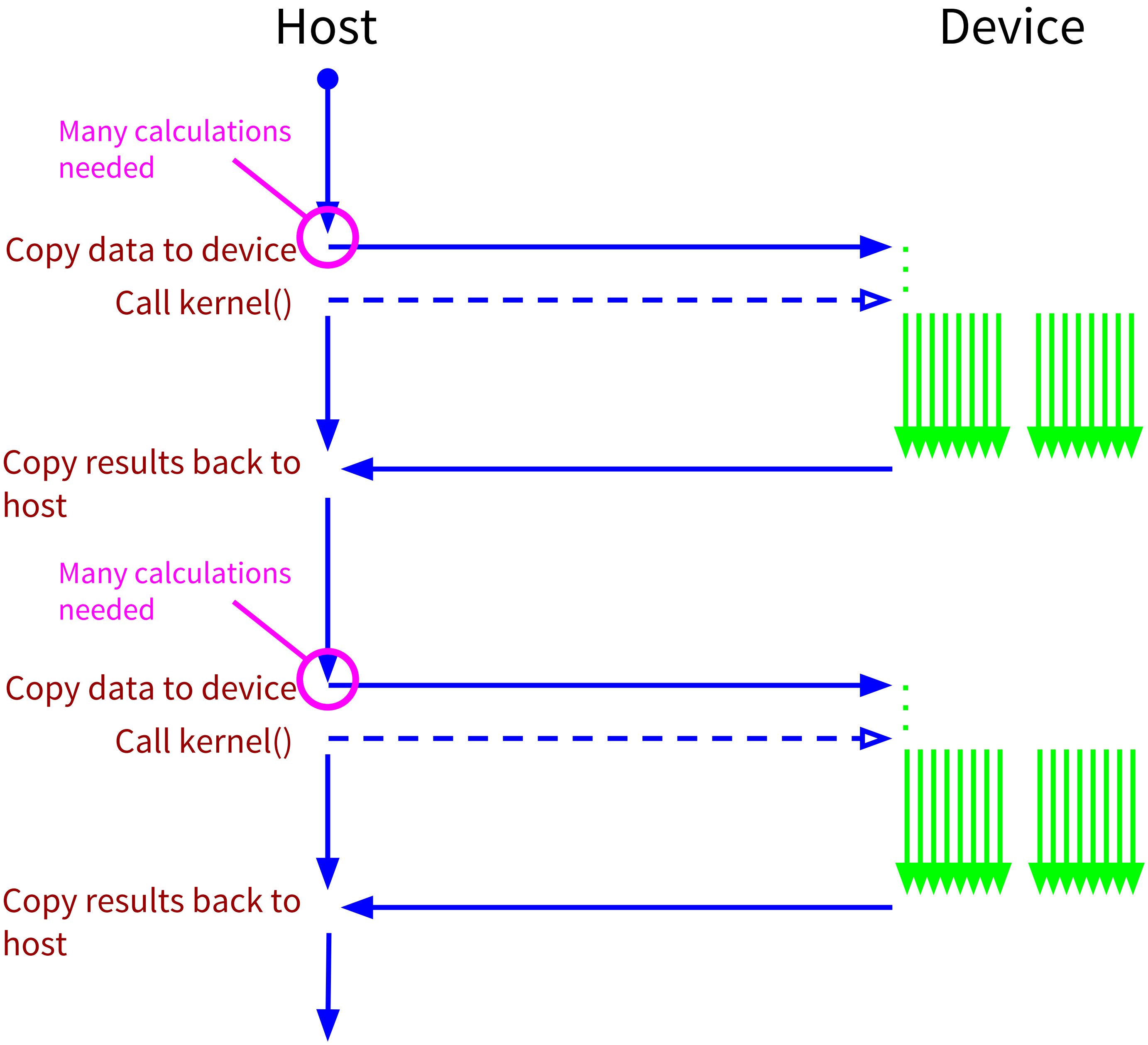
“Kernel”



Introduction to CUDA

- CUDA Language
  - C/C++-like syntax with some minor extensions
  - We'll use the C-subset
- GPUs are *accelerators*
  - Can’t run regular code
  - Invoked only for compute-intensive tasks
- How it works:
  - We write a *Host program*
  - Call CUDA API functions to control the GPU

Anatomy of a CUDA Program



Example - Vector Addition

A:		2	8	5	9	7	1
B:	+	7	2	6	1	4	8
C:		9	10	11	10	11	9

1. Starting Out

Starting Out

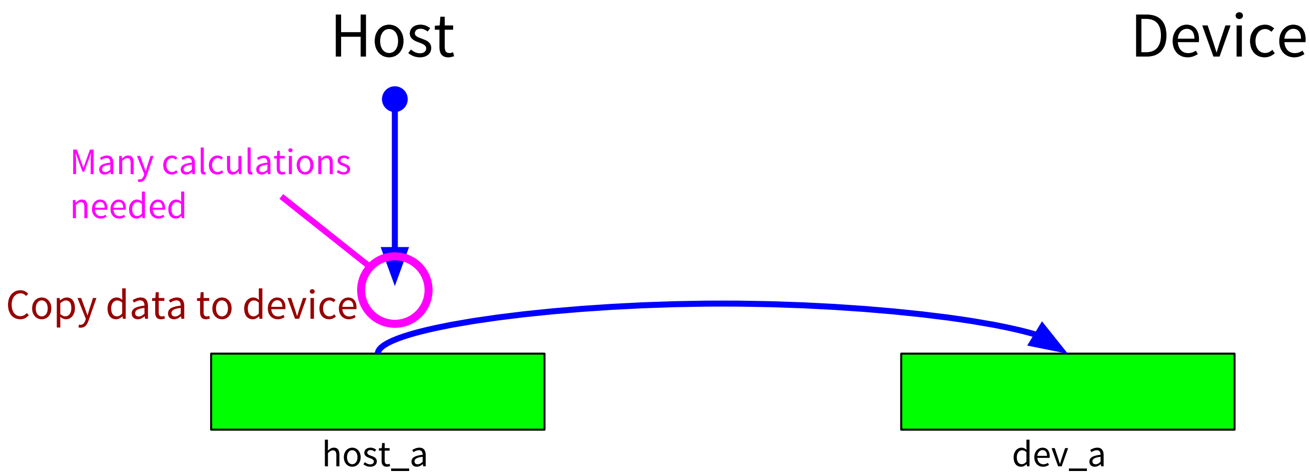
```c
int main(int argc, char *argv[]) {
    // grab n from command line
    const int n = parse_args(argc, argv);

    // allocate host buffers
    float *host_a = (float *) malloc(n * sizeof(float));
    float *host_b = (float *) malloc(n * sizeof(float));
    float *host_c = (float *) malloc(n * sizeof(float));

    // fill A and B with random floats
    init_vec(host_a);
    init_vec(host_b);
    ...

    return EXIT_SUCCESS;
}
```

2. Buffers & Transferring Data



Allocating Device Buffers

```
// cpu
float *host_a = (float *) malloc(n * sizeof(float));
```

```
// gpu
float *dev_a;
cudaError_t status;
status = cudaMalloc(&dev_a, n * sizeof(float));
```

- dev_a now points to a buffer in GPU's global memory
  - Do the same to create dev_b, dev_c

Transferring data

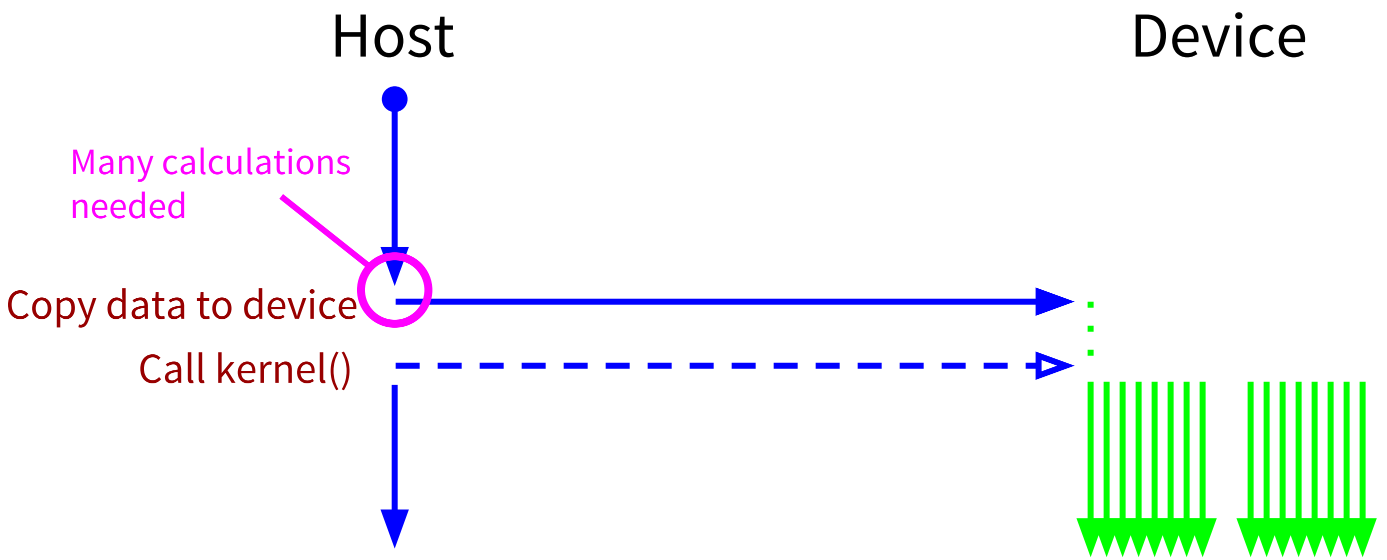
```
// Host -> Device
status = cudaMemcpy(
                 dev_a,                      // destination
                 host_a,                     // source
                 n * sizeof(float),          // size (bytes)
                 cudaMemcpyHostToDevice // direction
         );
```

- Same for dev_b

Transferring data

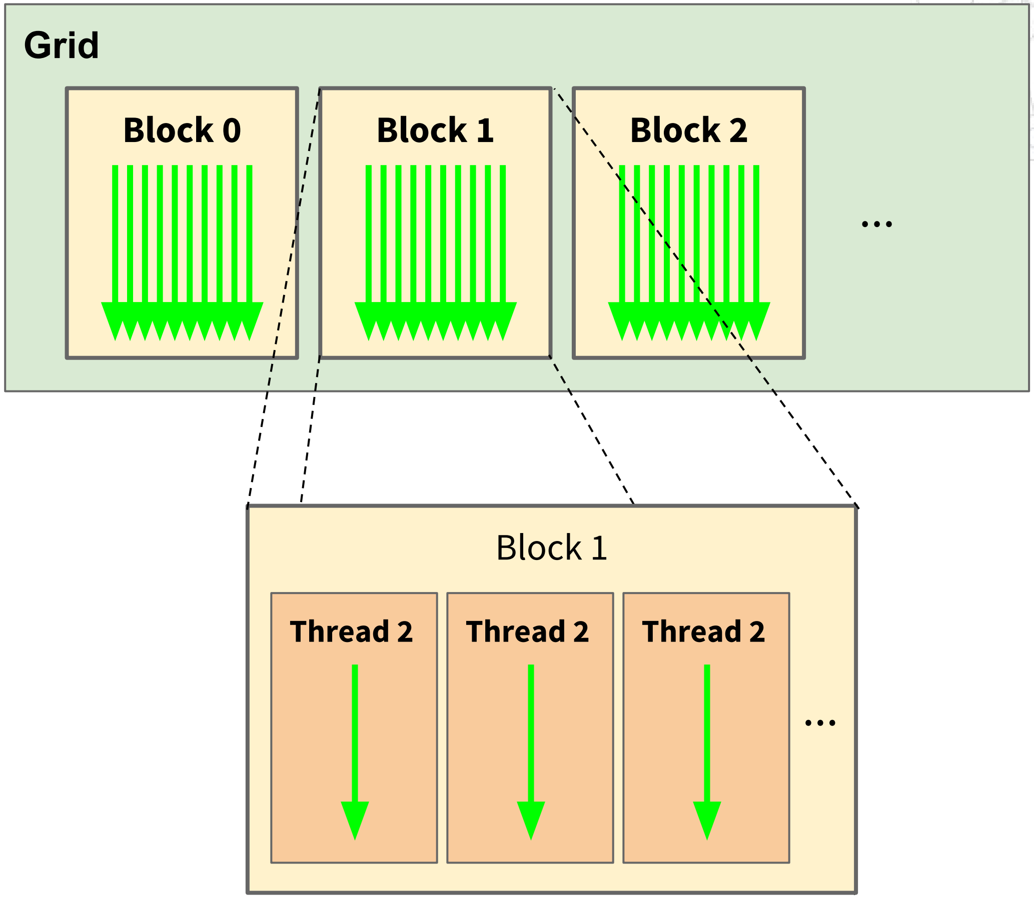
- `cudaMemcpy()` is *blocking*
  - Like `MPI_Send()`
  - Host waits…

3. Writing & Calling Kernels



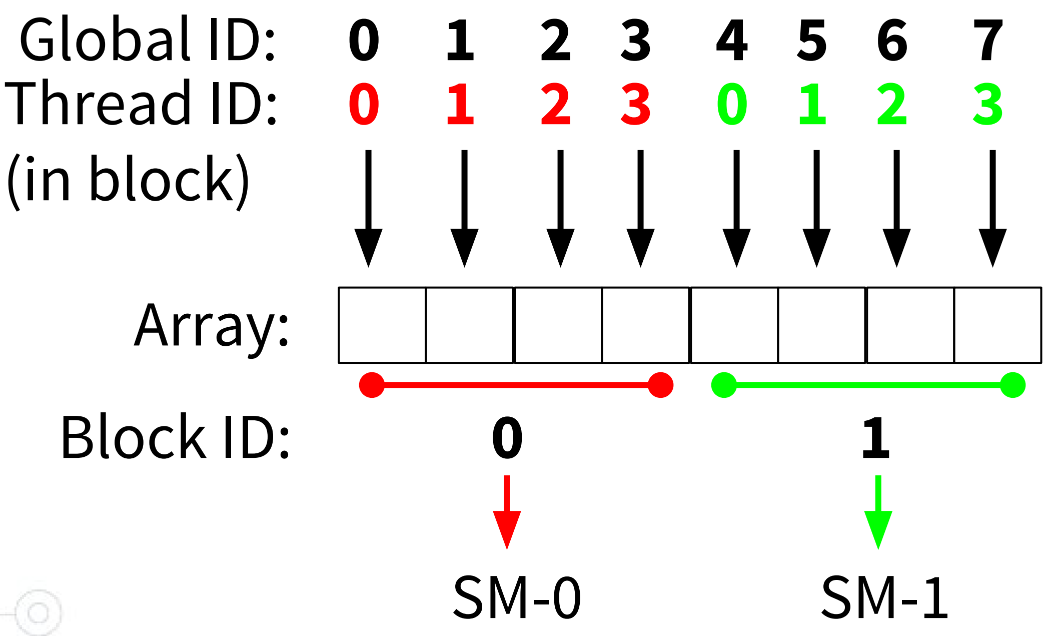
Threads in CUDA

- Thread grid
  - Contains all threads executing on GPU
  - Sub-divided into **thread blocks**



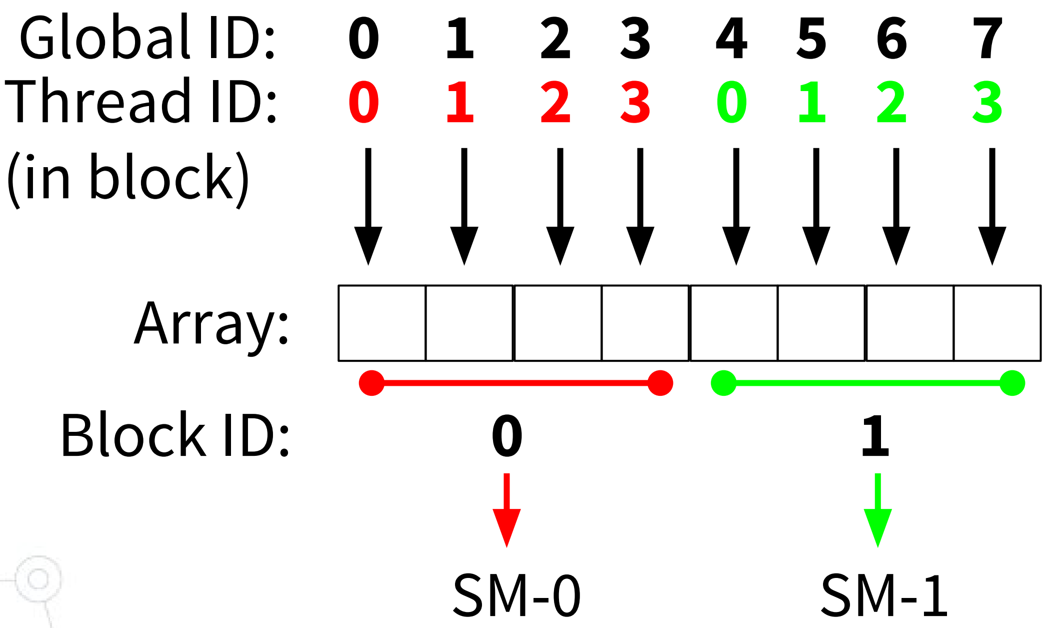
Threads in CUDA

- Why is the grid split into blocks?
  - GPU is made up of multiple SMs.
  - Each block runs on a separate SM.

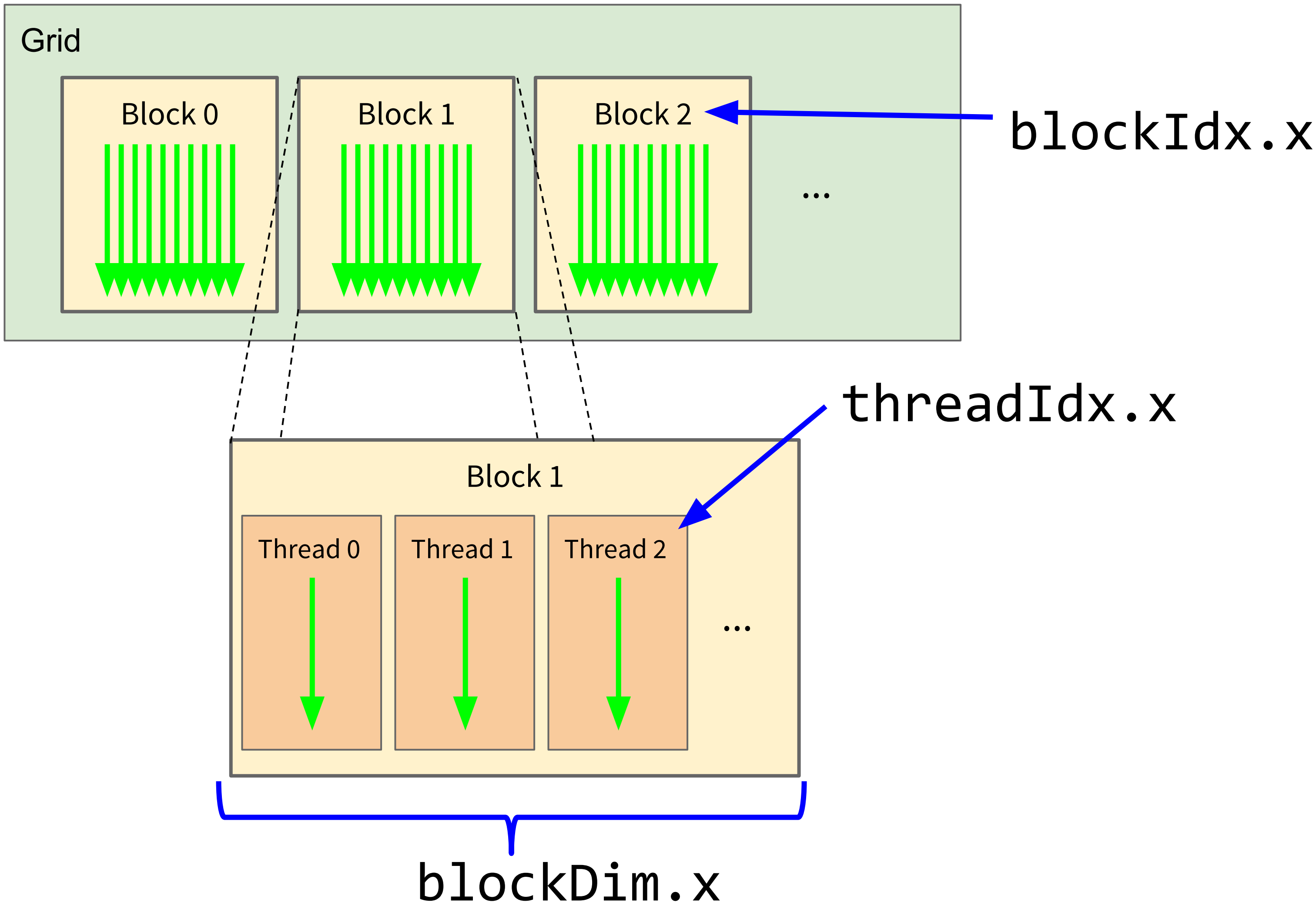


Threads in CUDA

- CUDA gives us **thread id** and **block id**
- Must use them to *calculate* **global id**



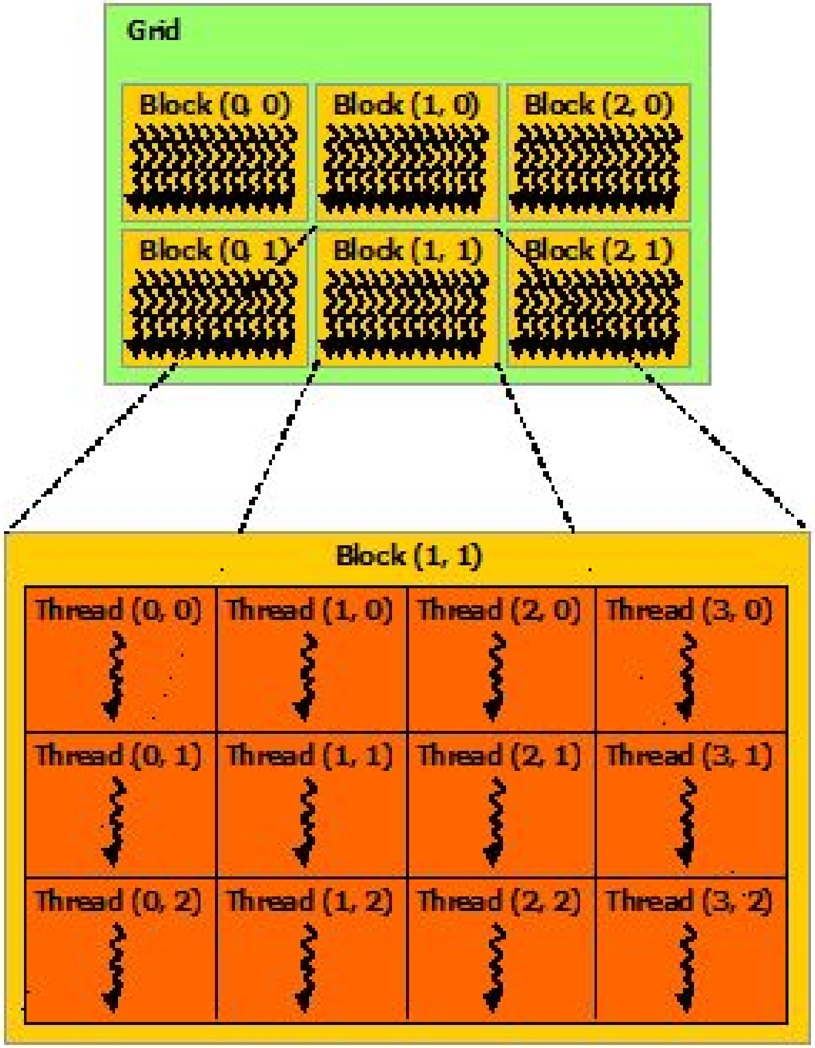
In kernel functions, we have access to:




```
global_id = blockIdx.x * blockDim.x + threadIdx.x
```

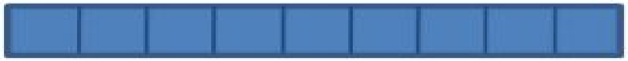
Grid / Block Dimensionality

- Thread grid can also be 2D
  - or 3D...



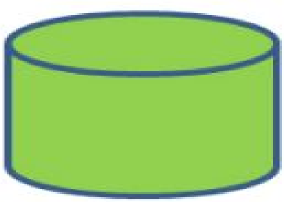
Grid / Block Dimensionality

- Why?

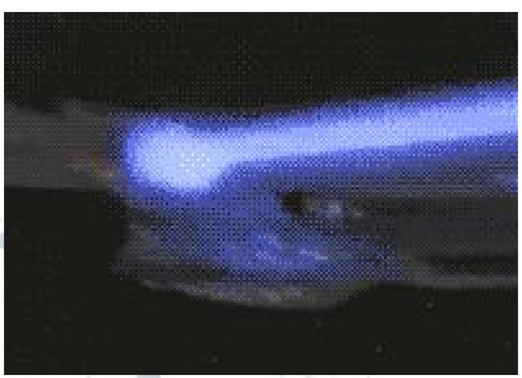


D = 1
Global ids: **x**

D = 2
Global ids: **(x, y)**



D = 3
Global ids: **(x, y, z)**



D = 4

Picking a grid and block size

Let's use a 1D grid and blocks.
To write our kernel, we need to know:

1. *How many threads do we need in our grid?*

   ```
   int threads = n;
   ```

2. *How many threads in a block?*
   - use max threads per block (max parallelism)
   - 512 on our GPU

3. *How many blocks?*

```
int blocks = threads / 512 + (threads % 512 > 0 ? 1 : 0);
```

- Note: This means we may have more threads than we need...

Launching the Kernel

- “***Launching***”: calling a kernel function from the host
- Like a C function call…
  - plus some syntax to tell CUDA how many threads & blocks to use!

```
vec_add<<<blocks, threads>>>(dev_a, dev_b, dev_c, n);
```

function name

Blocks in grid

Threads in block

Pointers to our device buffers

Vector length

Writing a Kernel Function

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

Writing a Kernel Function

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

- Marks this as a kernel function

Writing a Kernel Function

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

- Kernel functions can’t return anything
  - All communication between host & device done through data transfers

Writing a Kernel Function

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

- Function name

Writing a Kernel Function

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

- Args are passed using “call by copy”
  - Pointers are shallow-copied
  - Args on stack are copied
  - Placed in constant memory (limit 4KB)

Writing a Kernel Function

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

- Calculate global ID

Writing a Kernel Function

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

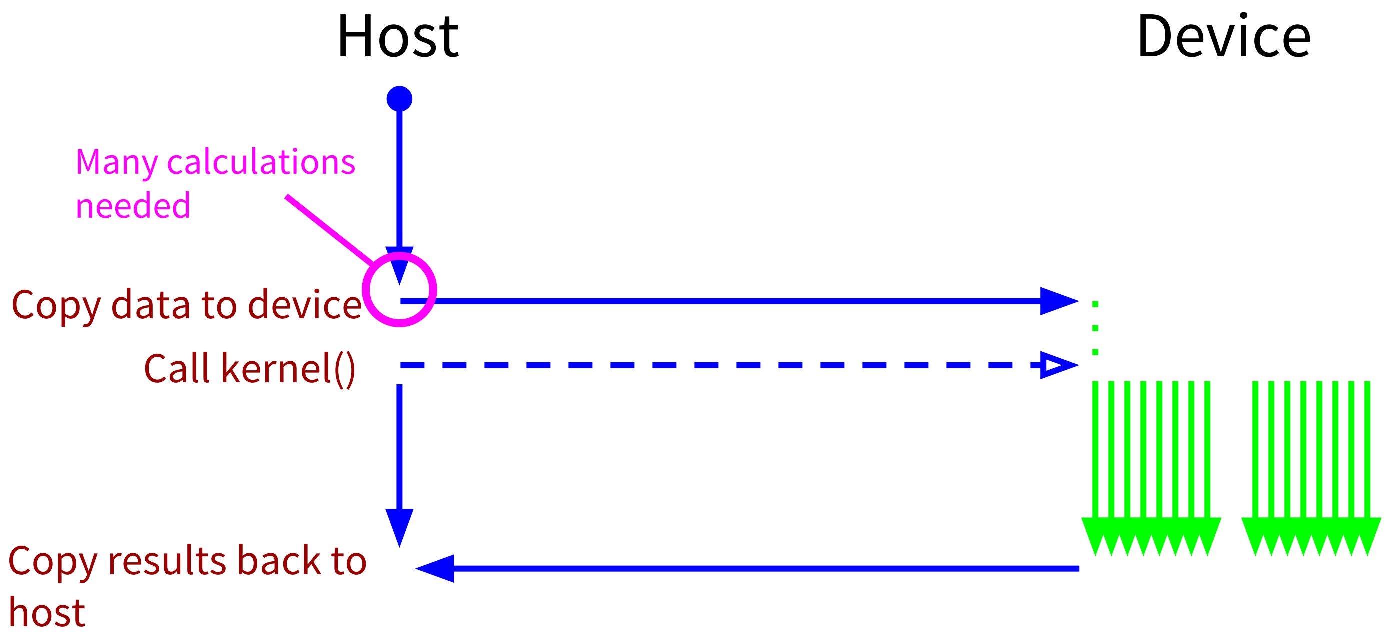
- Recall: we may have more threads than we need
  - last block...

Writing a Kernel Function

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

- global_id used to index vector
  - Each thread adds one column of vectors
- Result written to c (in dev memory)

4. Retrieving the Result



Synchronization

- Kernel calls are non-blocking!
  - Host program continues on to next instruction
  - Can sync up at end using:
    1. `cudaDeviceSynchronize()`, OR
    2. Issuing a (blocking) `cudaMemcpy()`

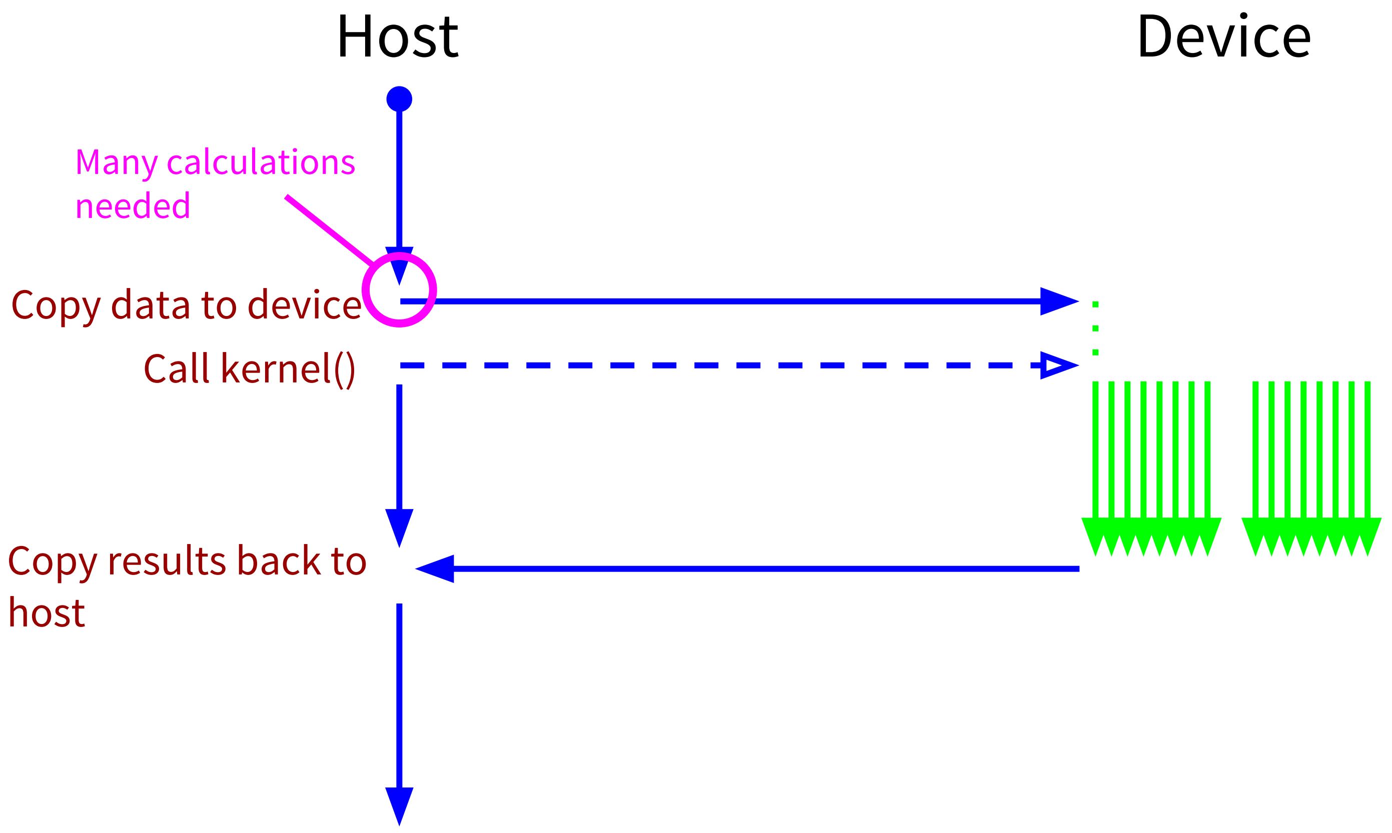
Preferred method if you need results back
(avoids redundant sync)

```
vec_add<<<blocks, threads>>>(dev_a, dev_b, dev_c, n);
// host continues immediately...
```

Retrieving the Result

```
// Device -> Host
status = cudaMemcpy(
                 host_c,                       // destination
                 dev_c,                        // source
                 n * sizeof(float),            // size (bytes)
                 cudaMemcpyDeviceToHost // direction
         );
```

5. Back on Host...

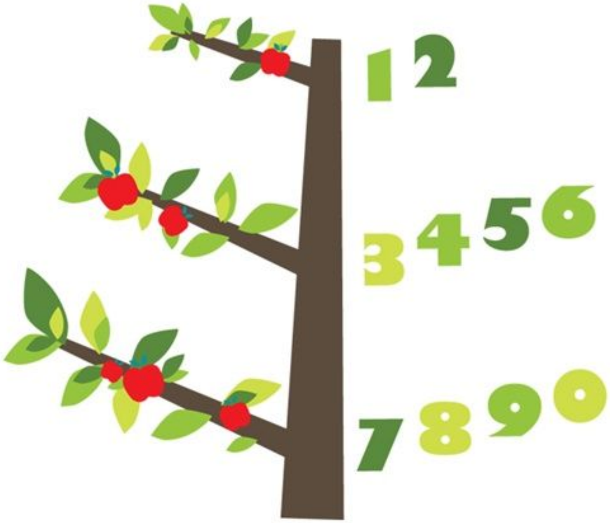


Example Code

- Full vector sum code up on course website
- GPUs available on cuckoo machines
  - See "**Programming Environments**" doc for which machine to log into!
    - No qsub...

- DEMO -

4. Case Study: Sum Reduction



Sum Reduction

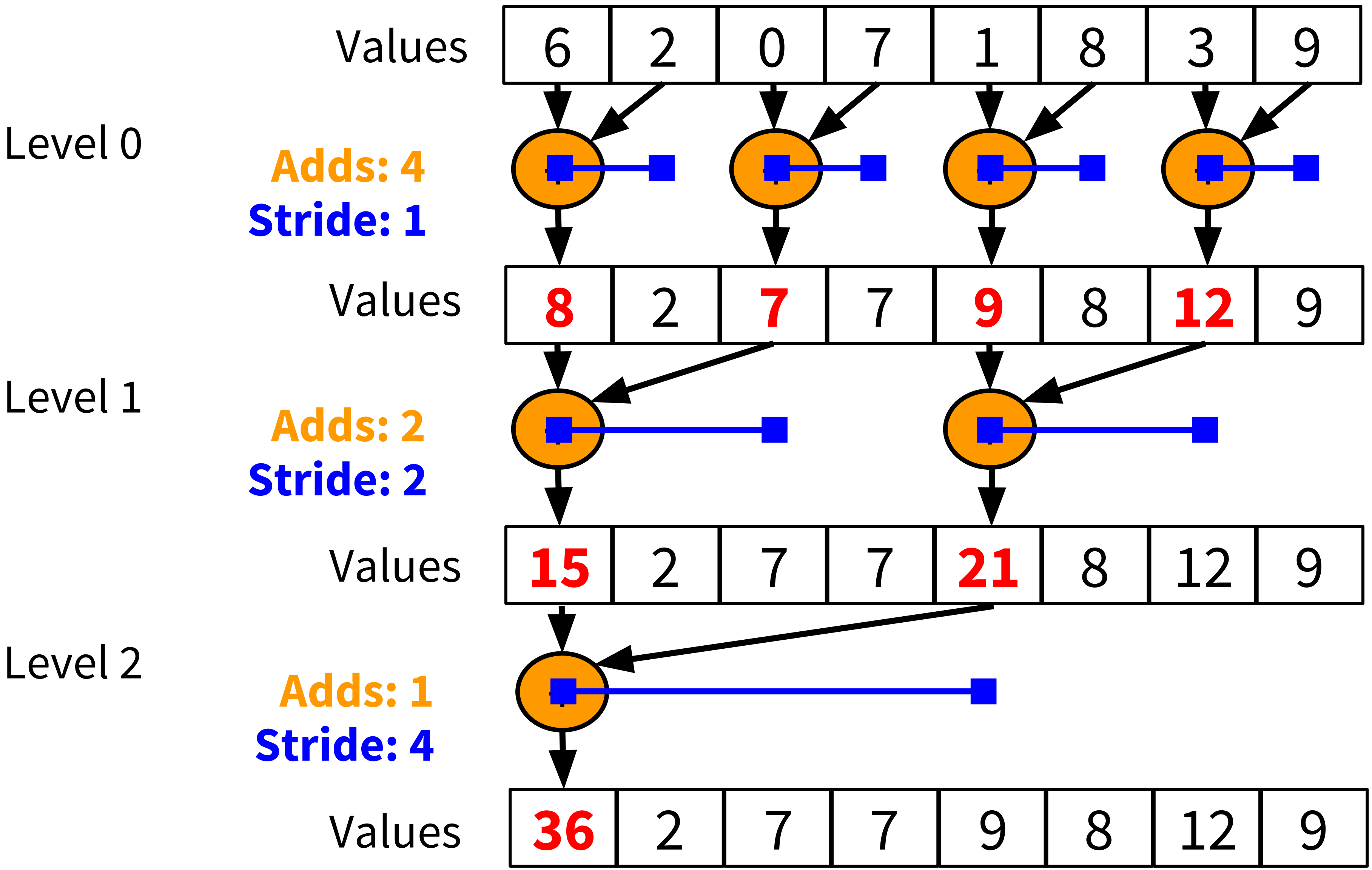
- Adding up the elements of an array
  - `MPI_Reduce()`
  - `#pragma omp parallel for reduction(+:sum)`

CPU Reduction

- Simple OpenMP implementation (2-cores)
  - $n = 2^{25}$

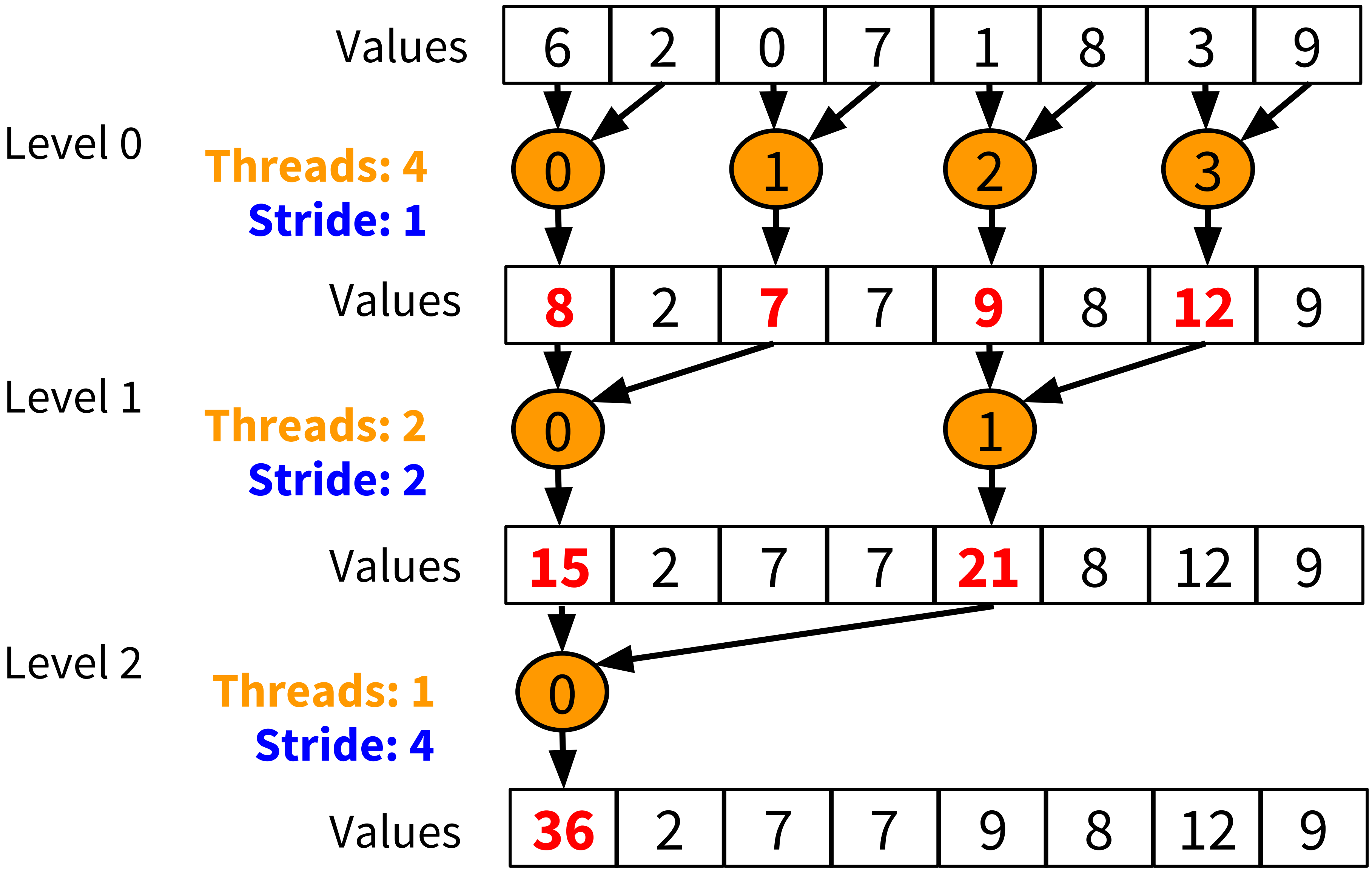
Approach	Throughput (MFLOPS)
CPU	**558**

An Array-based Approach



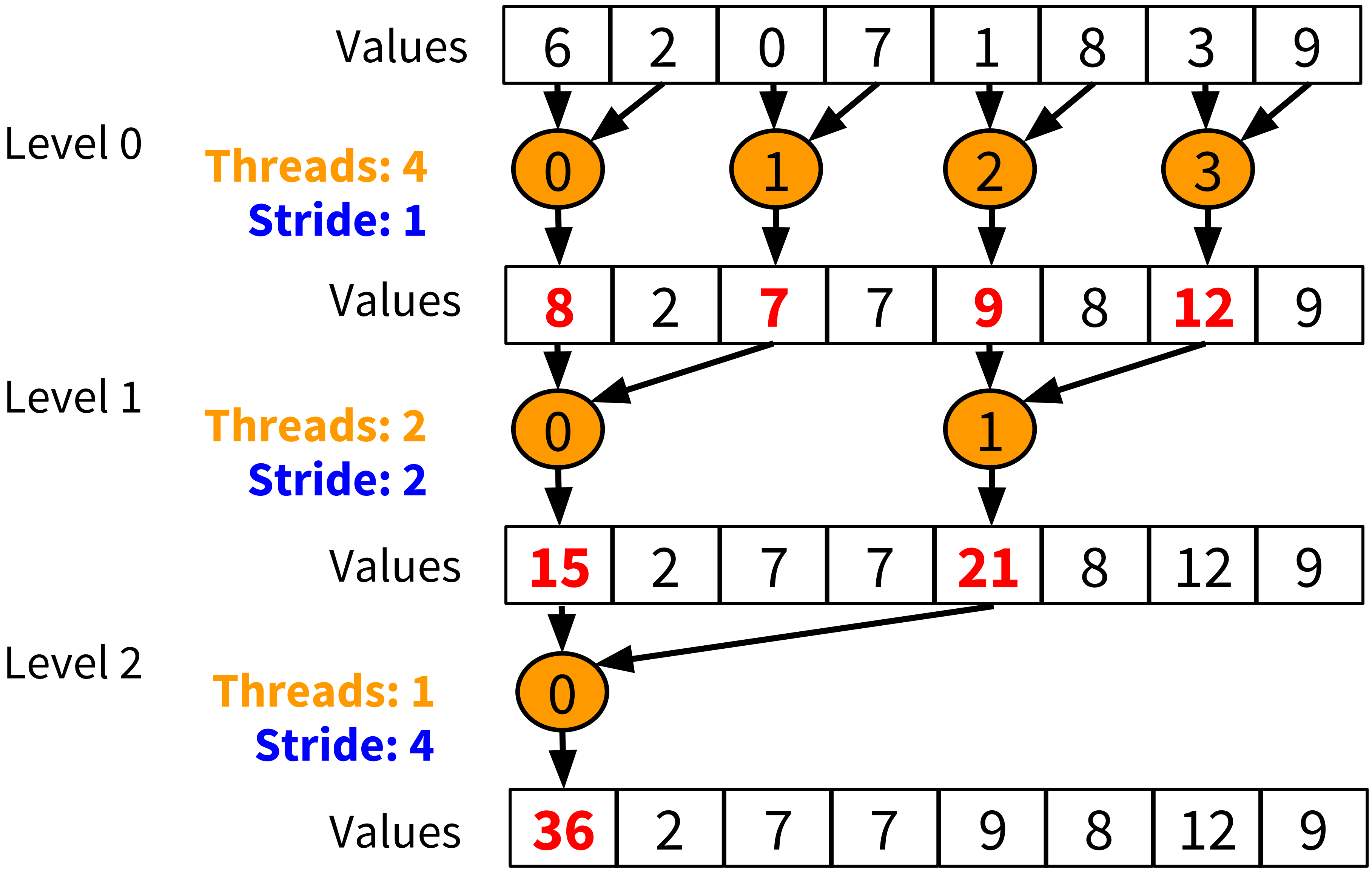
On each step: **Adds /= 2**, **Stride *= 2**

An Array-based Approach



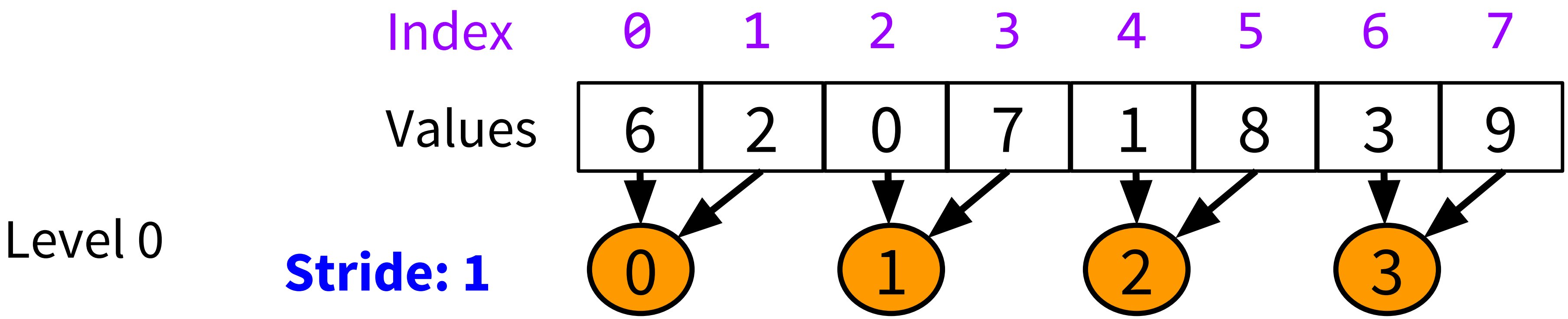
Each threads does one add. Must sync after each level.

An Array-based Approach

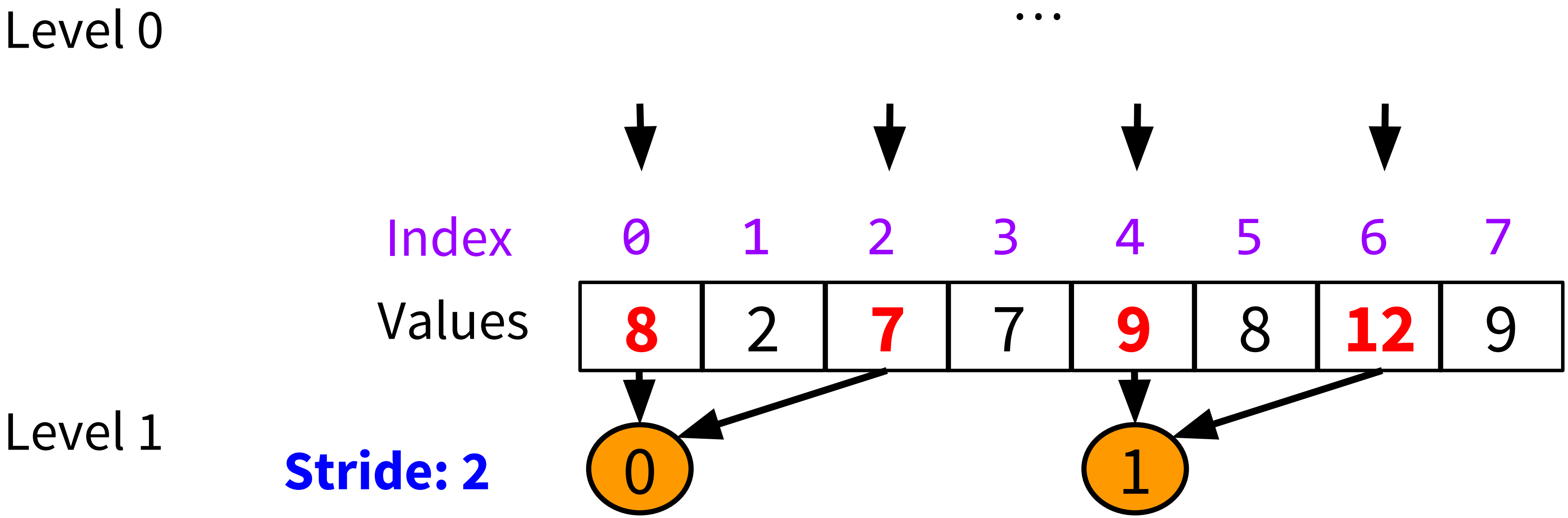


When do we stop? Threads == 0, or (equivalently) **Stride == n**

An Array-based Approach



- What's the index of each left number?
  - `0, 2, 4, 6`
- Say we're thread 1. How can we calculate our left index from our id?
  - `Multiply by 2`
- Generalizing, if we're thread id:
  - `left = id * 2`

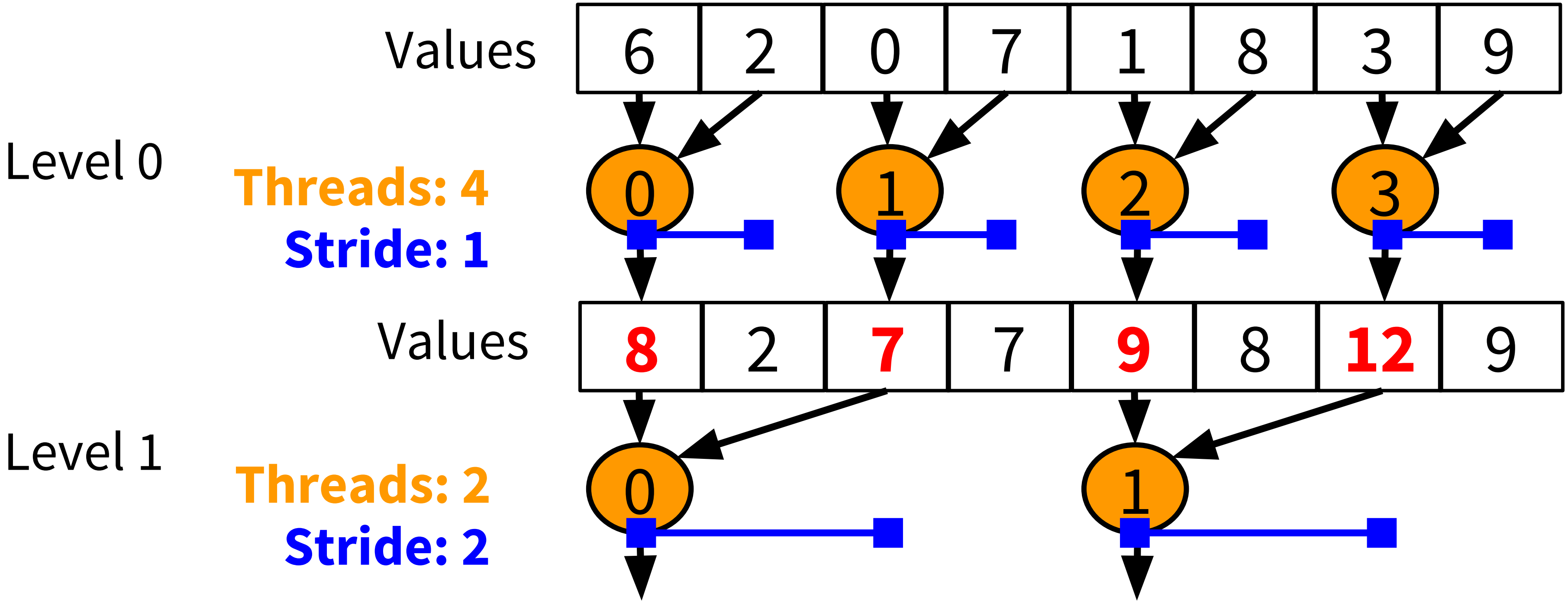


- What are the left indices at level 1?
  - `0, 4`
- Our pattern is `left = id * 2`. Does it work here?
  - No.
  - What should the pattern be here?
  - `left = id * 4`

A General Formula

- We have:
  - level 0: `left = id * 2`
  - level 1: `left = id * 4`
- What changes between levels?
  - Stride:
    - At level 0, stride = 1
    - At level 1, stride = 2

```
left = id * (stride * 2)
```



- So we have:

```
left = id * (stride * 2)
```

- What about a formula for the right index?
  - If we know `left`, then:

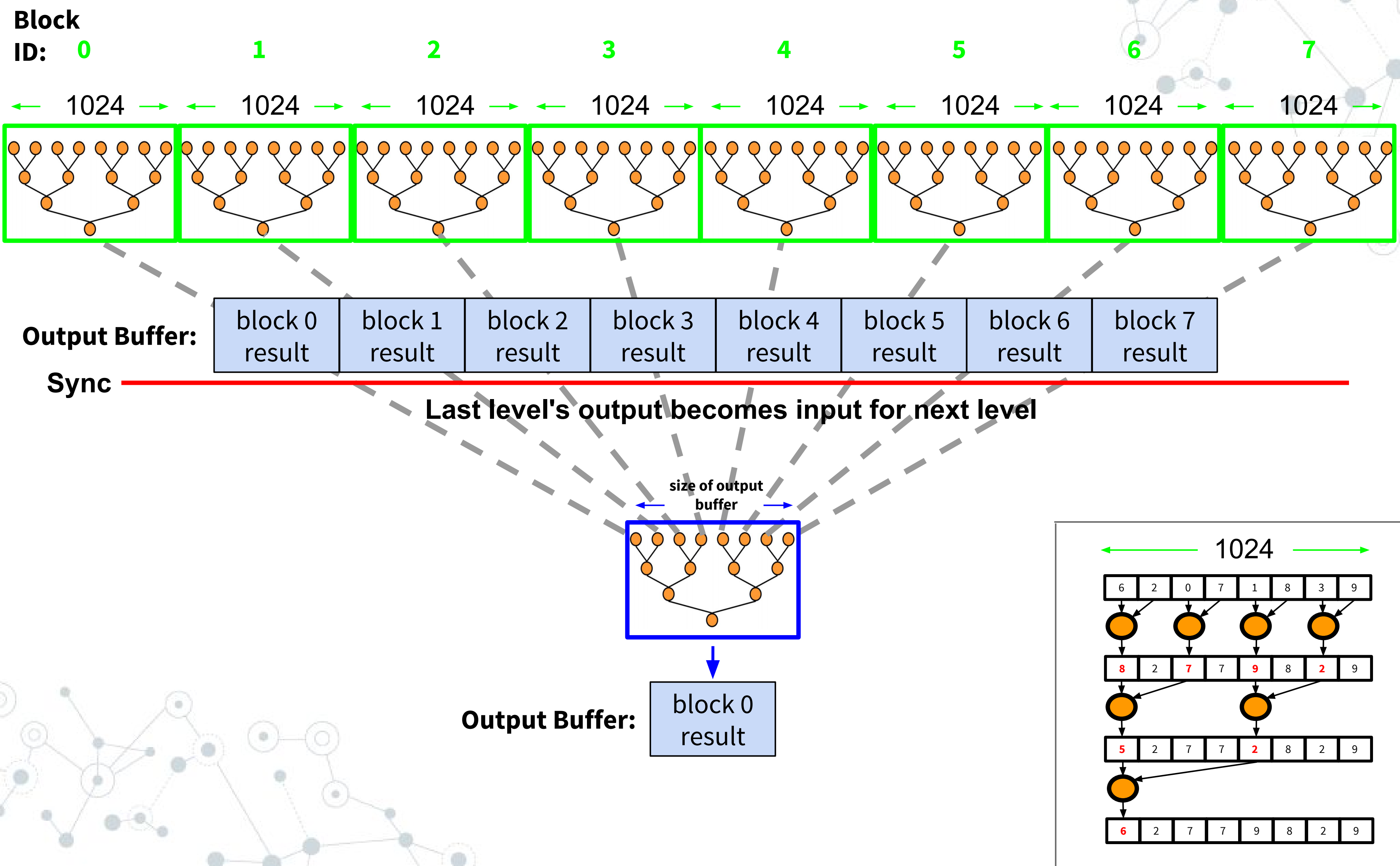
```
right = left + stride
```

Writing a Kernel

```
__global__ void reduce(float *array, int n) {
  int global_id = blockIdx.x * blockDim.x + threadIdx.x;
  int threads;
  int stride;
  int left, right;
  threads = n / 2;
  for (stride = 1; stride < n; stride *= 2, threads /= 2) {
     if (global_id < threads) {
        left = global_id * (stride * 2);
        right = left + stride;
        array[left] = array[left] + array[right];
     }
     __syncthreads();
  }
}
```

How many threads?

- If we have n elements
  - Need n / 2 threads
- How many blocks?
  - Our kernel assumes we can run all of the threads we need
  - Problem: Max block size is 512
  - Current code will only work for `n <= 2 * 512 = 1024`
- **Solution:** break array into chunks of size 1024
  - Use multiple blocks!



Block ID: 0 1 2 3 4 5 6 7
1024 1024 1024 1024 1024 1024 1024 1024
Output Buffer:
block 0 result
block 1 result
block 2 result
block 3 result
block 4 result
block 5 result
block 6 result
block 7 result
Sync
Last level's output becomes input for next level
size of output buffer
Output Buffer:
block 0 result
1024
6 2 0 7 1 8 3 9
8 2 7 7 9 8 2 9
5 2 7 7 2 8 2 9
6 2 7 7 9 8 2 9

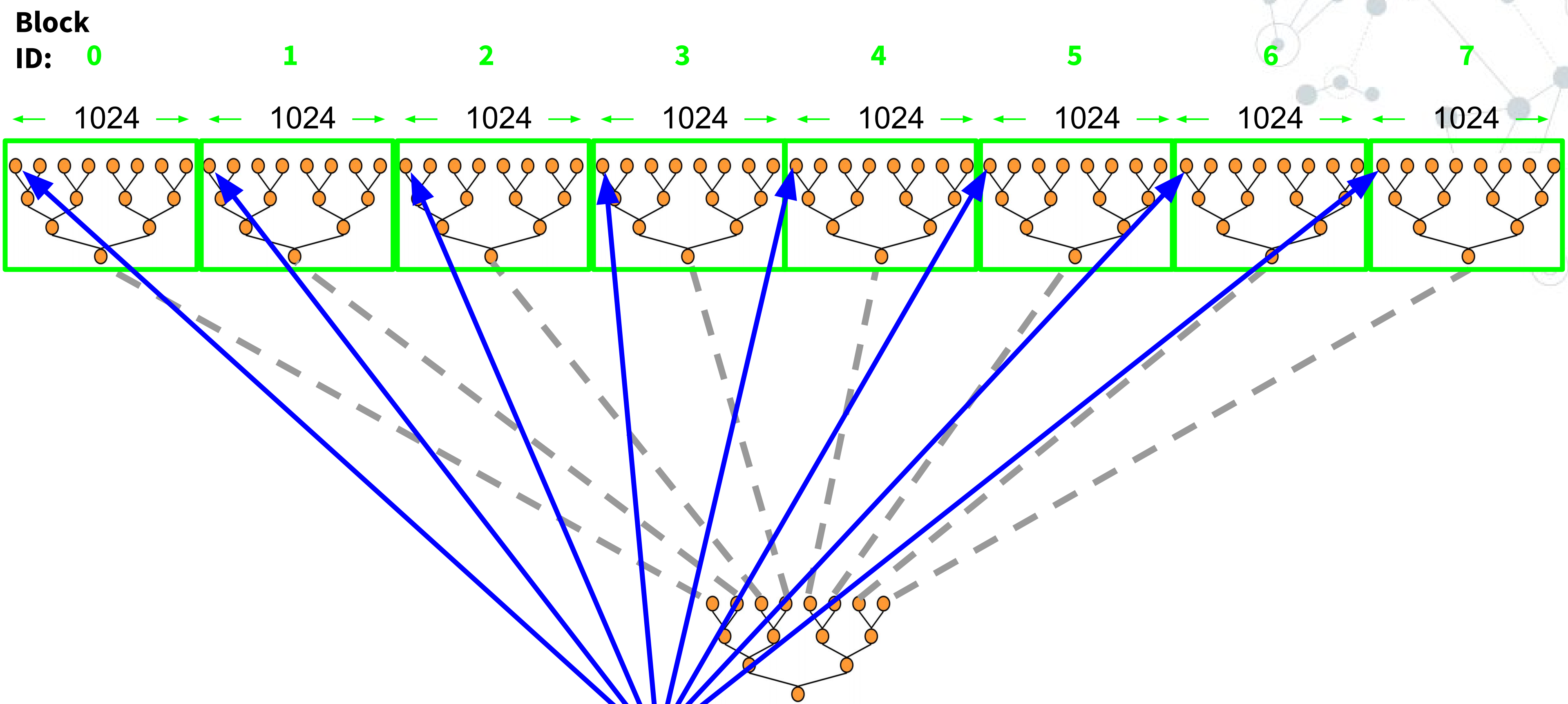
0. Initial Approach

- **Problem:** need to sync after each block-level
  - Can't synchronize thread blocks in CUDA
  - Except by returning control to host…
- **Solution:** launch kernel multiple times
  - Once for each block-level
  - Use a loop on the host
  - Data stays in global memory between launches

Host code

```
int threads = n / 2;
int blocks = threads / 512 + (threads % 512 > 0 ? 1 : 0);
int remaining = n;
while (remaining > 1) {
    // launch kernel
    reduce<<<blocks, 512>>>(input_buf, output_buf, remaining);
    // recalculate num threads & blocks for next iteration
    remaining = blocks;
    threads = remaining / 2;
    blocks = threads / 512 + (threads % 512 > 0 ? 1 : 0);
    // if we'll do another iteration, output becomes input
    if (remaining > 1) {
        float *temp = input_buf;
        input_buf = output_buf;
        output_buf = temp;
    }
}
```

Kernel Changes



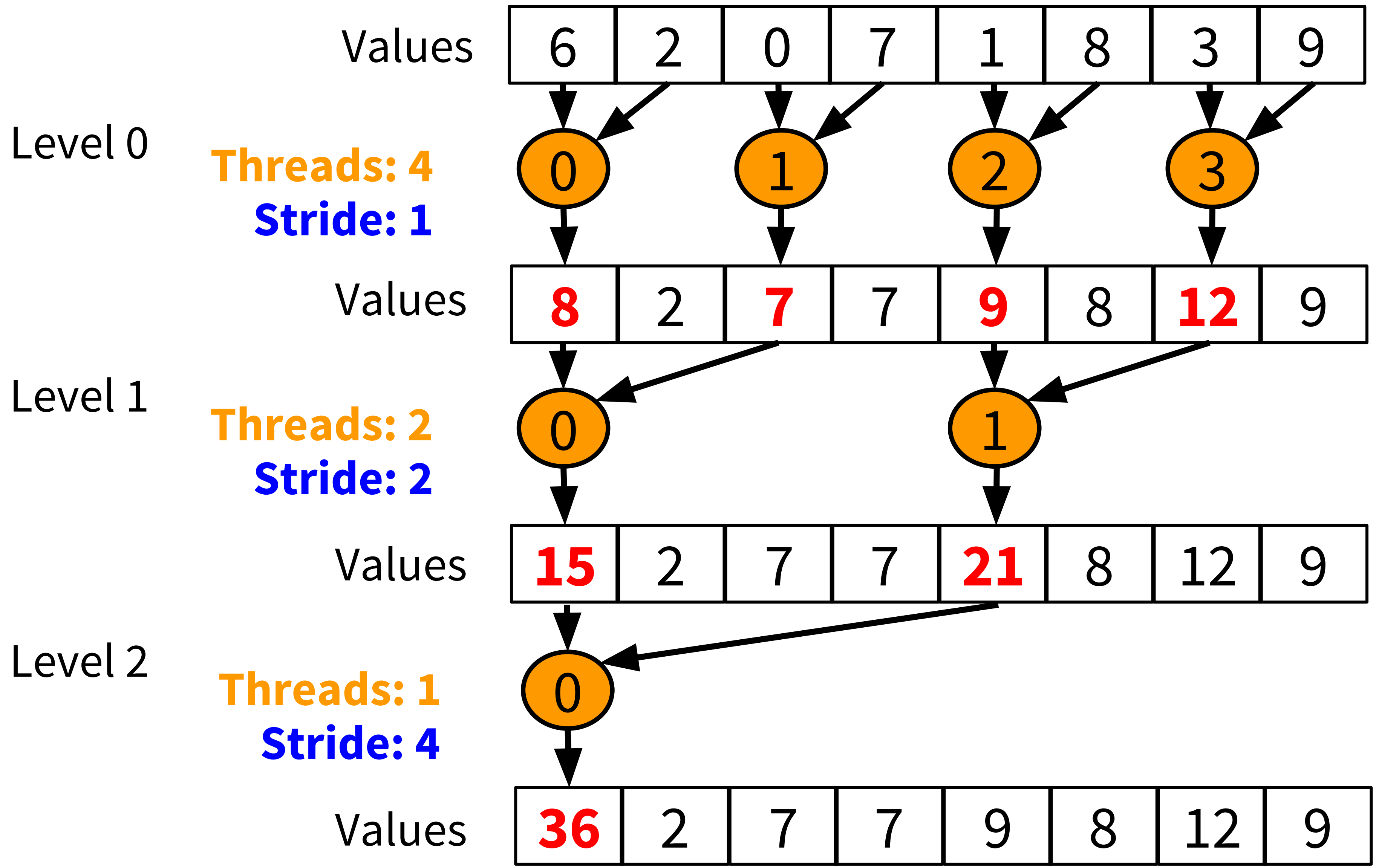

```
int block_start = block_id * 1024;
left = block_start + thread_id * (stride * 2);
right = left + stride;
```

0. Initial Approach - Results

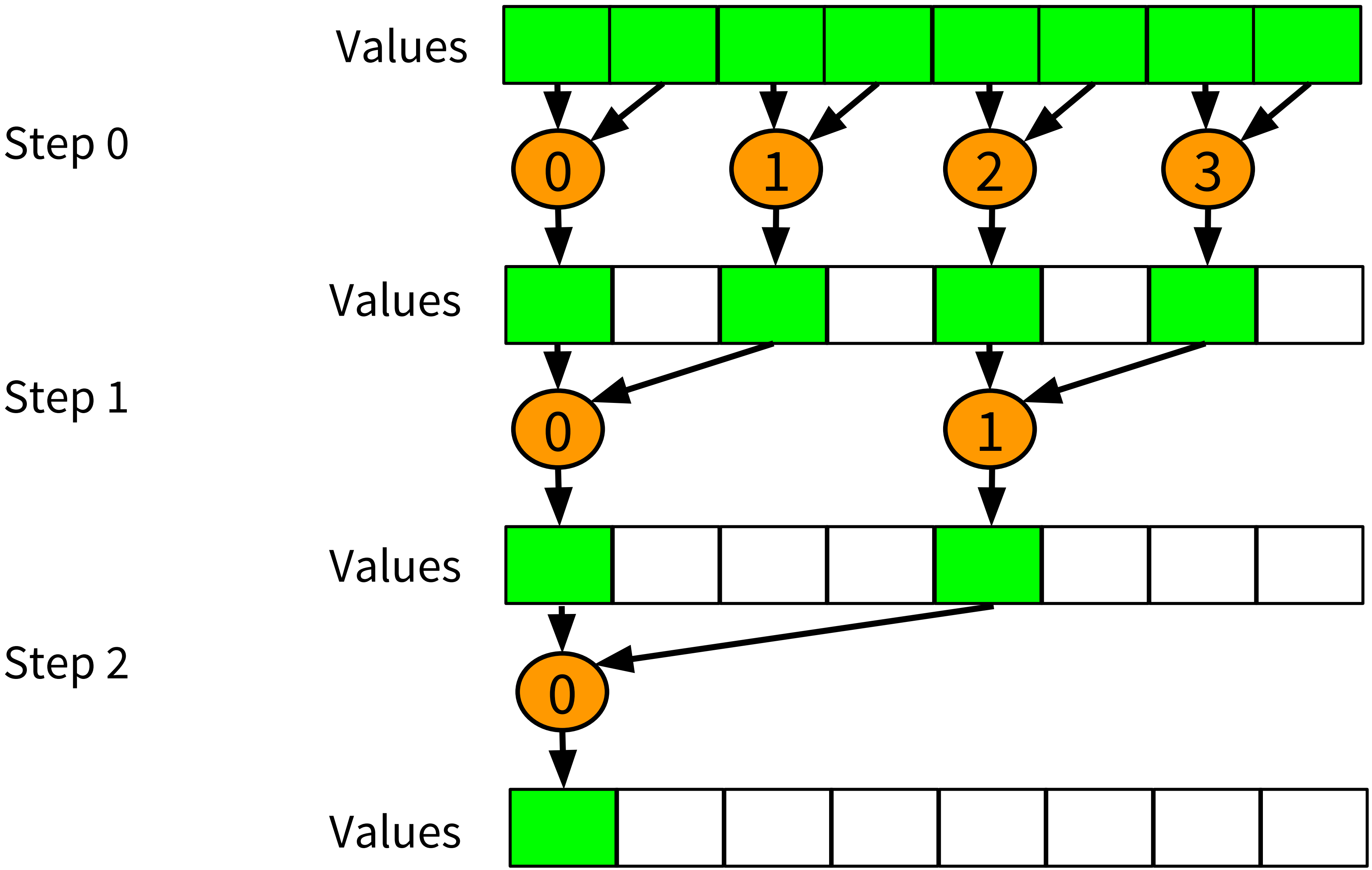
Approach	Throughput (MFLOPS)	Improvement (factor)
CPU	558	
0. Initial Approach	**500**	**-58 (0.9x)**

- Worse than CPU! What is going on?
  - We know memory access patterns matter
  - What do our kernel's look like?

Access Pattern



Reads & Writes: Active Locations



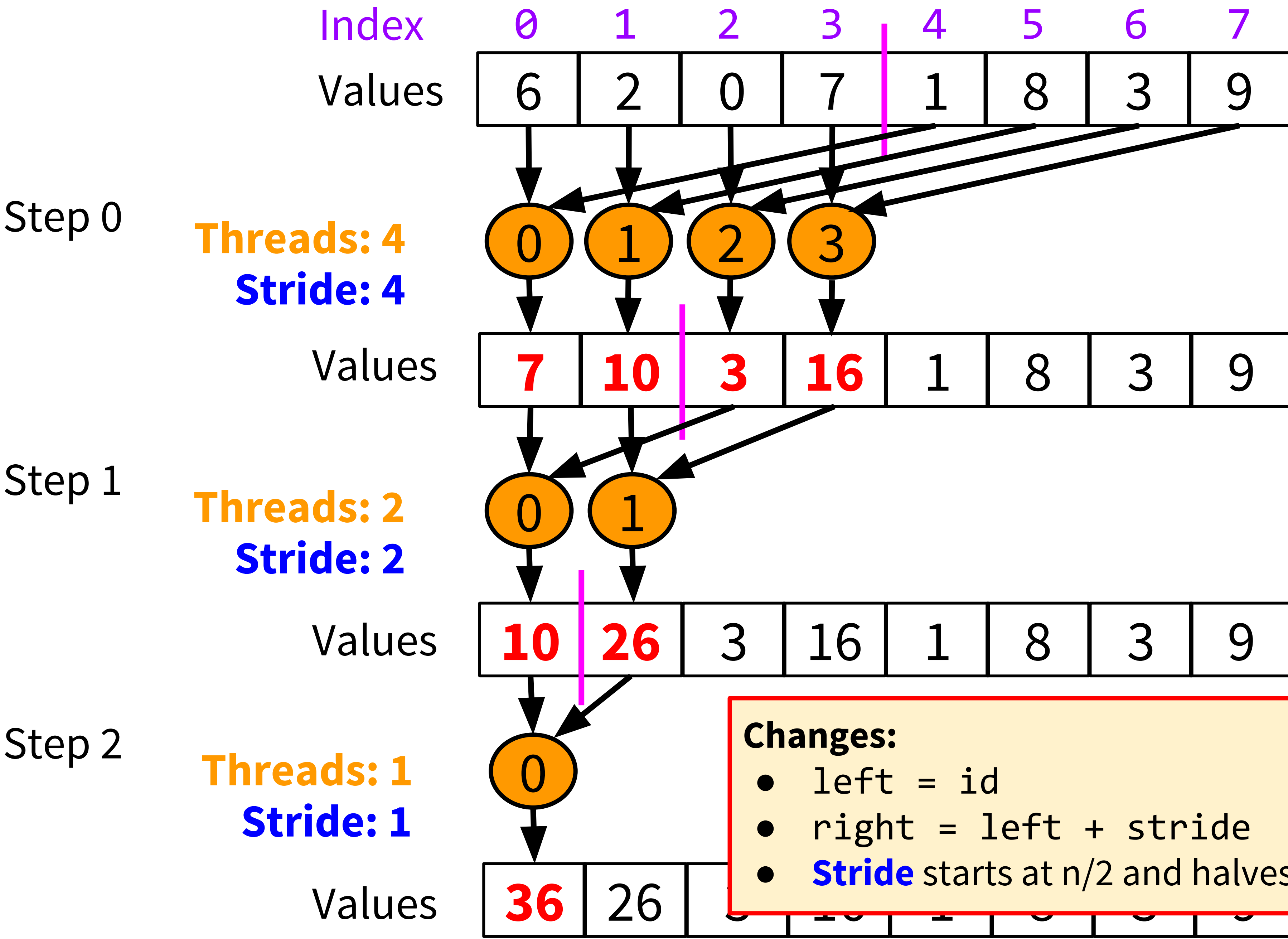
Wide gaps! Leads to poor global memory performance!

How can we fix it?

- Observation: addition is *commutative*
  - Order doesn't matter…
- We can choose which elements we add first
  - Can we eliminate the gaps?

```
x + y = y + x
```

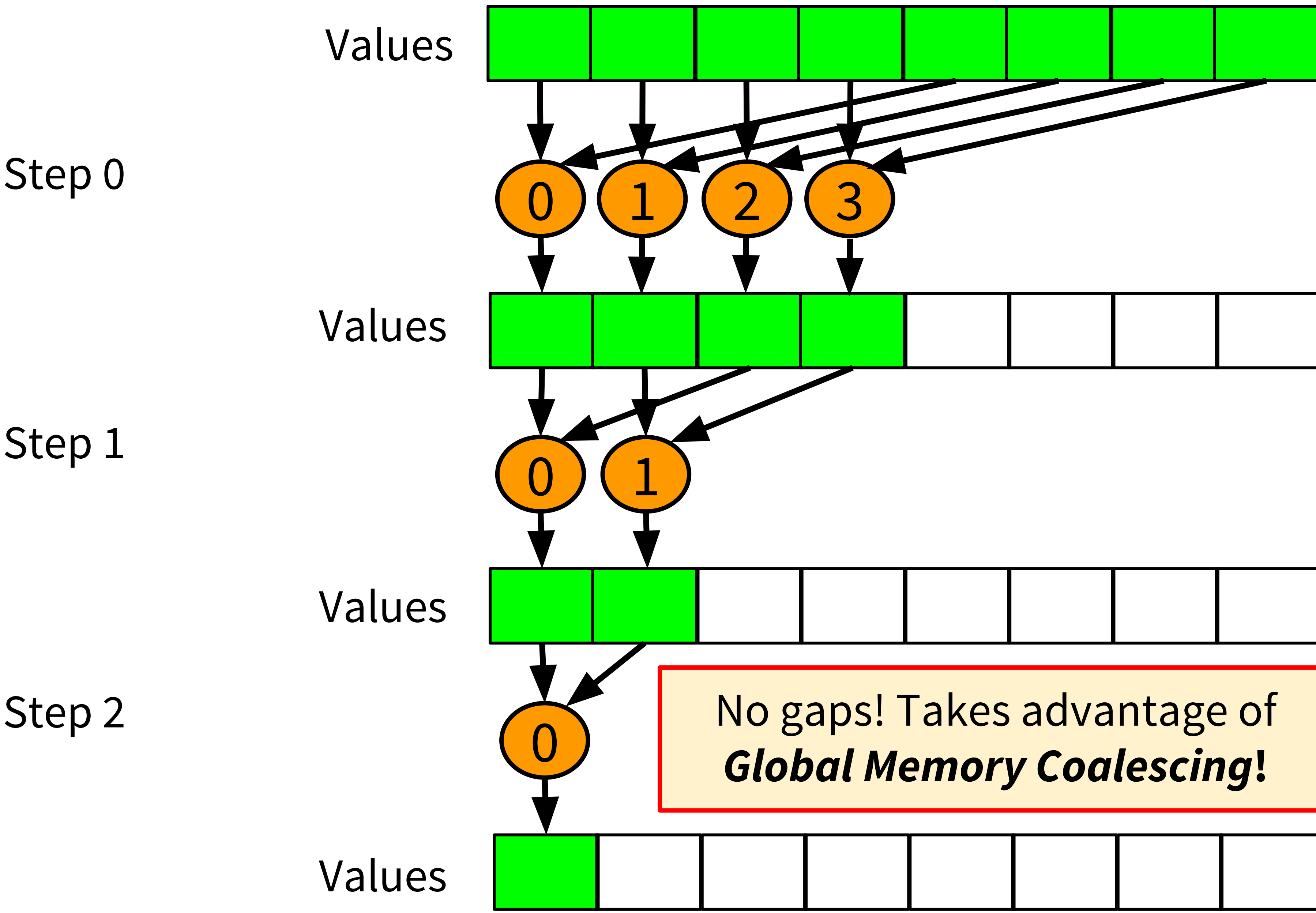
1. Global Memory Coalescing



Changes:

- `left = id`
- `right = left + stride`
- **Stride** starts at n/2 and halves

Reads & Writes: Active Locations



No gaps! Takes advantage of ***Global Memory Coalescing*!**

1. Global Memory Coalescing - Results

Approach	Throughput (MFLOPS)	Improvement (factor)
CPU	558	
0. Initial Approach	500	-58 (0.9x)
1. Global Memory Coalescing	**604**	**+104 (1.2x)**

- Finally better than the CPU!
- Can we do more?
  - **Useful question:** ***How is our execution time being used?***

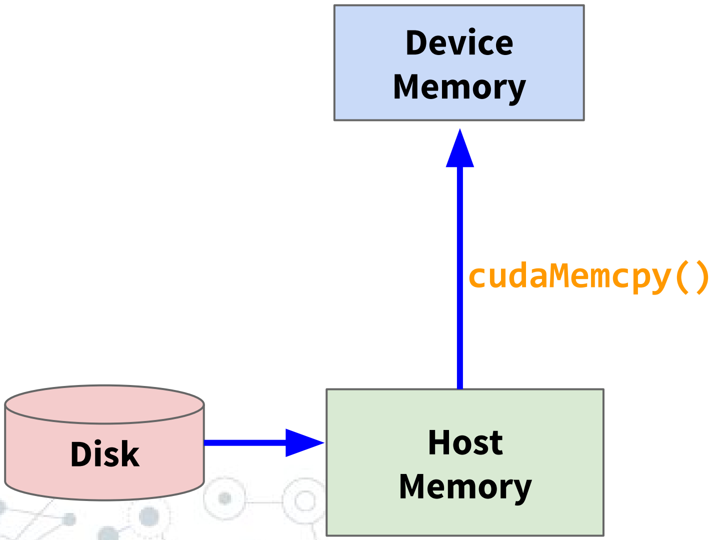
2. Using Pinned Memory

- OS uses virtual memory
  - Memory is segmented into “pages”
  - Can be “swapped out” to disk
    - Disk is slow (up to two orders of magnitude)
- Our array is large
  - May not all be in RAM...

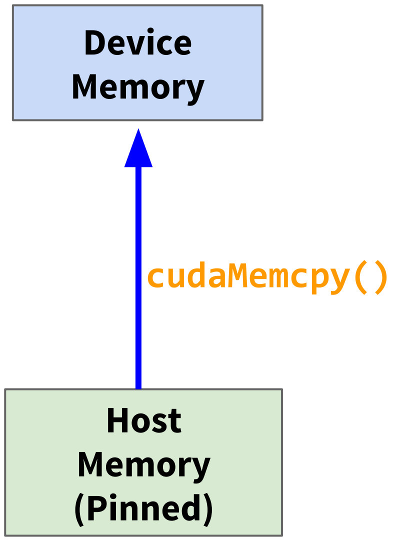
2. Using Pinned Memory

- **Memory pinning**: forcing a buffer to stay resident in host memory.

Regular Data Transfer



Pinned Data Transfer



How?

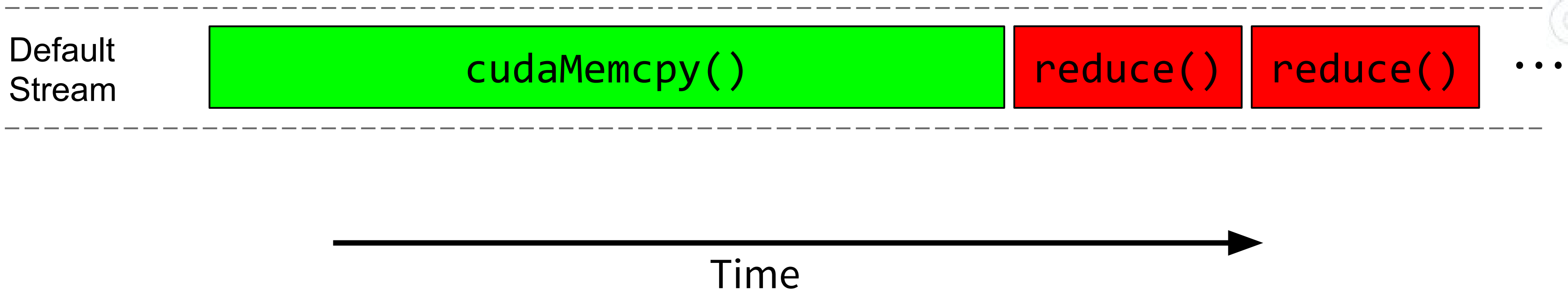
- Instead of `malloc()`ing host buffers, use `cudaMallocHost()`
- Instead of free(), use `cudaFree()`

2. Using Pinned Memory - Results

Approach	Throughput (MFLOPS)	Improvement (factor)
CPU	558	
0. Initial Approach	500	-58 (0.9x)
1. Global Memory Coalescing	604	+104 (1.2x)
2. Using Pinned Memory	**1041**	**+437 (1.7x)**

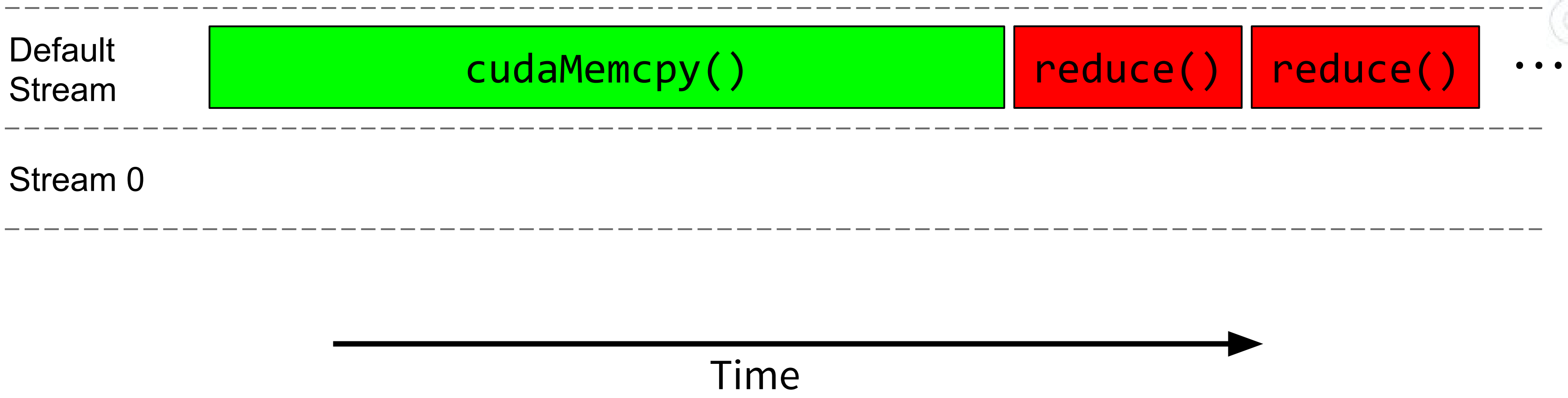
- Wow!
- Transfer time *still* outweighs kernel time though...

3. Using Streams



- Stream: a queue containing pending CUDA calls

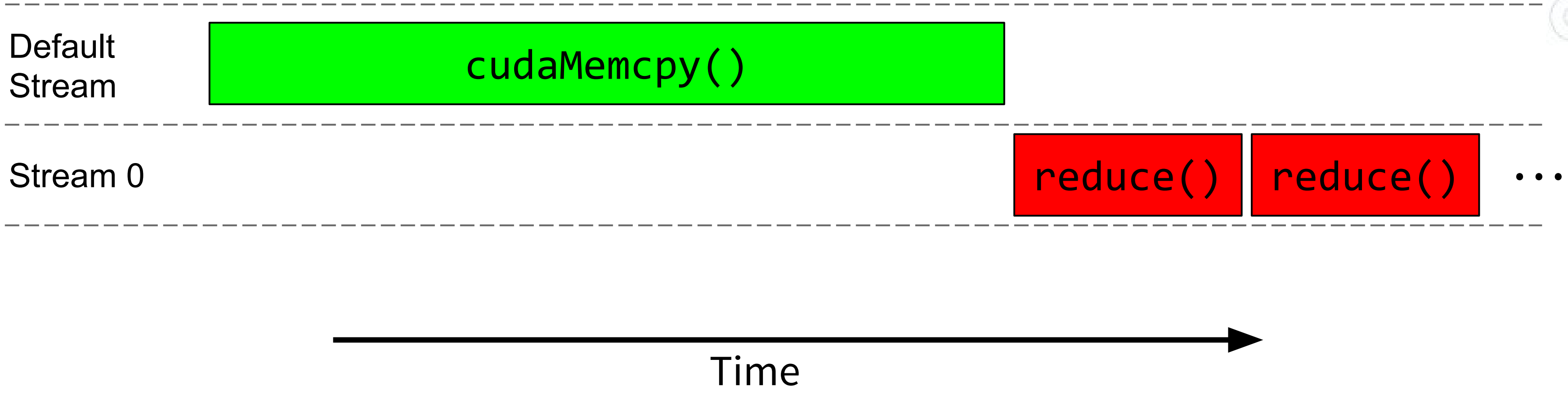
3. Using Streams



- We can create multiple streams...

```
cudaStream_t stream0;
status = cudaCreateStream(&stream0);
```

3. Using Streams



◎ ...and issue our CUDA calls into them

3. Using Streams

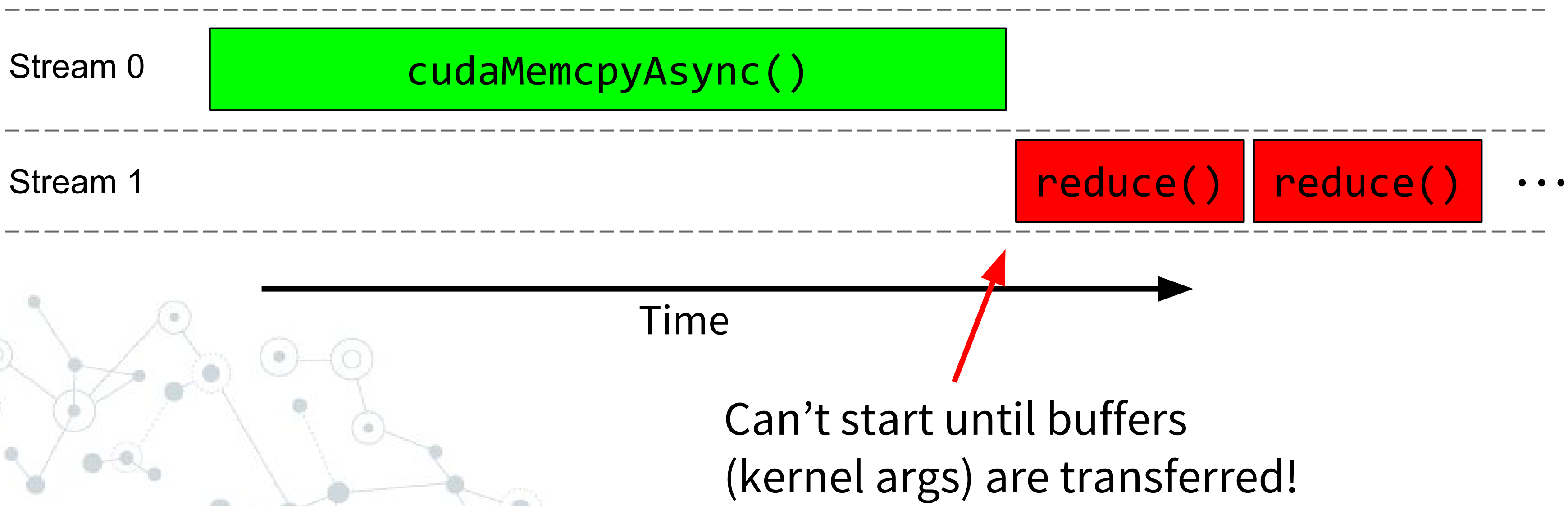
- How?

```
cudaMemcpyAsync(dest, src, size, cudaMemCpyHostToDevice, stream0);
reduce<<<blocks, 512, 0, stream1>>>(input_buf, output_buf, remaining);
```

- Both calls are *non-blocking*
  - Host will deposit call into stream and carry on

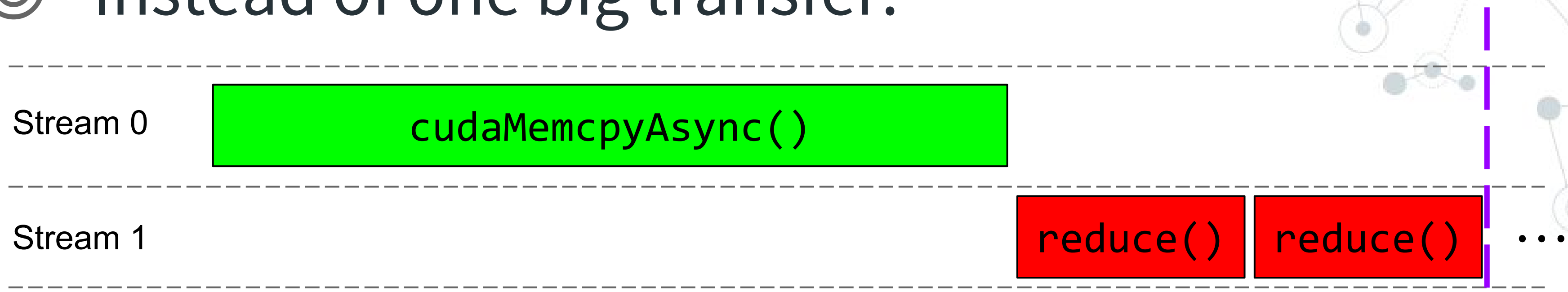
3. Using Streams

- GPU scheduler examines items in each stream
  - Picks up to 1 data transfer & 1 kernel to do next
- GPU *can* run data transfer & kernel concurrently!
  - Right now it doesn't…
  - To run kernel, all dependencies must be met
- How can we fix this?

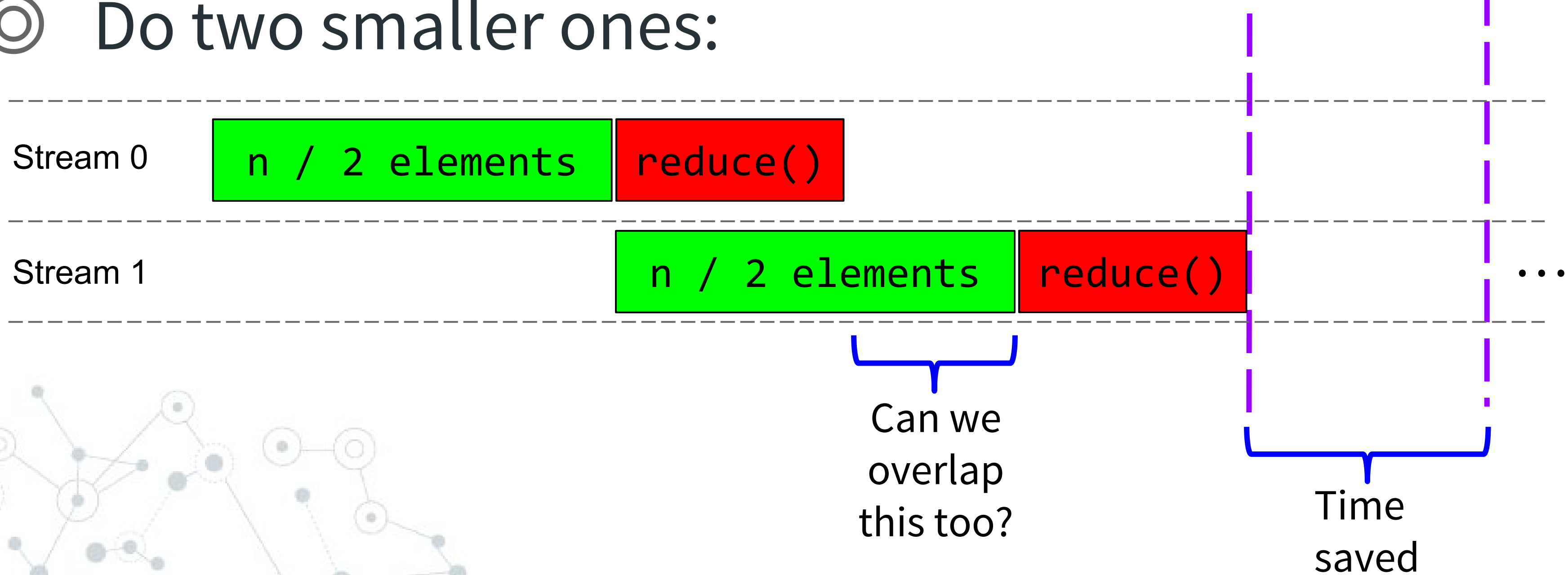


Idea: Partition the Transfer

- Instead of one big transfer:

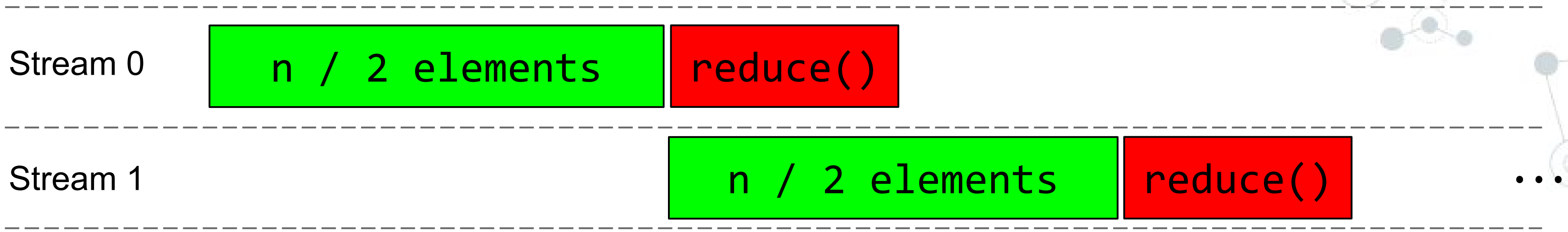


- Do two smaller ones:

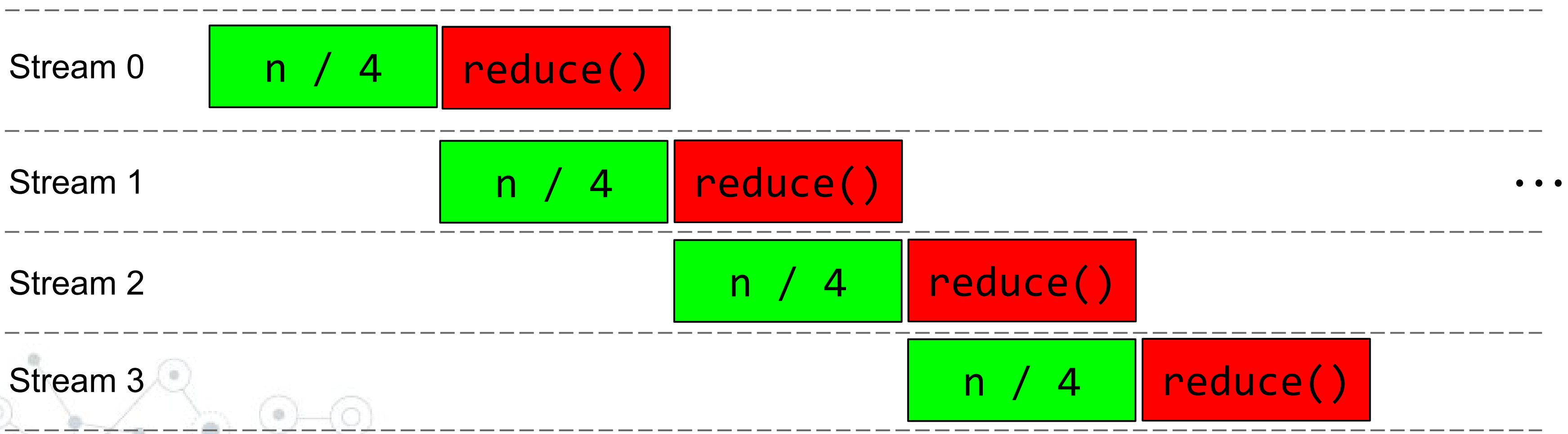


Rinse & Repeat

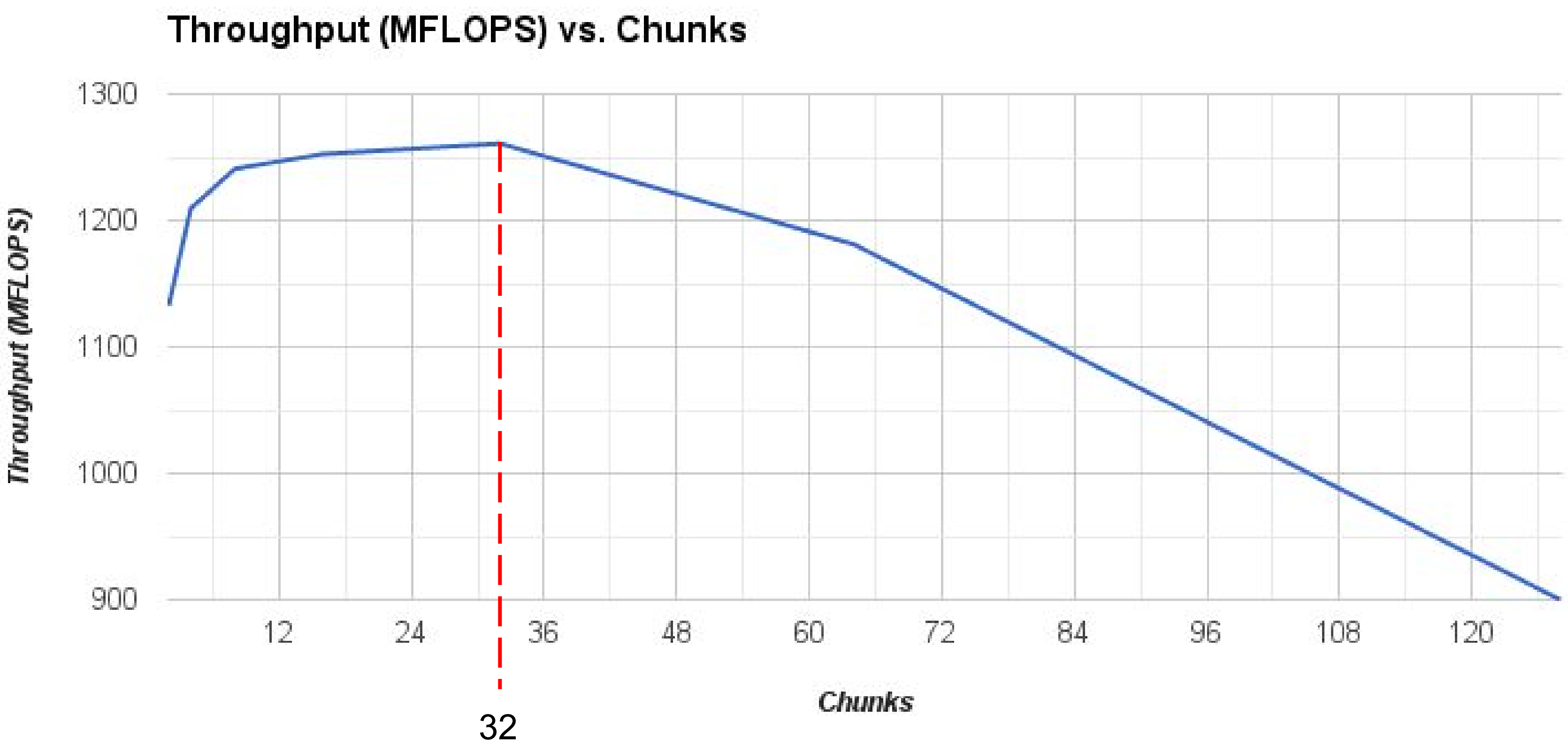
- Instead of using only two chunks:



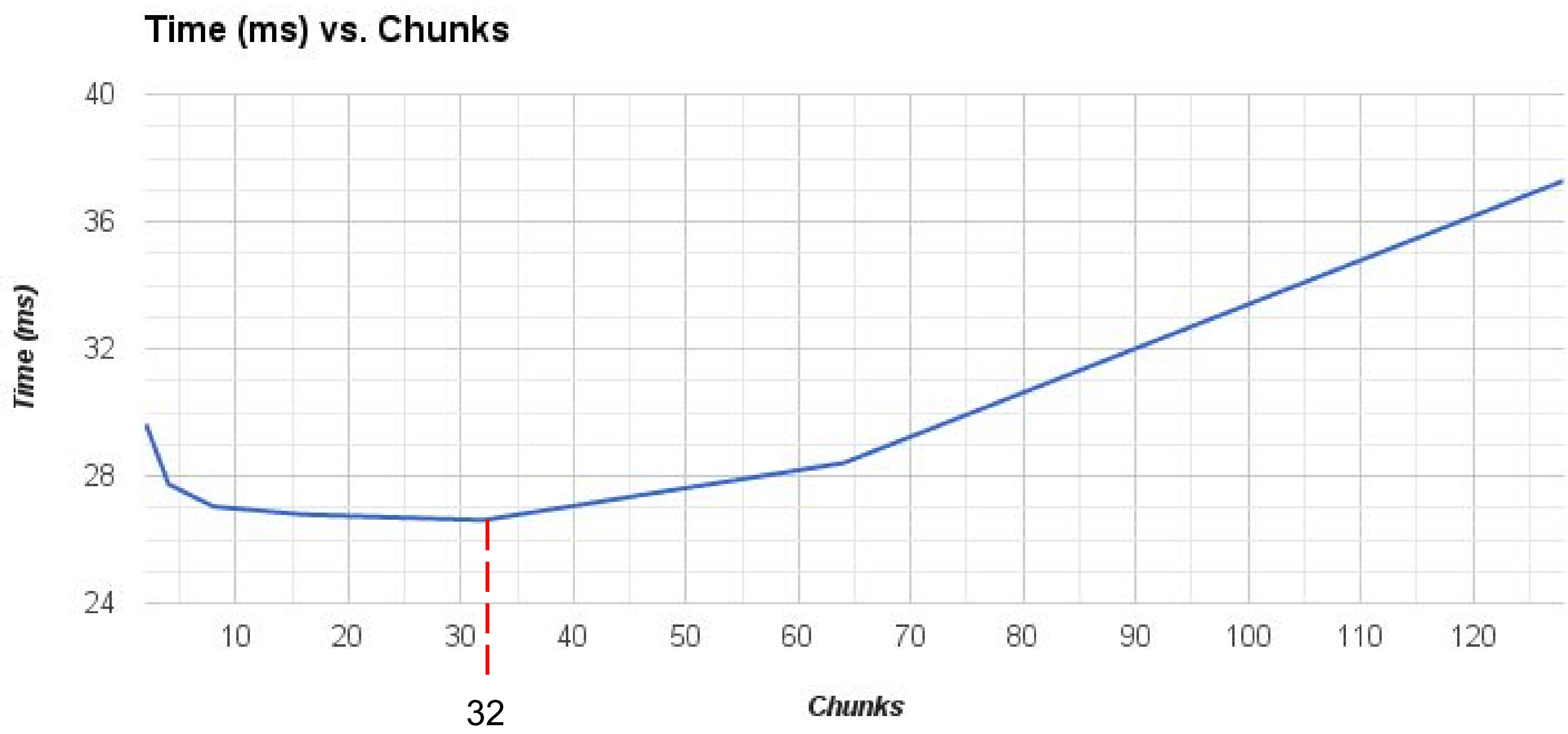
- Try using four:



Using Streams - Results



Using Streams - Results



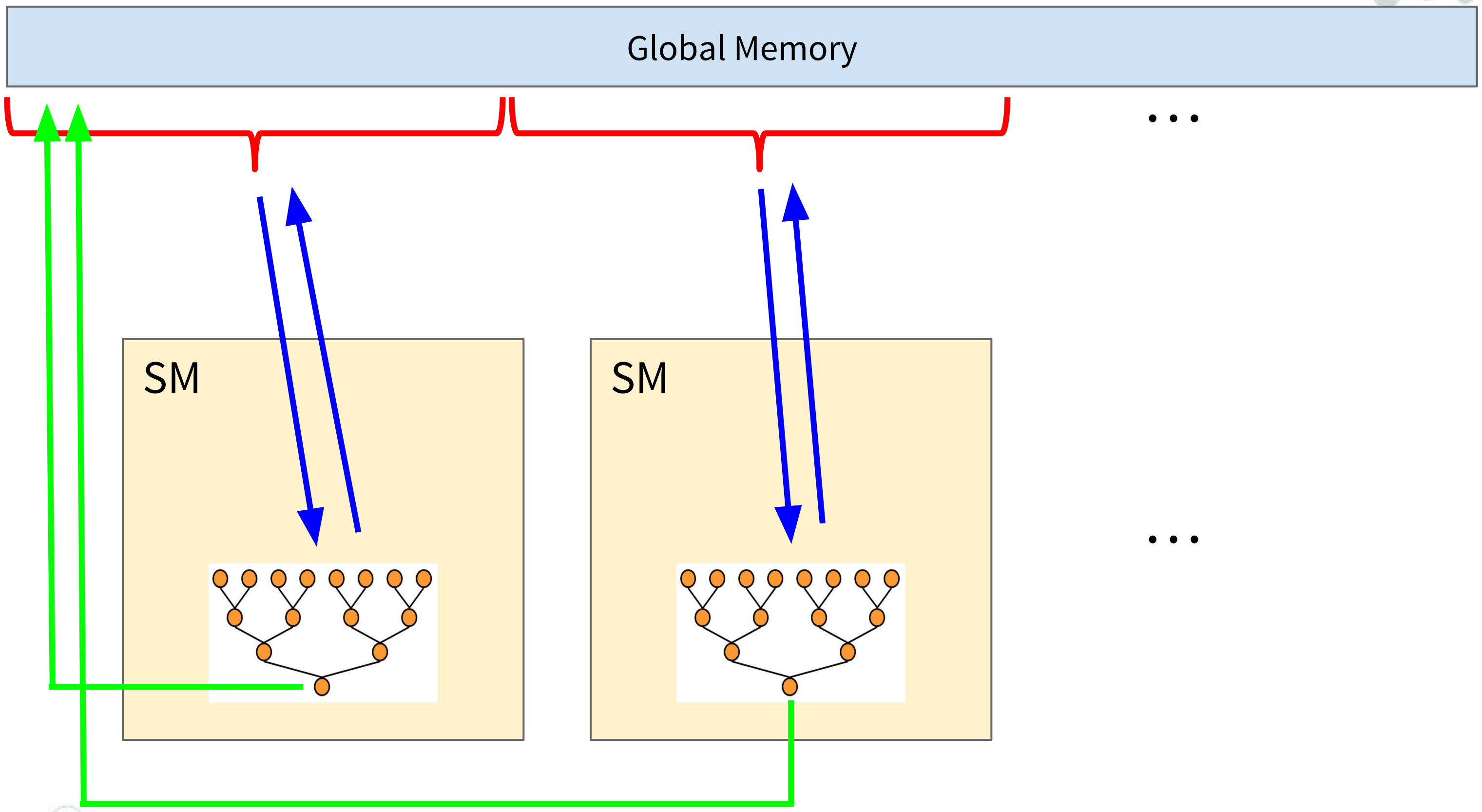
3. Using Streams - Results

Approach	Throughput (MFLOPS)	Improvement (factor)
CPU	558	
0. Initial Approach	500	-58 (0.9x)
1. Global Memory Coalescing	604	+104 (1.2x)
2. Using Pinned Memory	1041	+437 (1.7x)
3. Using Streams	**1265**	**+224 (1.2x)**

- What about shared memory?

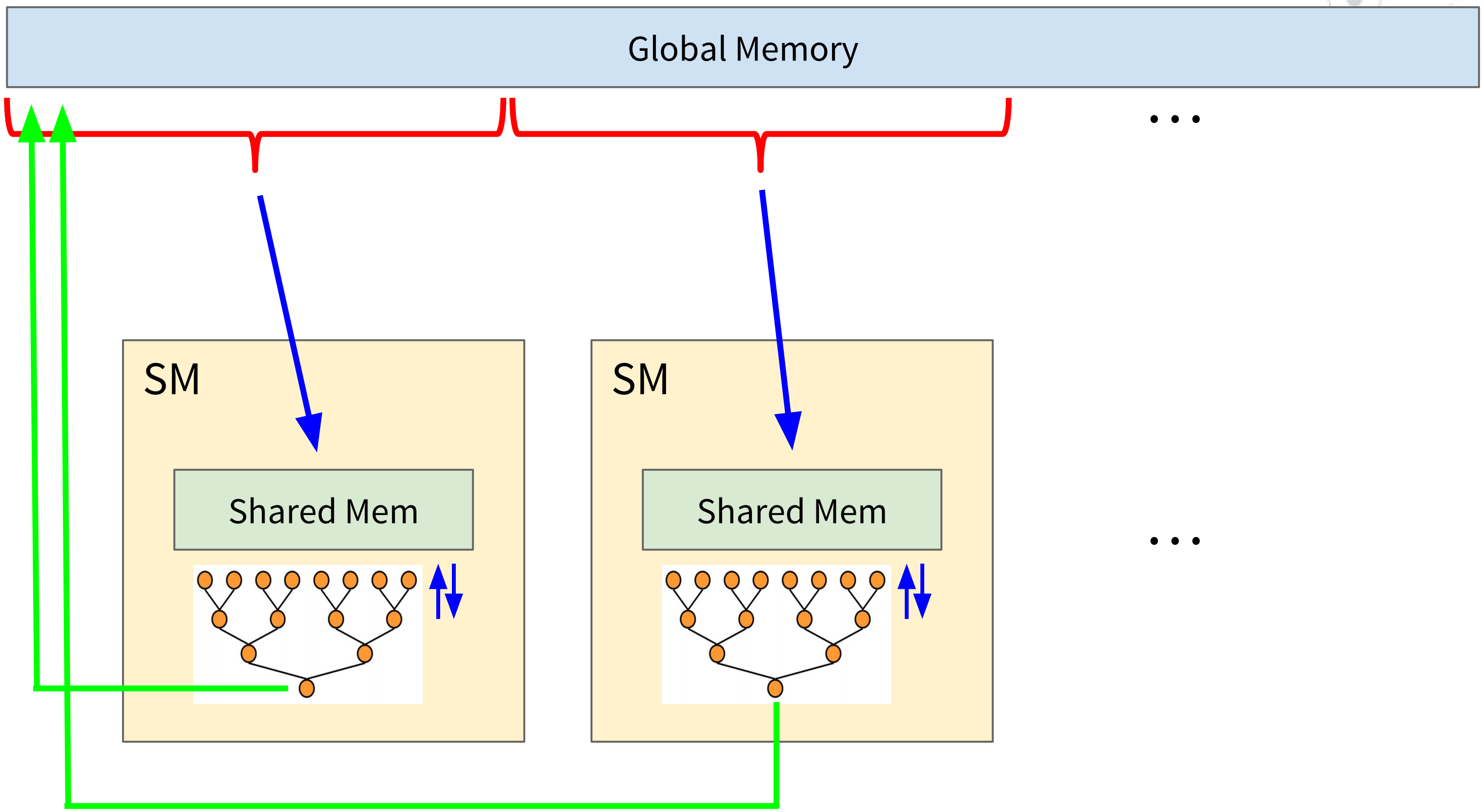
4. Using Shared Memory

- Right now, We're using Global Memory:



- Lots of global memory hits!

Idea



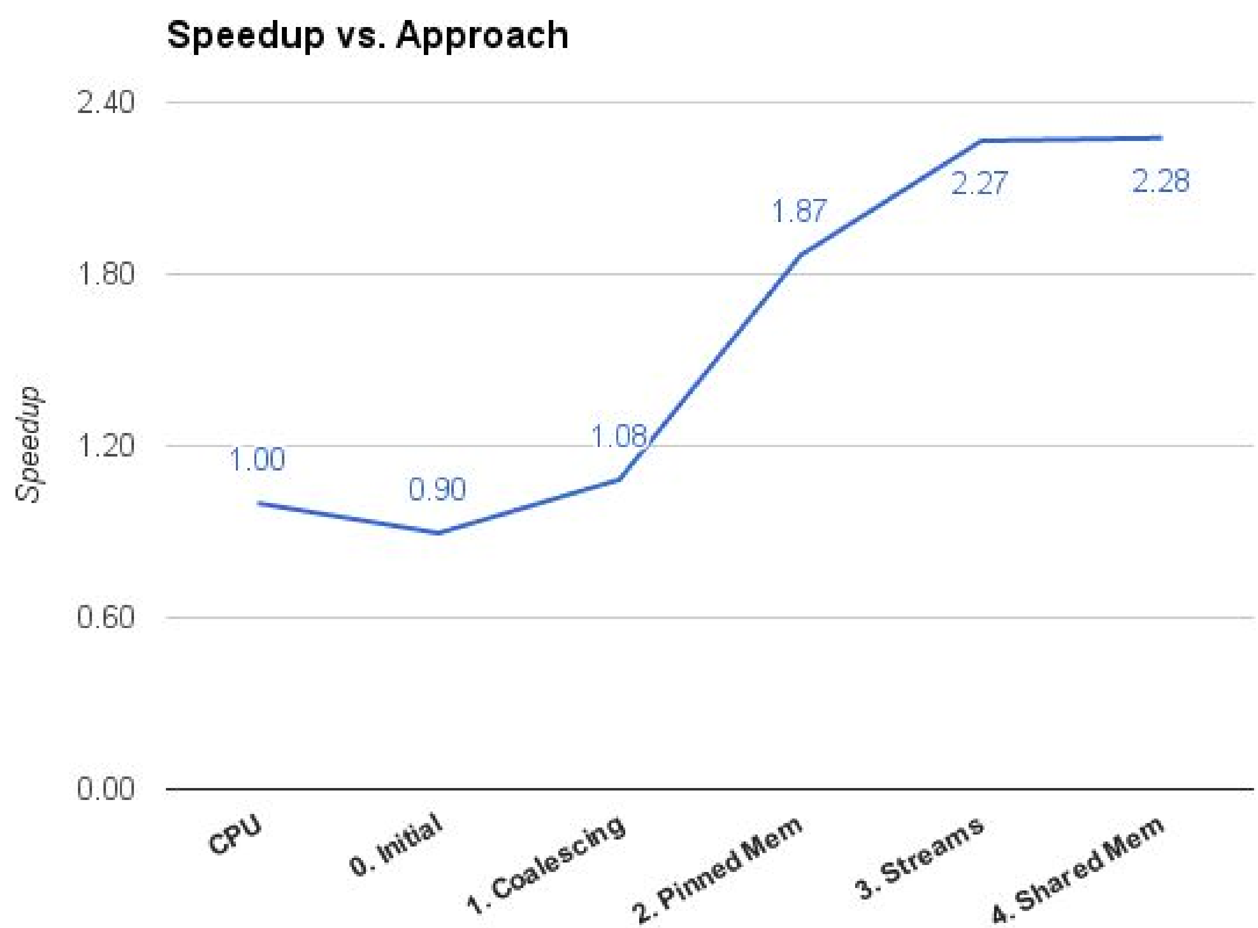
- Shared Mem doesn't stick around between kernel launches
    - copy back partial results after each host iteration

4. Using Shared Memory - Results

Approach	Throughput (MFLOPS)	Improvement (factor)
CPU	558	
0. Initial Approach	500	-58 (0.9x)
1. Global Memory Coalescing	604	+104 (1.2x)
2. Using Pinned Memory	1041	+437 (1.7x)
3. Using Streams	1265	+224 (1.2x)
4. Using Shared Memory	**1270**	**+5 (1.004x)**

- Small improvement!

Optimization Summary



- Note: We could keep going...

Final Thoughts

1. Computers are (in a sense) "towers of abstractions"
   - multiple layers...
   - As programmers we often tend to ignore anything below our level (software)
   - But as we've seen knowing about hardware makes a difference!
2. Parallel computing deals with the ***interaction between*** these levels of abstraction (h/w & s/w)

More information

- Nvidia's CUDA-C Programming Guide, esp. "Performance Guidelines" section
- Course materials made available by various Universities
- Nvidia Parallel Forall blog (lots of applications of GPGPU)
- More sum reduction optimizations (loop unrolling)

Fin

Image Sources

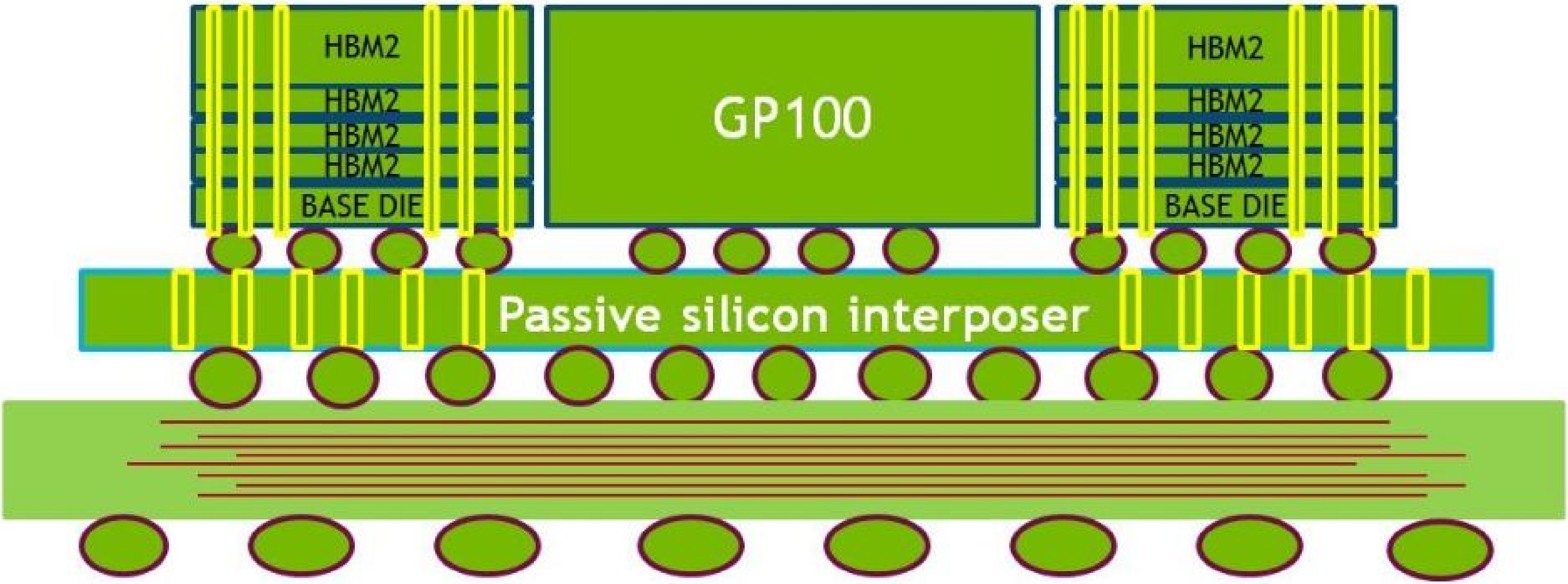
- http://cms.ipressroom.com.s3.amazonaws.com/219/files/20165/5767cb5d2cfac26f9e698465_97392_tesla1/97392_tesla1_169bb0b8-1ef0-4bd2-b8f9-e7cfb62b8884-prv.jpg
- http://images.anandtech.com/doci/10222/P100_678x452.jpg
- http://www.stoimen.com/blog/wp-content/uploads/2012/11/3.-Matrix-Multiplication.png
- http://cdn.wccftech.com/wp-content/uploads/2014/10/27151_1_intel_rejects_the_idea_that_they_are_going_bga_only_full.jpg
- http://www.nvidia.com/docs/IO/59921/NV_CUDA_2D_Color_large.jpg
- http://www.opencl.org/opencl_logo.jpg
- http://www.nvidia.co.uk/content/EMEAI/images/tesla/openacc-logo.png
- https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
- https://en.wikichip.org/wiki/File:broadwell_block_diagram.svg
- http://images.anandtech.com/doci/7793/cudacore.jpg
- http://cms.ipressroom.com.s3.amazonaws.com/219/files/20149/NVIDIA_CUDA_V_2C_r.jpg
- http://caig.cs.nctu.edu.tw/course/CG2007/images/ex1_wireframe.jpg
- https://upload.wikimedia.org/wikipedia/commons/thumb/5/5f/Utah_teapot_simple_2.png/220px-Utah_teapot_simple_2.png
- http://images.clipartpanda.com/eye-clip-art-1384151134.png
- http://www.techspot.com/articles-info/650/images/ibm-pc-mda.jpg
- http://blog.biipmi.com/wp-content/uploads/2013/01/balanced_scale_of_justice.png
- http://static.betazeta.com/www.fayerwayer.com/up/2009/12/intel48coreprocessor_5-960x623.jpg
- http://i.imgur.com/gO8ueXc.png?1?fb
- http://images.clipartpanda.com/screen-clipart-bcyLd8zcL.jpeg
- http://news.sciencemag.org/sites/default/files/styles/thumb_article_l/public/media/sn-memory.jpg?itok=vz4-1TiX
- http://petful.supercopyeditors.netdna-cdn.com/wp-content/uploads/2012/06/why-is-cat-scared-rain-thunder.png
- https://blogs.nvidia.com/wp-content/uploads/2015/03/cudablock.jpg
- http://sbel.wisc.edu/Courses/ME964/2013/Lectures/lecture1011.pdf
- http://www.flixist.com/ul/206006-12-best-jackie-chan-fight-scenes/jackie-chan-best-fight-scenes-future-620x.jpg
- http://image.slidesharecdn.com/s0514-gtc2012-gpu-performance-analysis-140731021114-phpapp02/95/gpu-performance-analysis-26-638.jpg?cb=1406772984
- http://openclipart.org
- http://i1-news.softpedia-static.com/images/news2/CES-2015-NVIDIA-GeForce-GTX-960-Graphics-Card-468509-2.jpg
- http://www.colorfulchildhoodstore.com/v/vspfiles/photos/Wall%20Fabric%20-%20Tree%20of%20Wisdom-3.jpg
- https://www.rspb.org.uk/Images/cuckoo_grey_tcm9-58729.jpg?width=530&crop=(596,980,2608,2112)
- http://roulerenligne.com/wp-content/uploads/2015/07/fin.jpg
- https://devblogs.nvidia.com/parallelforall/wp-content/uploads/2012/12/pinned-1024x541.jpg
- http://blog.scoutapp.com/articles/2011/02/10/understanding-disk-i-o-when-should-you-be-worried

Information Sources

- Nvidia CUDA-C Programming Guide, 2016
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- Dan Negrut, University of Wisconsin-Madison, High Performance Computing for Engineering Applications (Course lecture notes), 2013
  - http://sbel.wisc.edu/Courses/ME964/2013/Lectures/lecture1011.pdf
- Others:
  - https://www.microway.com/hpc-tech-tips/intel-xeon-e5-2600-v3-haswell-processor-review/
  - https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
  - http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf
  - http://ubiquity.acm.org/article.cfm?id=1513451
  - https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

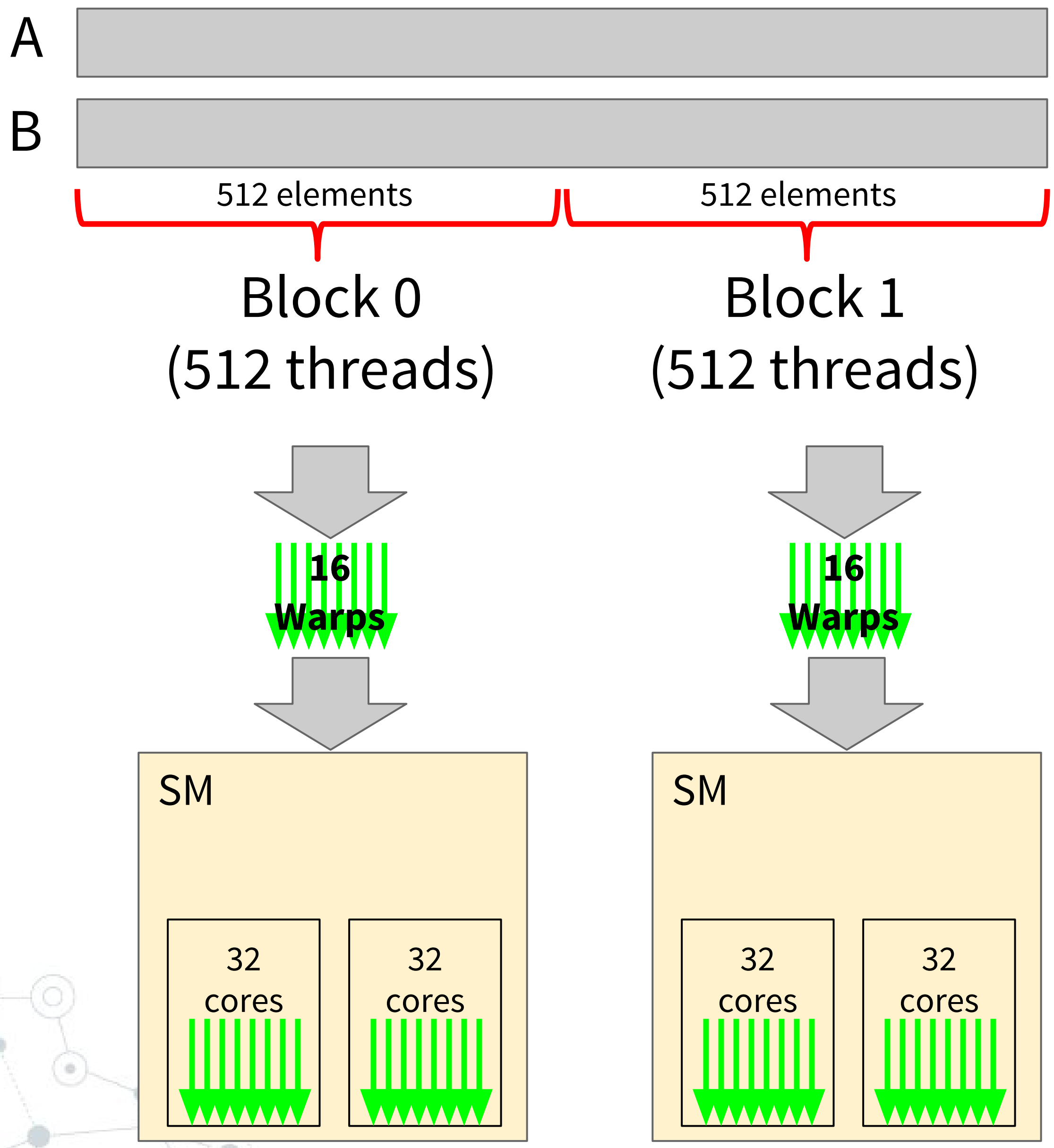
Extras

High Bandwidth (Stacked) Memory



Scheduling

n = 1024



2 / 27 SMs occupied

Scheduling

n = 1024