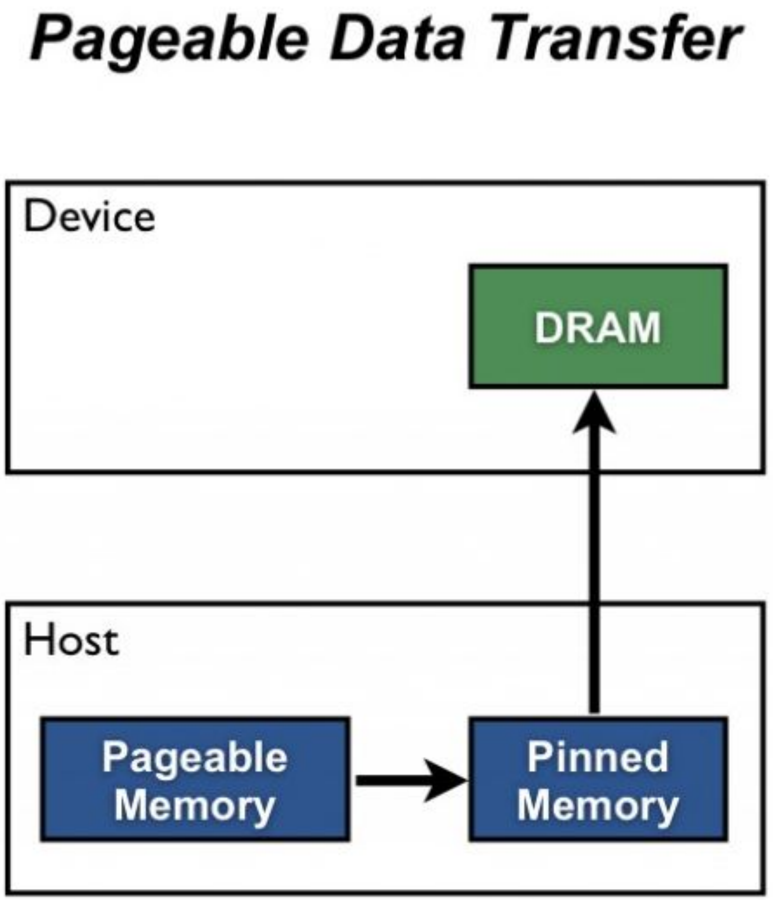
2. Using Pinned Memory

- OS uses virtual memory
  - Memory is segmented into “pages”
  - Can be “swapped out” to disk
  - Disk is significantly slower than memory (several orders of magnitude)

2. Using Pinned Memory

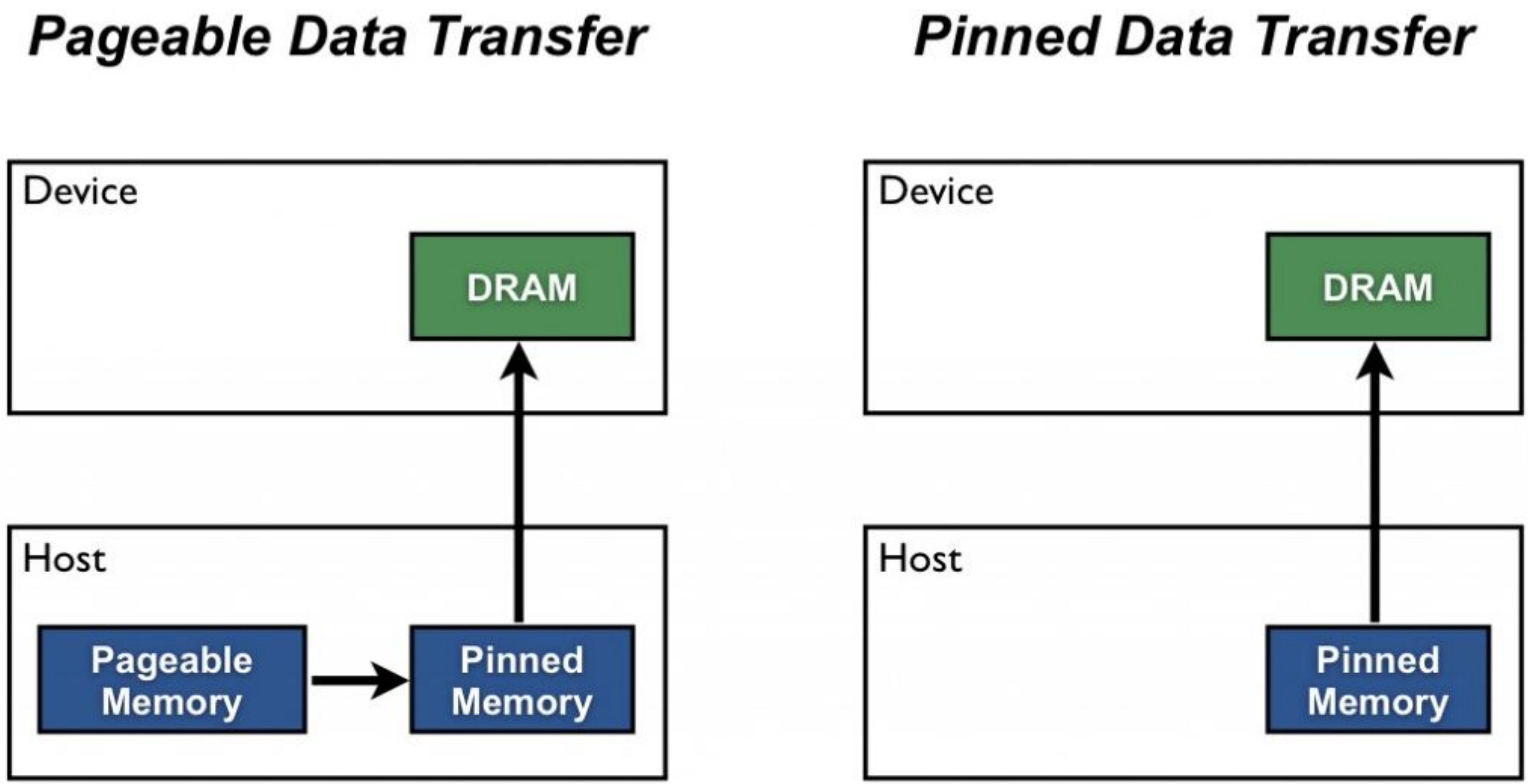
- Before data transfer to GPU, buffers must first be copied to non-pageable memory



Images: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

2. Using Pinned Memory

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



Images: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

How?

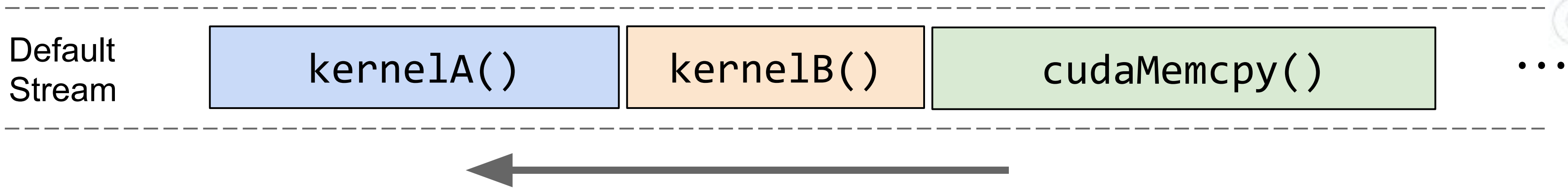
- Instead of `malloc()`ing host buffers, use `cudaMallocHost()`
- Instead of `free()`, use `cudaFree()`

2. Using Pinned Memory - Results

Approach	Throughput (MFLOPS)	Improvement
CPU	504	
0. Initial Approach	1911	1407
1. Global Memory Coalescing	1978	67
2. Using Pinned Memory	**2687**	**709**

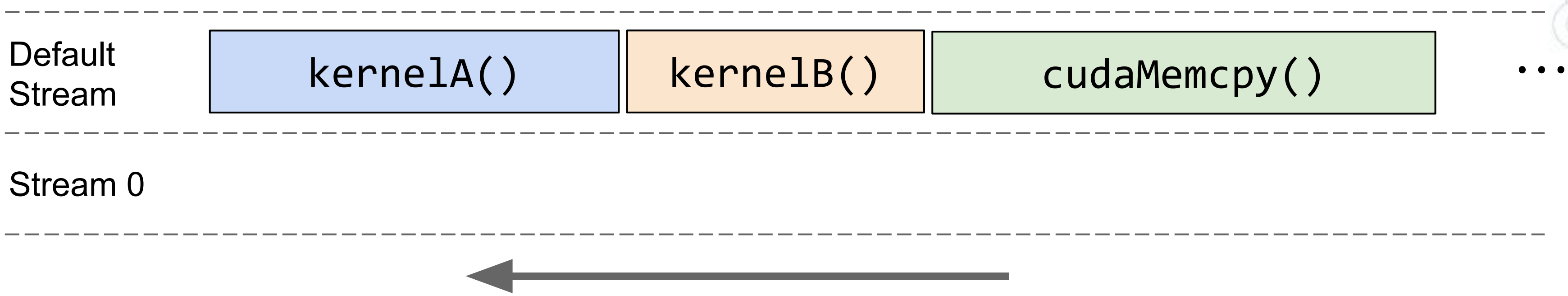
- Since transfers occupy such a high percentage of our execution time, speeding them up makes a ***big*** difference
- Transfer time ***still*** outweighs kernel time though...

3. Streams



- Stream: a queue containing pending CUDA calls

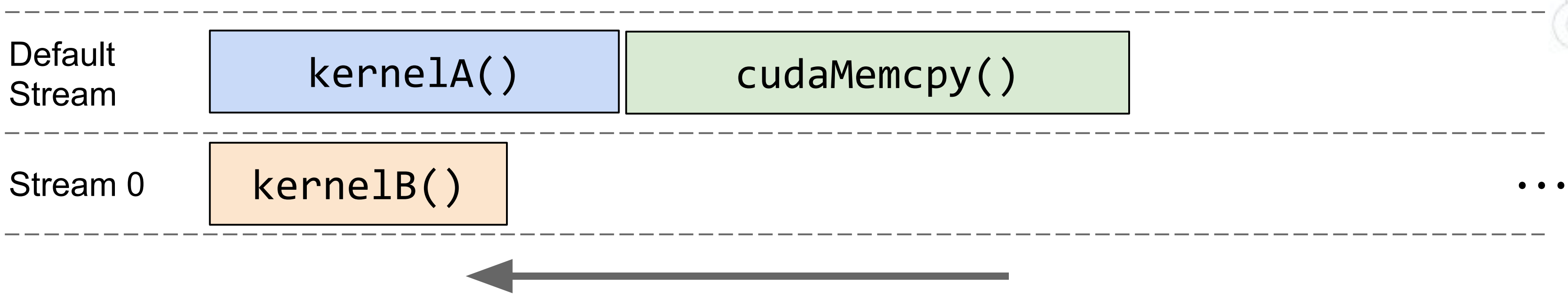
Creating Streams



- We can create additional streams...

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

Using Streams



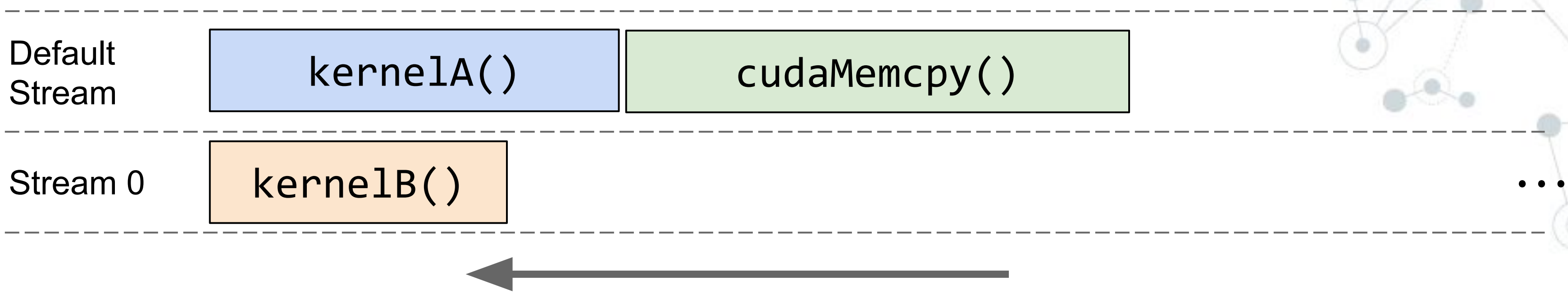
- ...and issue our CUDA calls into them

```
kernelB<<<blocks, bsize, 0, stream0>>>();
```

Dynamic Shared Memory (not covered in this course)

Stream to use (if omitted, default stream is used)

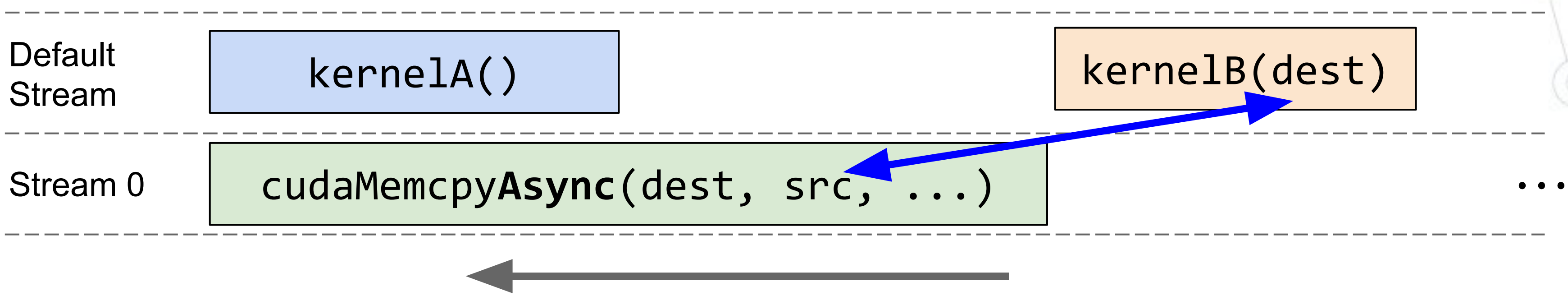
Concurrent Execution



- GPU scheduler examines items at front of all stream queues
  - and tries to run them concurrently if possible
- Kernels can be run simultaneously if there are enough resources (Eg. SMs)

Concurrent Execution

kernelB must wait until the transfer completes because it uses the dest buffer



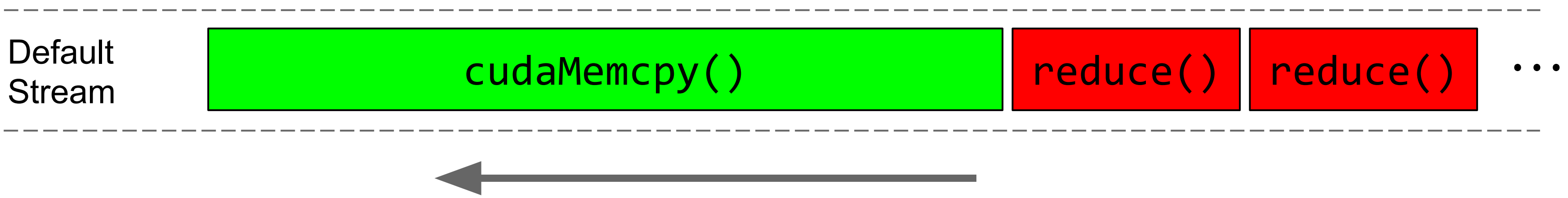
- Data transfers and kernels can run concurrently if the kernel doesn't use the data being transferred
  - **Note:** cudaMemcpy() is blocking
  - Use cudaMemcpy**Async**() instead

```
cudaMemcpyAsync(dest, src, size, cudaMemCpyHostToDevice, stream0);
```

Pointer to stream

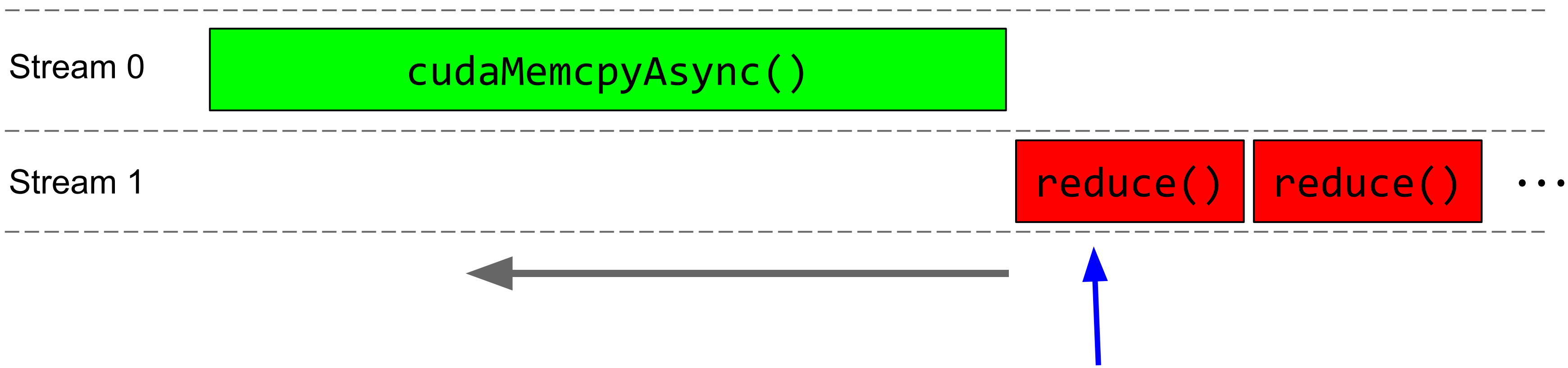
Our Situation

- We're transferring data, then calling `reduce()` repeatedly:



Data Dependency

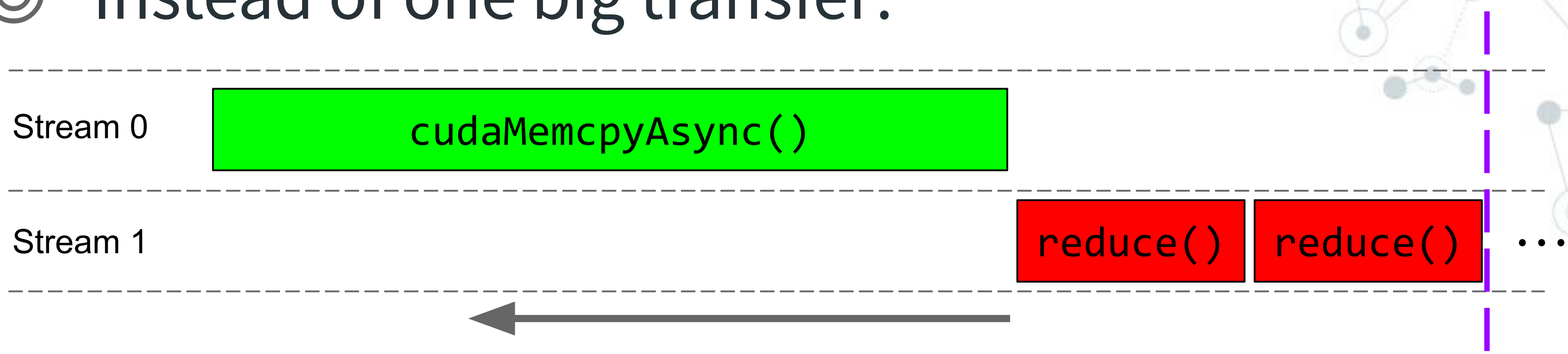
◎ **Problem**: We can create another stream, but we can't overlap `memcpy()` and `reduce()`



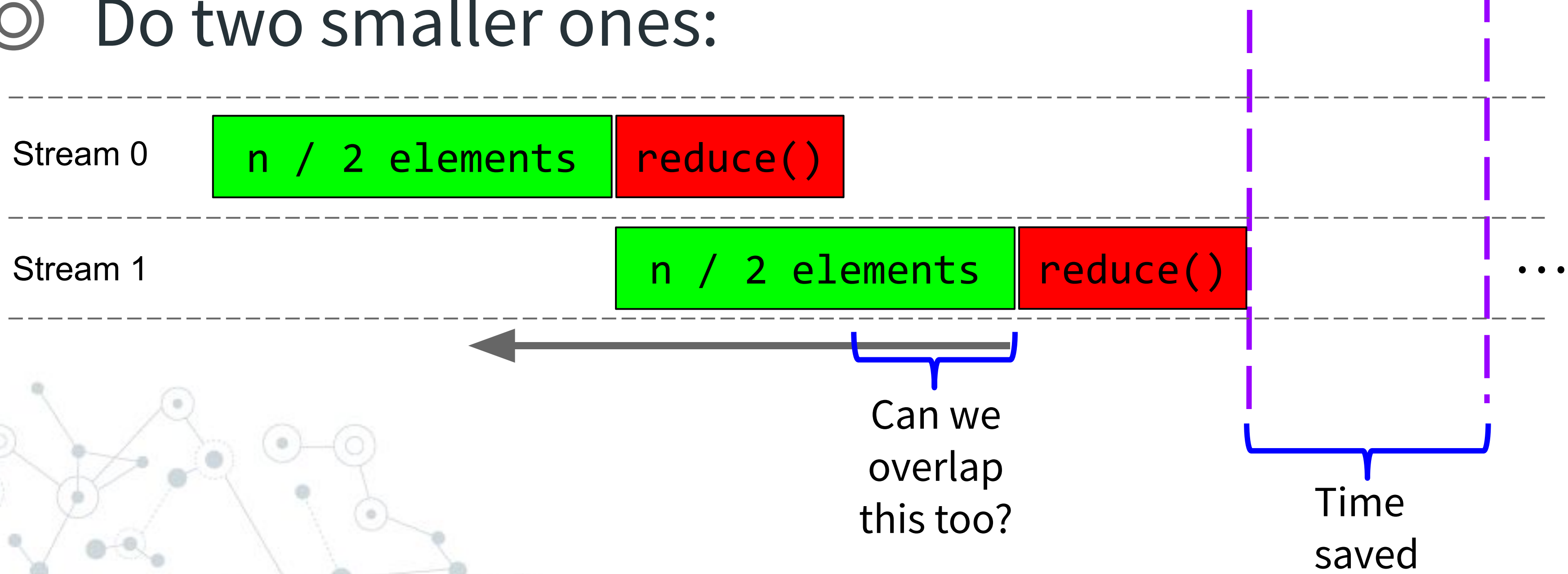
Can’t start kernel until buffers (kernel args) are transferred!

Idea: Partition the Transfer

- Instead of one big transfer:

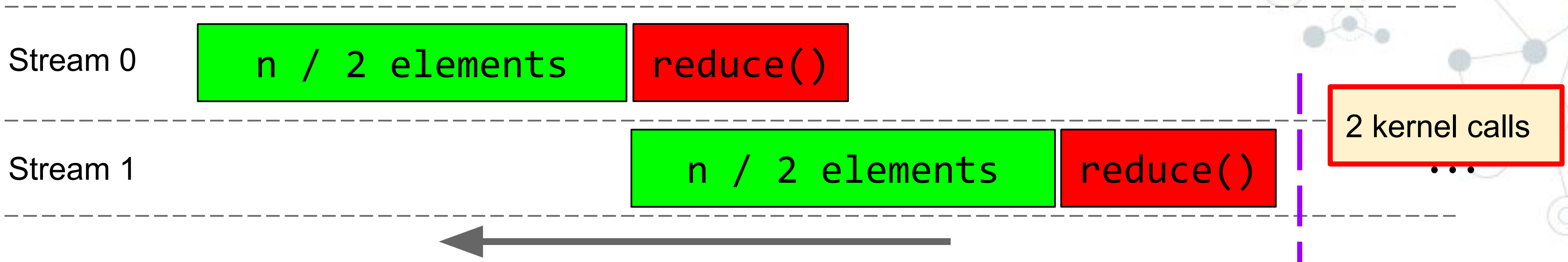


- Do two smaller ones:

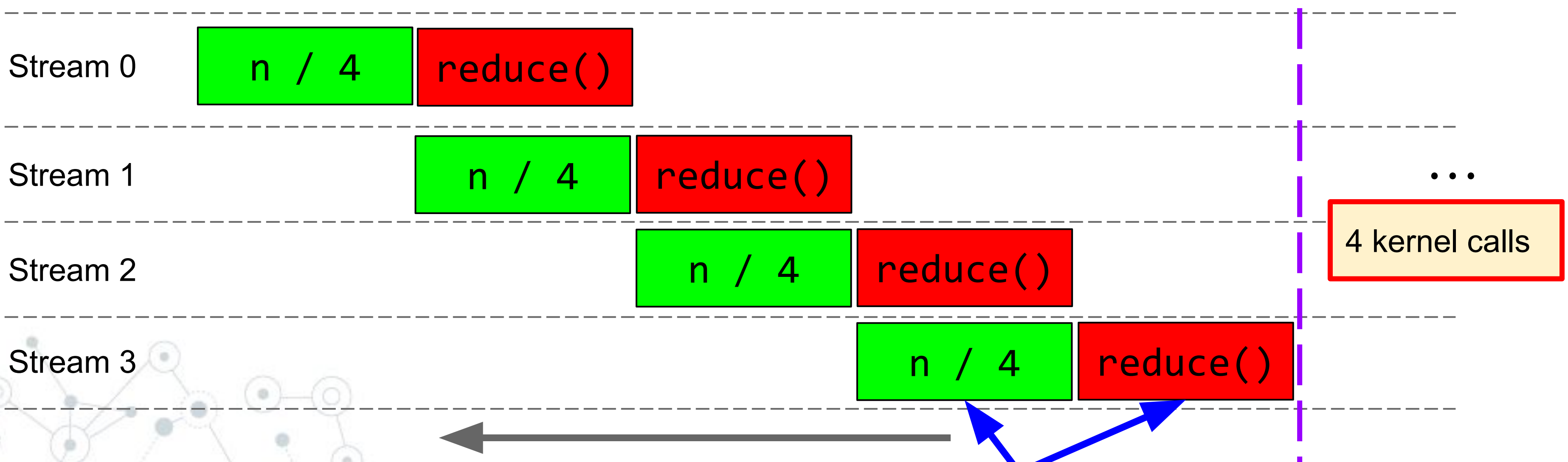


Adjusting the chunk size

- Instead of using only two chunks:

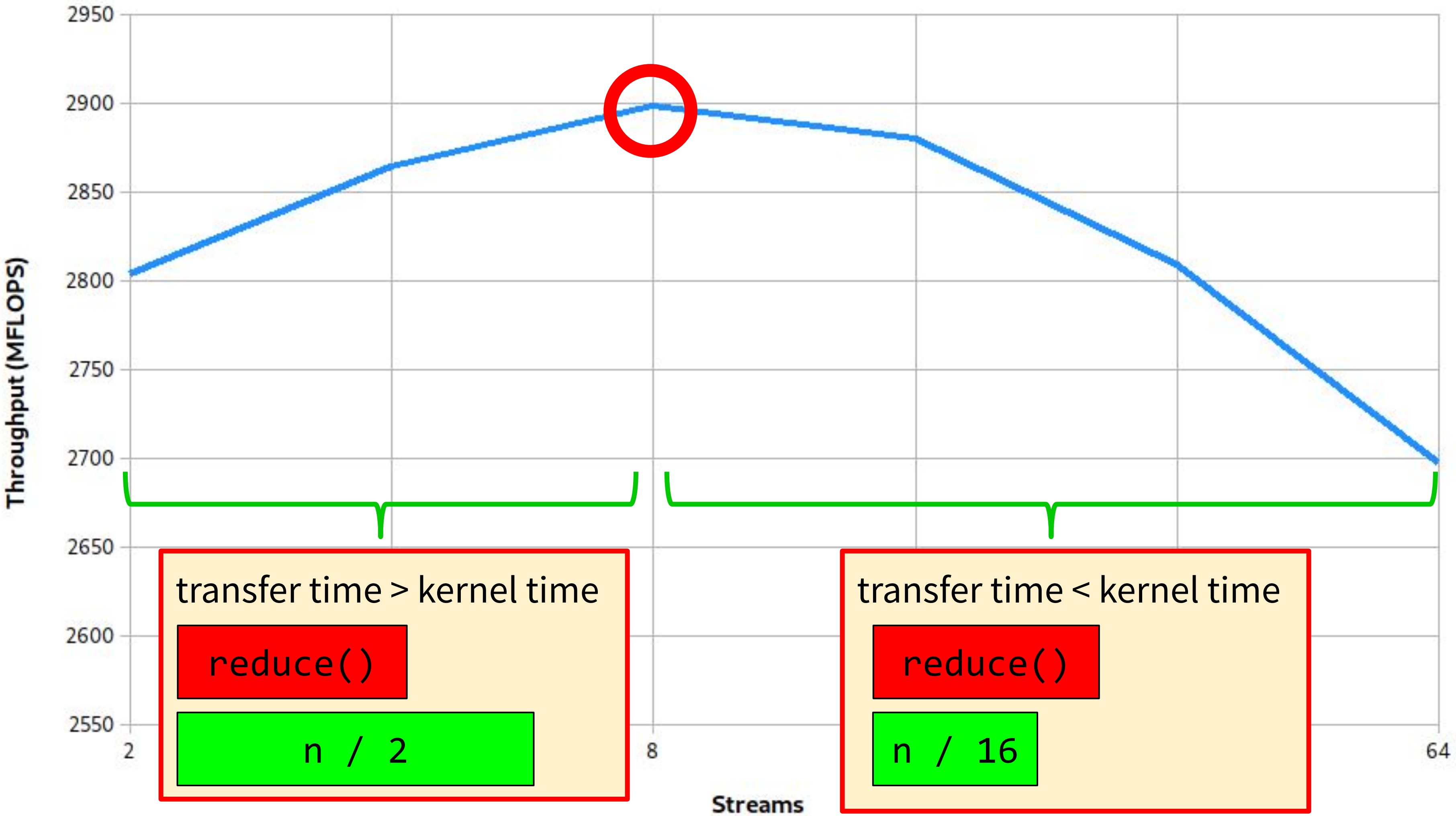


- Try using four:



Goal: Keep decreasing chunk size until these are the same length.

Scaling up the Number of Streams



(Chunk size is n / # of streams)

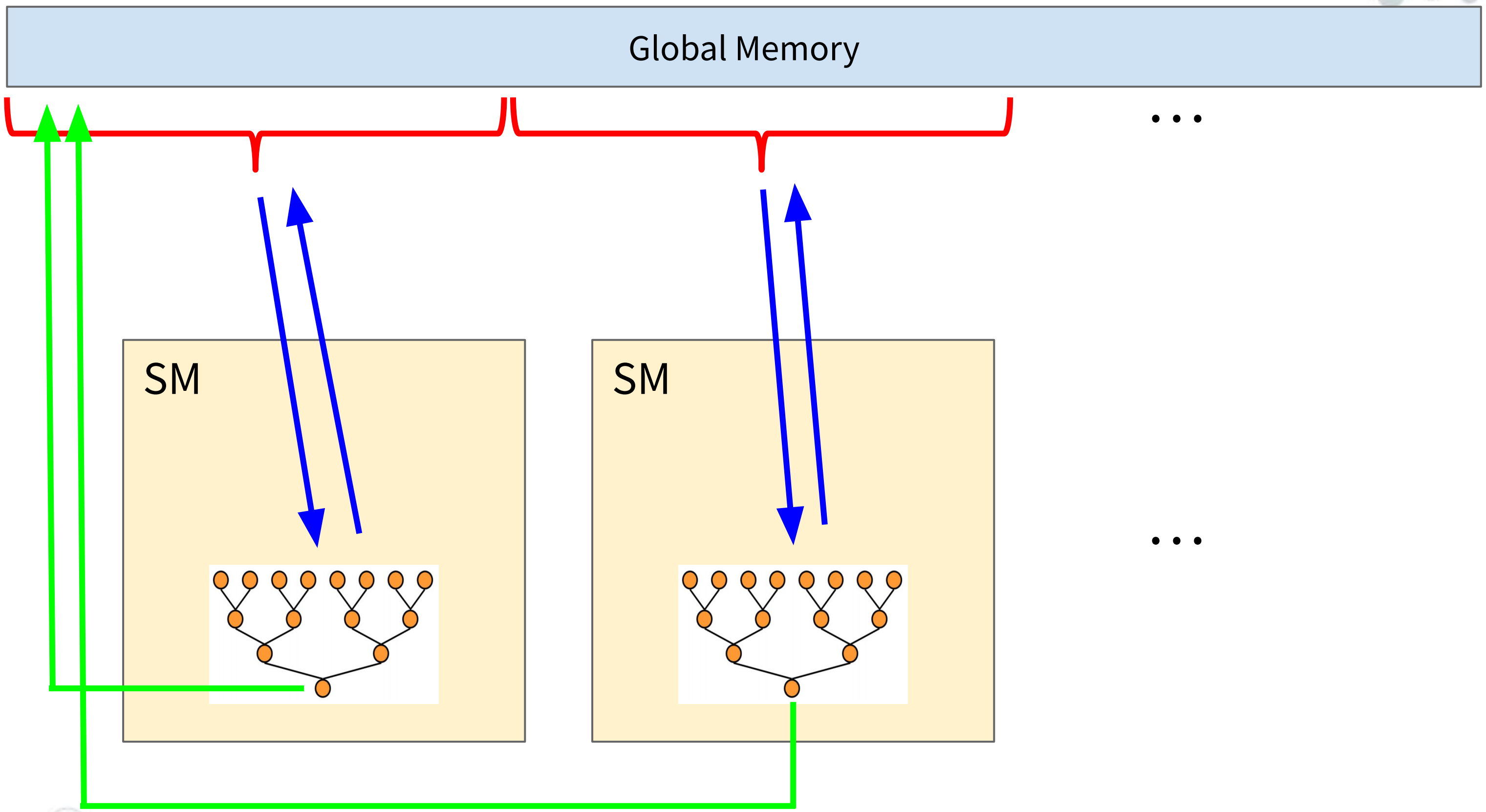
3. Using Streams - Results

Approach	Throughput (MFLOPS)	Improvement
CPU	504	
0. Initial Approach	1911	1407
1. Global Memory Coalescing	1978	67
2. Using Pinned Memory	2687	709
3. Using Streams	**2898**	**211**

- Note: We are ***not reducing*** transfer time; we are ***overlapping*** the transfer with computation

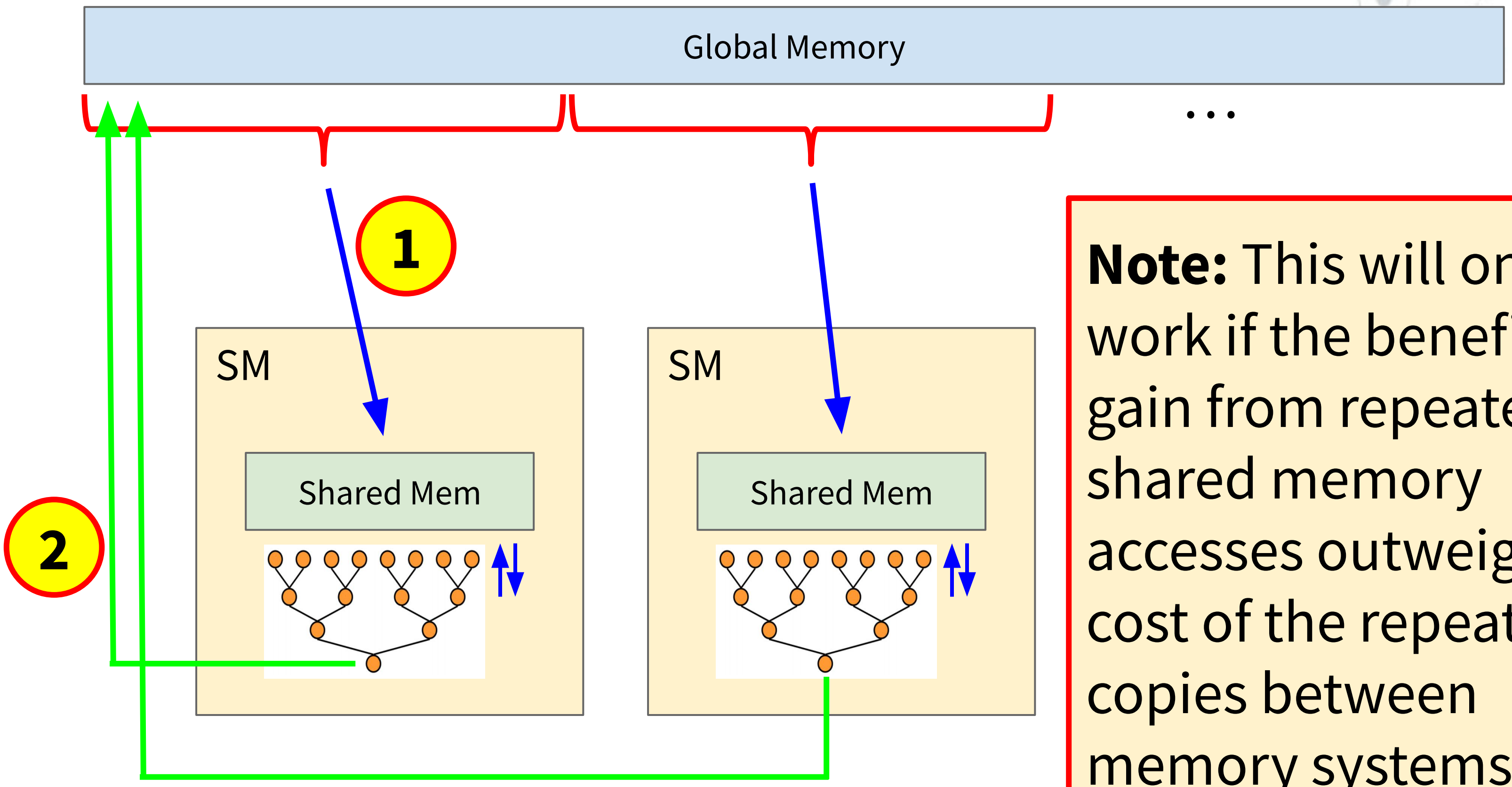
4. Using Shared Memory

- Right now, We're using Global Memory:



- Lots of global memory accesses!

Idea



Note: This will only work if the benefit we gain from repeated shared memory accesses outweighs the cost of the repeated copies between memory systems.

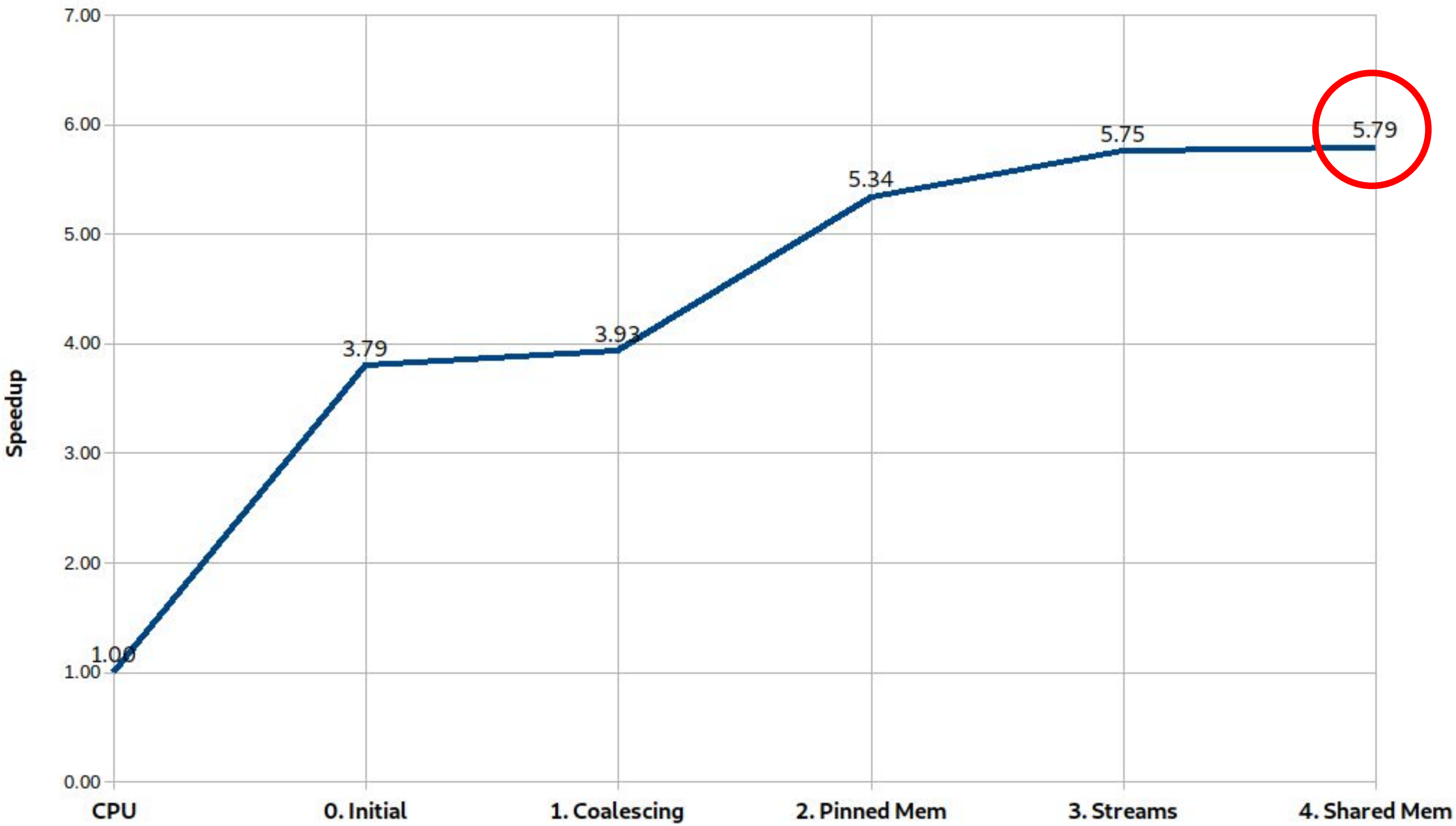
- Shared Mem doesn't stick around between kernel launches
  - copy back partial results after each host iteration

4. Using Shared Memory - Results

Approach	Throughput (MFLOPS)	Improvement
CPU	504	
0. Initial Approach	1911	1407
1. Global Memory Coalescing	1978	67
2. Using Pinned Memory	2687	709
3. Using Streams	2898	211
4. Using Shared Memory	**2917**	**19**

- Small improvement!
  - if we were doing more shared memory accesses after the copy, we'd see a greater benefit here

Optimization Summary

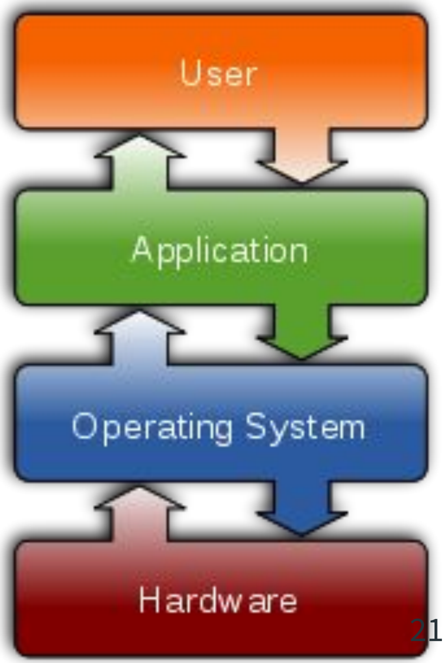


- Note: We could keep going...

Final Thoughts

1. Computers are "towers of abstraction layers"
   - As programmers we often tend to ignore anything below (or above) our level (software)
   - But as we've seen, knowing about hardware makes a difference!

2. Parallel computing deals with the ***interaction between*** these layers of abstraction
   - Parallel Programmers can benefit from the ***ability to move between layers***



More information

- Nvidia's CUDA-C Programming Guide, esp. "Performance Guidelines" section
- Nvidia Developer Blogs (lots of applications of GPGPU)
- Course materials made available by various universities