CUDA

Introduction

Programming model specific to NVIDIA GPUs.

GPU Design Tenets

Core GPU design tenets are as follows…

1. Many simple compute units vs few powerful compute units
   GPUs trade control for simpler computational units. For the programmer, that just means

that you're more restricted in the way you write your programs.

2. Explicitly parallel programming model
   You have to model your software such it makes use of lots of execution threads. It isn't like programming for the CPU where you're more likely to write large chunks of code intended to run on a single thread.

   Unlike programming for the CPU, the GPU is efficient at launching lots of threads and running them all in parallel. If you aren't running lots of threads, you aren't using the GPU effectively: "the GPU doesn't even get out of bed in the morning for fewer than 1000 threads."

3. Optimized for throughput
   GPUs are optimized to run many tasks at once, but not necessarily to run those tasks quickly.

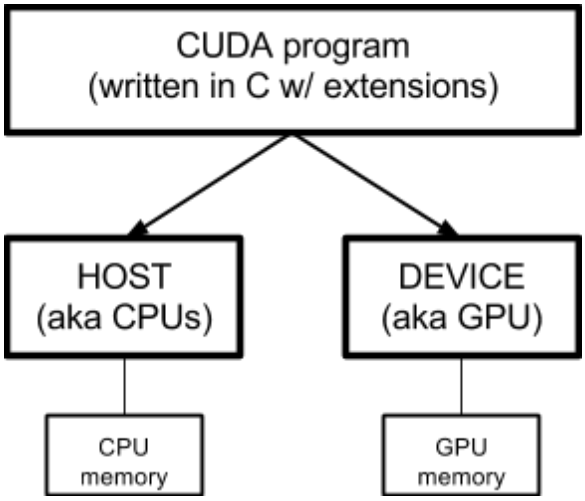
Terminology

- **HOST** → PC that the GPU(s) are in.
- **DEVICE** → GPU.
- **KERNEL** → program to run on GPU (multiple copies of the same kernel run in parallel).
- **THREAD** → execution thread of our kernel (just like a CPU thread)
- **BLOCK** → a block can run up to n threads of your kernel
- **SM** → streaming multiprocessor (hardware component) that executes blocks
- **WARP CORE** → SMs are made up of many warp cores (see thread divergence section)
- **SIMD** → single instruction, multiple dataset (see thread divergence section)
- **SIMT** → single instruction, multiple threads (see thread divergence section)
- **BARRIER** → thread synchronization pattern (wait for all threads to reach some point before continuing), see memory synchronization sections
- **ARRAY OF STRUCTS** (AOS) → see transpose pattern section
- **STRUCT OF ARRAYS** (SOA) → see transpose pattern section
- **STEP COMPLEXITY** → see complexity section
- **WORK COMPLEXITY** → see complexity section
- **WORK EFFICIENT**→ see complexity section
- **WEAK SCALING** → using parallelism to scale a problem so larger/more problems can be run at once
- **STRONG SCALING** → using parallelism to scale a problem so it can be run faster
- **OCCUPANCY** → number of threads running per SM vs number of threads SM is capable of running

Programming Basics

Access Pattern

CUDA's programming model essentially looks like the following….



CUDA's tools allow you to write a single C program and target parts of your code to run on the GPU (called device) instead of your CPU (called host). You can write your program as a single whole C program, and the parts identified to run on the GPU (called kernels) will be compiled accordingly to run by the CUDA compiler.

Important things to note here…

1. The host (CPU) is in charge -- it runs the program and it gives directions to the device (GPU) on what to do: run task, copying to, copy from, etc...
2. The host (CPUs) and the device (GPUs) have separate memory -- there is no implicit way to view the memory between the two, you have explicitly shuttle data back and forth.

> **NOTE**: Although multiple languages are supported, CUDA's main language is C.
>
> **NOTE**: CUDA's "compiler" actually delegates complication of CPU-side C code to either GCC or MSVC. The only compiling that it actually does is the GPU portion?

Workflow

The typical workflow for a CUDA program is as follows…

1. Host (CPU) allocates storage on the device (GPU)
2. Host (CPU) copies input data to the device (GPU)

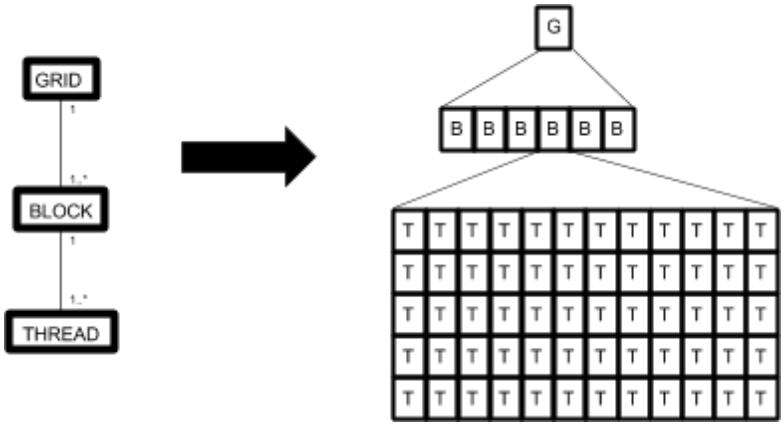
3. Host (CPU) launches kernel(s) on the device (GPU) to process that input data and produce output data
4. Host (CPU) copies the output data from the device (GPU)

Obviously, there's overhead to copying data back and forth. As such, your typical GPU program should have a high ratio of computation to communication -- you send your data, do a lot of work, then grab the resulting data.

Programming Model

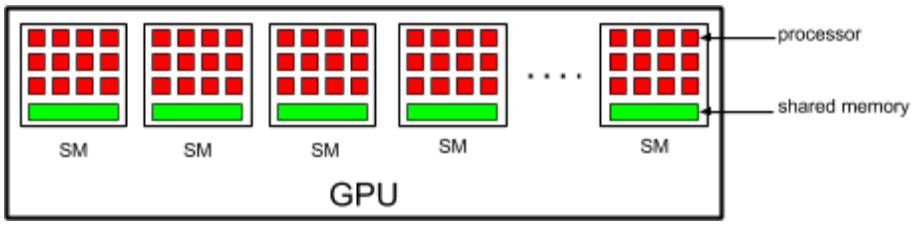
Software View

In software, you organize your execution threads to run as part of one or more blocks. Those blocks make up a grid.



Hardware View

In hardware, the part of the NVIDIA GPU that executes your kernel is called an SM (streaming multiprocessor). There are multiple SMs in each GPU, and each of them is capable of running some fixed number of threads at once + it has some small amount of fixed memory that's local to it.



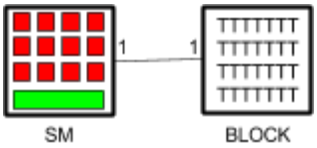
NOTE: For example, an NVIDIA 1080Ti has 28 SMs.

Full View and Restrictions

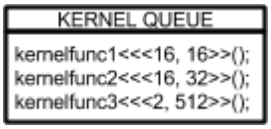
Ultimately, a block maps entirely onto one SM. Since the threads within the block are all running on the same SM, those threads can cooperate with each other to potentially solve some sub-problem (remember that each SM has some shared memory).

You as the programmer don't have to worry about SMs. The underlying driver/hardware does the work of assigning blocks to SMs. Just be aware that you have no guarantees other than…

- your block will only ever run on 1 SM
  (won't be split between SMs -- this is a driver/hardware level thing that we don't control)



- only 1 kernel runs at once
  (all blocks in a kernel finish before blocks on the next kernel run)



Other than that there are NO GUARANTEES, including...

1. there's no way to get multiple kernels to run at once
   (all blocks in a kernel finish before blocks on the next kernel run)
2. there's no way to specify which SM runs which blocks
   (you can't assign a block to an SM)
3. there's no way to specify when threads or blocks are run
   (you can't specify execution order of threads or blocks)
4. there's no guarantee that all your blocks will run at once
   (but there is a guarantee that all the threads in a single block will run at once)
5. there's no way to split your blocks between multiple SMs
   (remember that blocks can only run on a single SM, and we can't access SMs directly)
6. threads from different blocks cannot cooperate
   (only threads from the same block can communicate)

NOTE: These restrictions are what make the GPU fast. Just remember that the only guarantee you have is that all the threads in a block will run together.

Efficiency/Optimization

To write efficient programs/kernels for GPUs, we need to maximize arithmetic intensity: amount of math operations we do vs the amount of time of time we spend fetching operands from memory for those math operations.

This boils down to...

- maximizing the math we do per thread (compute ops)
- minimizing the time spent on memory per thread (fetch/store ops)

We want to optimize for spending more time working and less time on accessing memory.

> **NOTE**: Our focus here isn't necessarily to do LESS memory access, but to spend less TIME on memory access. There'll be more details on this in the memory section.

Libraries

It can be very tedious and error prone to write your own implementations of low-level parallel algorithms in raw CUDA. Just like how you wouldn't want to write your own sorting algorithm every time you need to sort something, you don't want to write your own GPU parallel primitives every time you need to do something on the GPU.

There are a number of CUDA libraries that hide many of the little details when it comes to GPU programming…

- thrust → C++ STL for CUDA, it lets you do operations like scan, reduce, and a ton of other stuff via an STL-like library. It doesn't let you run kernels directly.
- cub → cub (CUda unBound) is a library that makes it easier to write CUDA kernels. It figures out low-level particulars of how your kernel should run (e.g. how much shared memory can it use, thread block size, etc..).
- cublas → blas library that uses CUDA
- cuFFT → fft library that uses CUDA

There are many more, but these seem to be the most popular.

CUDA Kernels

The code your host (CPU) sends to your device (GPU) to run is called a kernel. Kernels are coded as if they're serial programs -- they do not explicitly define anything related to parallelism. The device (GPU) will launch multiple instances of the kernel to work on pieces of the data.

Remember that a thread runs 1 copy of your kernel: threads are organized in blocks, and blocks are part of a grid.

Example: CPU vs GPU

For example, imagine that you wanted to run some code to go over an array of floats and multiply power each by 2 (output = input^2).

Here's how that would look if you were to write it serially for the CPU…

```
for (i=0; i<64; i++) {
    out[i] = in[i] * in[i];
}
```

Here's how that would look if you were to write it for CUDA…

```
//
// Device-side (GPU) code. This is the kernel.
//   __global__ declspec lets the compiler know that this is for the GPU
//
__global__ void square(float *d_out, float *d_in) {
    int idx = threadIdx.x;
    float f = d_in[idx];
    d_out[idx] = f * f;
}

//
// Host-side (CPU) code. This launches the kernel.
//
void launchKernel() {
    // allocate data on host (CPU)
    float h_in[64] = { /* initial values */ }
    float h_out[64];

    // 1. allocate data on device (GPU)
    float * d_in;
    float * d_out;
    cudaMalloc((void **) &d_in, 64*sizeof(float))
    cudaMalloc((void **) &d_out, 64*sizeof(float))
```

```
    // 2. copy inputs from host (CPU) to device (GPU)
    cudaMemcpy(d_in, h_in, 64*sizeof(float), cudaMemcpyHostToDevice);

    // 3. LAUNCH 1 BLOCK OF 64 THREADS + WAIT FOR THEM TO FINISH
    square<<<1, 64>>>(d_out, d_in);

    // 4. copy outputs from device (GPU) to host (CPU)
    cudaMemcpy(h_out, d_out, 64*sizeof(float), cudaMemcpyDeviceToHost);

    // 5. free data on device (GPU)
    cudaFree(d_in);
    cudaFree(d_out);
}
```

Notice that vars that…

- point to memory on the device (GPU) are prefixed with d_
- point to memory on the host (CPU) are prefixed with h_

This is an important visual identifier. You can't access memory on the device (GPU) from the host (CPU) or vice versa.

Notice what's happening host-side (CPU)...

1. it allocates memory on the device
2. copies input data to the device
3. runs the kernel (1 block of 64 threads)
4. copies output data from the device
5. frees the memory that was allocated on the device

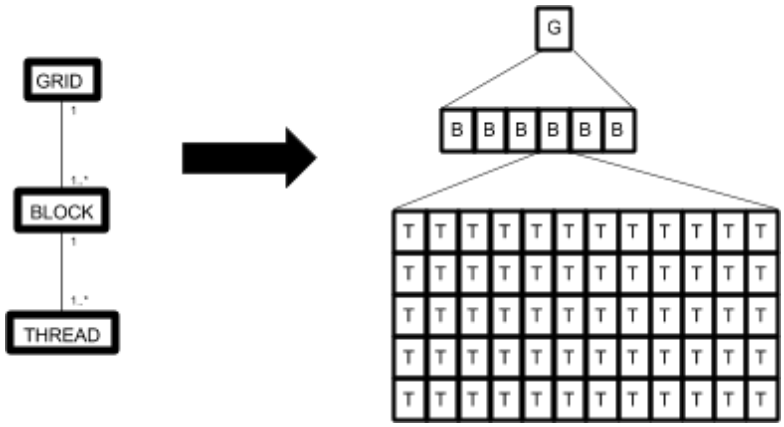
Notice what's happening for device-side (GPU)…

1. code is written as if it's single threaded and intended to work on a single element
2. __global__ declspec is used to tell the compiler that this is device code
3. threadIdx is a global var that specifies which thread the copy of the kernel is on
4. the params match the args passed in

There's some extra overhead here (in terms of code we have to write) when compared to the single-threaded CPU equivalent. There isn't much that we can do about this.

Dimensionality

It turns out that a lot of what CUDA is used for is image processing (2D) and real-world simulations (3D). As such, you can specify your blocks and/or threads as 1D, 2D, or 3D. In the graphic below, our threads are defined as a 2D block of 12x5 and our grid is a 1D block of 6...



The importance of this will be explained in further sections.

Launching

If you looked at the example, you saw that we launched 1 block of 64 threads. The syntax for launching a kernel is…

```
func<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK, SHAREDMEM_PER_BLOCK>>>(...);
```

This is exactly like calling a function, except that you specify your launch parameters between the <<< and >>>.

Here's an example…

```
square<<<1, 64>>>(d_out, d_in);
```

> **NOTE**: SHAREDMEM_PER_BLOCK wasn't specified in the example. It defaults to 0 if left out. This parameter will be discussed further in the memory section of the document.

Kernels launch asynchronously. That means that it won't block until the kernel finishes executing -- control will very likely be returned to you before the kernel finishes. If you want to block until the kernel finishes, you can call cudaDeviceSynchronize().

Remember that only 1 kernel can execute at once. If you launch multiple kernels back-to-back, they'll be put into a queue.

> **NOTE**: Typically, kernels are launched from the host. But, newer versions of CUDA will allow you to launch a kernel from inside a kernel, where that parent kernel waits for the child kernels to finish. Look up dynamic parallelism.

Defining Threads and Blocks

It turns out that a lot of what CUDA is used for is image processing (2D) and real-world simulations (3D). As such, it has native support for specifying things in 2D and 3D.

If you looked at the threading example, you saw that there was a threadIdx global var that we used to determine which item to work on. We used the x member of that var, but that var is a struct with 3 members: x, y, and z.

When we launch our kernel, we can actually pass in dim3 structures for the NUM_OF_BLOCKS and NUM_OF_THREADS inputs. For example…

```
func<<<dim3(2, 2, 1), dim3(128, 128, 1)>>>(arg1, arg2, ...);
```

Then, in our kernel, we can go ahead and make use of the y and z members of threadIdx.

> **NOTE**: There are other important global variables that need to be used to determine which part of the data you're suppose to work on. See the Identifying Threads section.

If we use ints instead of dim3s, it's effectively the same as specifying 1 for the y and z parameters…

```
// The following 2 lines are equivalent
func<<<1, 64>>>(arg1, arg2, ...);
func<<<dim3(1, 1, 1), dim3(64, 1, 1)>>>(arg1, arg2, ...);
```

Thread Limits

A device (GPU) can run many blocks at once, but there's a maximum number of threads per block. For older GPUs, this is 512. For newer GPUs, this is 1024 (possibly more? The lessons this is coming from are old).

Pick the combination of blocks and threads-per-block that makes the most sense for you. For example, if you wanted to run 1280 threads, you can try doing func<<<10,128>> (10 blocks of 128 threads-per-block). What you can't do is func<<<1,1280>> -- this is too many threads-per-block.

> **NOTE**: Does the threads-per-block value have to be a power of 2? Tests show that it doesn't. But, it looks like in the example codes provided, the total size of the array

passed into the kernel as an arg, and a guard is in place in the kernel to make sure it doesn't process out of bounds items. It may be that this is needed in certain cases.

Identifying

Remember that multiple copies of your kernel run at once (each in its own threads). There are global variables that you can access in your kernel code to help identify which thread it's running in.

In the squaring example, we used the global var threadIdx to determine which part of the data we need to work on. This is all we needed because we used a block size of 1. If we used more than 1 block, we would need to make use of other global vars as well to work out which part of the data we need to work on…

- threadIdx → which thread we're running on (relative to the block we're running on)
- blockIdx → which block we're running on

- blockDim → number of threads specified for each block
- gridDim → number of blocks specified

For example, if our kernel only deals with 1 item per thread (and we're only dealing with 1D data), we can find out which item we should be working on using this simple formula…

```
unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x
```

Debugging

You can debug your CUDA kernels by calling printf() directly in the kernel. Once your kernel launch finishes, you need to call cudaDeviceSynchronize() to flush the stuff you printed out.

Obviously, there's overhead here. You should avoid calling printf() unless absolutely required.

Here's an example...

```
#include <stdio.h>
#include <stdlib.h>

__global__ void hello() {
      printf("I'm a thread %d of block %d\n", threadIdx.x, blockIdx.x);
}

int main(void) {
      // launch kernels
```

```
    hello<<<16, 1>>>();

    // force the printf()s to flush
    cudaDeviceSynchronize();

    return 0;
}
```

This is what the output looks like...

I'm a thread 0 of block 9
I'm a thread 0 of block 8
I'm a thread 0 of block 13
I'm a thread 0 of block 5
I'm a thread 0 of block 10
I'm a thread 0 of block 2
I'm a thread 0 of block 4
I'm a thread 0 of block 6
I'm a thread 0 of block 12
I'm a thread 0 of block 14
I'm a thread 0 of block 11
I'm a thread 0 of block 1
I'm a thread 0 of block 3
I'm a thread 0 of block 0
I'm a thread 0 of block 15
I'm a thread 0 of block 7

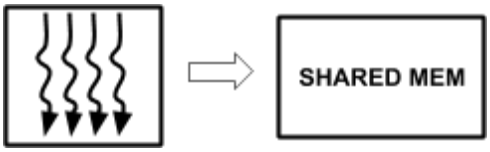
NOTE: If you're accessing shared memory from the printf call, you will need to call __syncthreads() before you do, because the writes HAVE TO COMPLETE before you can call printf()!!!! THIS IS SUPER IMPORTANT. Don't know what __syncthreads() is? Check the barrier section.

CUDA Memory

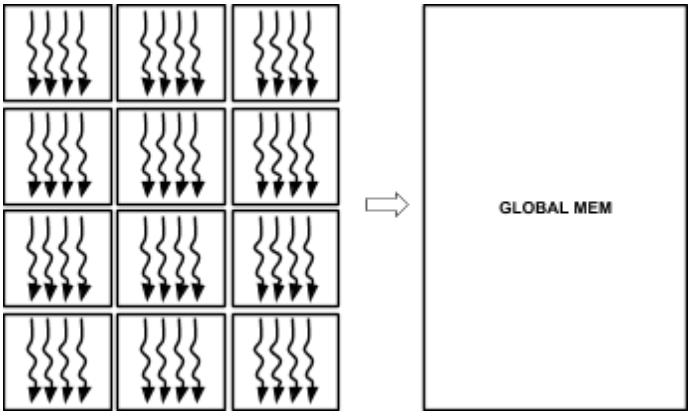
The device (GPU) provide 3 levels of memory: local, shared, and global.

- A thread has access to local memory
  (this is memory that's local to the thread -- e.g. stack memory?)

- Threads in the same block have access to shared memory
  (this is memory that can be accessed by any thread in the block)



- All threads, regardless of the block, have access to global memory
  (this is memory that can be accessed by any thread)



Essentially all this means is that…

1. threads from the same block can work together via shared memory
2. threads from different blocks can work together via global memory

The problem here is that there are multiple threads running at once, so if you're going to be reading and writing to the same location from different threads, you're going to run into thread/memory synchronization issues. Working around these will be discussed in a further section.

Obviously, local memory would be the fastest to access, followed by shared memory, followed by global memory.

Shared Memory

You can specify some amount of memory to be shared by the threads in each block. Each thread in a block will have access that shared memory.

The 2 important things to note about shared memory…

- All blocks have the same amount of shared memory
- You cannot dynamically allocate shared memory in the kernel

There are 2 ways to specify shared memory…

- As a launch parameter → use this if you need to define the amount of mem at runtime

- As an array in the kernel → use this if you know the amount of mem at compile time

> **NOTE**: Why use shared block memory? Apparently it's memory that's directly on the chip and as such is up to 100x faster to access than global GPU memory. There's also some nuance here when it comes to memory synchronization and stuff. I'll try to cover this in some other topic but for now just see this…
> https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/
>
> **NOTE**: Apparently using shared memory takes away from the L1 cache on the device (GPU). Also the amount of shared memory you can have seems to be super small (16 to 64kb?).

Specify as Launch Parameter

You can specify the amount of memory via the the 3rd launch parameter (SHAREDMEM_PER_BLOCK)...

```
func<<<NUM_BLOCKS, NUM_THREADS_PER_BLOCK, SHAREDMEM_PER_BLOCK>>>(...);
```

For example, if I wanted to have 16 floats of shared memory in each block…

```
func<<<2, 64, sizeof(float)*16>>>(arg1, arg2);
```

> **NOTE**: The value passed in doesn't have to be a literal. You can compute some value and pass that in.

Once I want to access this in my kernel, I'd go to my kernel and stick in an extern'd unsized float array with a declspec of __shared__…

```
extern __shared__ float s[];
```

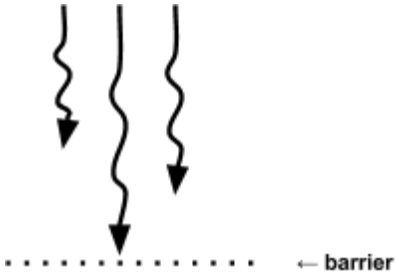
Specify in Kernel

If you know the amount of memory you want to share at compile time, you can declare the array right inside the kernel instead of using a launch parameter. All you have to do is make sure you using a literal when declaring your array and have a declspec of __shared__ on it...

```
__shared__ int s[64];
```

Synchronization Barriers

You can synchronize your threads via a barrier. A barrier just means that all your threads will PAUSE execution at a certain point and wait for the others to finish…



To add a barrier in your kernel, you need to call __synchthreads(). Below is a trivial example of using __syncthreads() to shift the elements in an array left by one…

```
#define SIZE 512

__global__ void shift_left(int * d_arr) {
      int i = threadIdx.x;

      // create shared memory space for threads in block
      __shared__ int shared[SIZE];

      // populate shared memory from global memory
      shared[i] = d_arr[i];
      __syncthreads();

      // read the element at i+1 (guarding for last)
      int temp = i == SIZE - 1 ? shared[i] : shared[i + 1];
      __syncthreads();

      // write that element to i
      shared[i] = temp;
      __syncthreads();

      // populate global memory from shared memory
      d_arr[i] = shared[i];
}

int main(void)
{
      // ... removed ...

      // launch kernels
      shift_left<<<1, SIZE>>>(d_arr);

      // ... removed ...
```

```
}
```

The important thing with the above example is that 1 thread is assigned/isolated to 1 element in the array. So each thread…

1. copies the element it's responsible from global mem to shared mem
2. waits for other threads to catch up (all elements should be in shared mem at this point)
3. reads in the neighbouring element from shared mem
4. waits for other threads to catch up (all elements should be read at this point)
5. writes the neighbour to the element it's responsible for
6. waits for other threads to catch up (all elements should be written at this point)
7. copies the element it's responsible for back from shared mem to global mem

NOTE: This was just a trivial example to illustrate barriers.

NOTE: Calling printf on shared memory? You need a barrier before the printf.

Global Memory

Global memory is memory that's shared between all threads on the device (GPU). Any thread can access it at any time. Just like how you can allocate/free memory on the host (CPU), you can allocate/free global memory on the device (GPU).

The thing to be aware of is that you cannot allocate global memory directly from the kernel. The allocating/freeing of global memory on the device (GPU) must happen on the host (CPU).

Memory Management Functions

The memory management functions for the device (GPU) are very similar to the host (CPU)...

- malloc → cudaMalloc
- memcpy → cudaMemcpy
- memset → cudaMemset
- free → cudaFree

Remember that you must allocate device global memory on the host. You cannot allocate memory directly in the kernel thread. This means that cudaMalloc, cudaMemcpy, etc.. can only ever be called from the host.

For memory that sits on the host, the convention is to name the variable with a h_ prefix.
For memory that sits on the device, the convention is to name the variable with d_ prefix.

NOTE: For the full list of memory management functions, see http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html#axzz4rHUZY4cw

Here's an example of the memory management functions being used...

```
// THIS IS ALL HAPPENING ON THE HOST
// YOU CANNOT ALLOCATE/FREE GLOBAL MEMORY IN KERNEL, ONLY IN HOST

// allocate data on host (CPU)
float h_in[64] = { /* initial values */ }
float h_out[64];

// 1. allocate data on device (GPU)
float * d_in;
float * d_out;
cudaMalloc((void **) &d_in, 64*sizeof(float))
cudaMalloc((void **) &d_out, 64*sizeof(float))

// 2. copy inputs from host (CPU) to device (GPU)
cudaMemcpy(d_in, h_in, 64*sizeof(float), cudaMemcpyHostToDevice);

// 3. LAUNCH KERNEL
square<<<1, 64>>>(d_out, d_in);

// 4. copy outputs from device (GPU) to host (CPU)
cudaMemcpy(h_out, d_out, 64*sizeof(float), cudaMemcpyDeviceToHost);

// 5. free data on device (GPU)
cudaFree(d_in);
cudaFree(d_out);
```

Note what's going on in the example…

- global device memory is being allocated on the host
- global device memory is being free'd on the host
- pointers to the allocated global device memory are being passed in as args to the kernel
- memory is being copied from host to device (can't device memory directly from host)
- memory is being copied from device to host (can't host memory directly from device)
- memory on host is prefixed with h_ (this is just a style convention)
- memory on device is prefixed with d_ (this is just a style convention)

Synchronization Barriers

There's no way to synchronize access to global memory like you do with shared memory.

The problem is that your only guarantee is that threads within the same block will be started/running. As such, those are the only threads you can synchronize via barriers.

If you have multiple blocks that have threads reading/writing to the same location in global memory, there's no guarantee that those blocks will all run at the same time. Some blocks may have to finish before others can run (we have no control over which blocks run or how many run at once).

We do have access to some synchronization functionality for global memory via atomics (explained in the next section).

Atomics

Atomics (short for atomic memory operations) are special instructions that the device implements for the purpose of fetching/storing memory atomically. Some examples include…

- Arithmetic ops → atomicAdd, atomicSub, atomicInc, atomicDec
- Min/max ops → atomicMin, atomicMax
- Bitwise ops → atomicAnd, atomicOr, atomicXor
- Swap ops → atomicCAS, atomicExch

  NOTE: Many of these atomic functions are for integers types only. Apparently one of the issues is that there's no ordering constraints when these atomic operations are being applied. As such, you can't get accurate computations for floats.

  The math property of associativity doesn't apply for floating point numbers… for example (a+b)+c != a+(b+c) -- plug into any c compiler where a=1, b=10^{10}, c=-10^{10}. If atomics were available for floats, you'd get different results for each run depending on the order in which your data was computed.

If you need to implement an atomic operation that isn't available (or for a type that isn't available), you can do so with atomicCAS, but it will be really awful. For example, here's atomicAdd written using atomicCAS…

```
__device__ int atomicAdd(int* address, int val) {
    int old = *address, assumed;
    do {
        assumed = old;
        old = atomicCAS(address, assumed, val + assumed);
    } while (assumed != old);
```

```
    return old;
}
```

These operations are implemented directly on the hardware. There is no special magic happening under the hood -- the hardware is pausing threads and shifting things around so that these atomic operations can happen.

It's not recommended you use atomics unless you really need to. They've been known to kill performance if you aren't careful.

For more information, see
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions

> **NOTE**: Having trouble thinking about this? Think of AtomicInteger in Java. It stops one thread from clobbering the results of another and provides access to basic instructions like add-and-get, get-and-add, etc...
>
> **NOTE**: Here's an important note from the docs that may be handy in the future…
>
> On GPU architectures with compute capability lower than 6.x, atomics operations done from the GPU are atomic only with respect to that GPU. If the GPU attempts an atomic operation to a peer GPU's memory, the operation appears as a regular read followed by a write to the peer GPU, and the two operations are not done as one single atomic operation. Similarly, atomic operations from the GPU to CPU memory will not be atomic with respect to CPU initiated atomic operations.
>
> Compute capability 6.x introduces new type of atomics which allows developers to widen or narrow the scope of an atomic operation. For example, atomicAdd_system guarantees that the instruction is atomic with respect to other CPUs and GPUs in the system. atomicAdd_block implies that the instruction is atomic only with respect atomics from other threads in the same thread block.
>
> **NOTE**: Atomics are for NVIDIA GPUs only?

CUDA Optimization

The following subsections detail optimization patterns specific to the CUDA architecture.

Tiered Memory

Remember how memory is broken up into a hierarchy: local, shared, and global. The way this is usually broken down…

- Local → arguments and stack variables, sits on registers or L1 cache
- Shared → shared between threads on a block, sits on SM that block is assigned to
- Global → shared between all threads, sits on GPU card itself

The general rule is, as far as speed goes: Local > Shared > Global… Local will be faster than shared, and shared will be faster than global. As such, you should move/cache more frequently-accessed data to faster memory.

> **NOTE**: This is general rule… but memory access can actually be a bit more nuanced.
>
> **NOTE**: Shared memory is like a CPU's cache block, except that you have explicit control of it.
>
> **NOTE**: Even though global memory is the slowest of the 3, it's still much faster than host (CPU) memory.

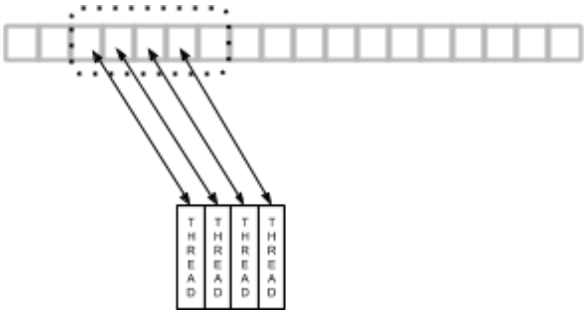
Coalesced Global Memory Access

It's much more efficient for all threads in a block to access global memory locations that are close together.

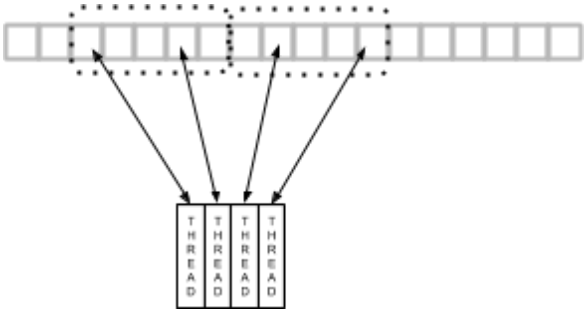
This basically has to do with memory caching. When a thread fetches (or stores) data in global memory, the GPU accesses that memory in large chunks. If other threads also try to fetch global memory data that's near the same location, those fetches will likely fall into that same large chunk. As such, the GPU won't need to pull down that same large chunk because it'll already be there.

Typical access pattern…

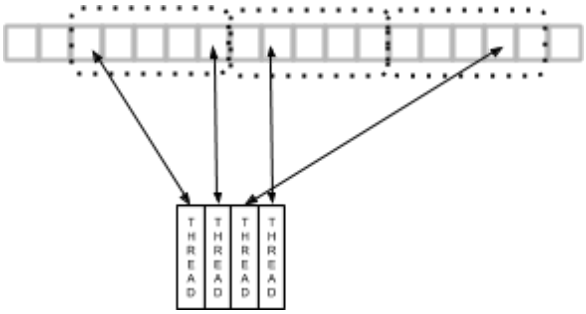
- contiguous ← this is good because mem being accessed is neighbouring each other



- strided ← this is bad (if too large) because a bunch of mem is skipped for each access



- random ← this is bad because it's just randomly grabbing data



NOTE: You can optimize the strided pattern to be coalesced. Check out Data Layout Transformation section below.

NOTE: All of this just means that the GPU is most efficient when the threads read from and write to contiguous memory locations at the same time.

Thread Occupancy

Occupancy refers to the number of threads that are actually running vs the number of threads that could be running. Each SM has properties that may potentially limit the number of threads it can run (e.g. number of cores, amount of memory, number of registers, etc..).

For example, shared memory for a thread block sits on the SM. If the SM has 48kb of shared memory space available and your thread blocks are for 24kb of shared memory each, you're going to be limited to 2 blocks per SM.

Another example is the number of threads per thread block. If the SM has 1568 cores but your thread block requires 1024 threads per SM, each SM will only ever run 1 block. In this case we can say that we have a occupancy rate of 66% -- the SMs will never use more than 1024 of their 1568 threads.

> **NOTE**: Look up CUDA occupancy calculator -- it's a spreadsheet/tool that will tell take in the specs of your thread block and return back how that block will perform on various CUDA GPUs.

Thread Divergence

The GPU likes it when all your threads go through the same execution path. If threads end up taking a different execution path, the threads that finish first will end up waiting for the longer executing threads to catch up. This may become a problem if your threads have lots of loops and/or control structures (e.g. if/elseif/else, switch, for, while, do/while, etc..).

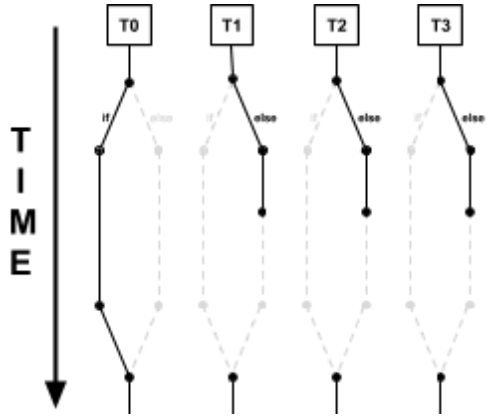
> **NOTE**: It almost sounds like a block/SM/whatever may give the user a view that multiple threads are executing, but internally it only executes 1 thread -- 1 instruction at a time, but that instruction takes in a ton of different operands (supplied by each "thread"/kernel instance). But, if this were the case, why would we need __synchthreads()?
>
> The answer to this is explained in lesson 5. The SMs in CUDA GPUs are made up of "warp cores." These "warp cores" are an NVIDIA extension to the concept of SIMD (single instruction, multiple datasets) called SIMT (single instruction, multiple threads). It looks like SIMT is essentially the same thing as SIMD. It operates on 32 operands (max of 32 threads per warp core), but you can turn on/off which operands are being worked on.
>
> For example, if you reach a if/else statement, the warp core will execute both branches -- it'll first execute the if block (only affecting the data for the threads that should be taking the if branch), then execute the else (only affecting the data for the threads that should be taking the if branch).

For example, imagine if you had the following if/else block in your kernel…

```
// ... code before ...

if (threadIdx.x == 0) {
   // takes a long time to finish
} else {
   // takes a short time to finish
}

// ... code after ...
```

The block above looks like it will take a long time to execute only for thread 0, but this isn't the case. All threads will wait until thread 0 finishes before they continue executing. It doesn't matter

that the vast majority of your threads finish the if/else quickly -- they will all pause until the longest running thread finishes.



For example, imagine if you have had the following loop block in your kernel…

```
// ... code before ...

for (int i = 0; i < threadIdx.x * 1000; i++) {
    val = val * val;
}

// ... code after ...
```

The same thing applies for loops. The block above looks like it'll take progressively longer to execute as the threadIdx goes up, but that isn't the case. Threads that finish executing the loop will just pause until the longest running thread is finishes executing the loop.

The take away here is to avoid large switches, nested if/elses, and looping by non-constants.

Thread Synchronization and Memory Access

One important fact about thread blocks is that it isn't always a good thing to have more threads per block. In certain cases, having lots of threads in a thread block where __syncthreads() calls are used will cause those threads to run slower. This is because many of the threads will be waiting on memory access operations to complete.

This is more likely to happen if the code has poor memory access patterns (e.g. large strides or scattered/random access).

Fast Math

Certain common operations from the C/C++ math library (math.h) may take a long time to execute on a GPU. CUDA provides optimized math functions via built-ins/intrinsics. For example, instead of…

- sin(), use __sin()
- cos(), use __cos()
- exp(), use__exp()

The downside to these built-in/intrinsics math functions is that you may lose a few bits of precision (which should be okay most of the time). You can find more on intrinsics by going to http://docs.nvidia.com/cuda/cuda-math-api/index.html.

In addition to that, you can get better execution times by using single-precision floating point vs double-precision floating point. Avoid double-precision unless you absolutely need it (this is common knowledge).

Memory Copy (Host ⇆ Device)

CUDA reserves a page-locked (pinned) piece of memory on the host as a staging area for transferring data to the GPU. When you copy data from the host to the GPU, what's actually happening is that memory is first being copied over to that staging area, then being sent from that staging area to the GPU.

You can avoid the staging area entirely by page-locking the part of host memory that you're working with. If you…

- already have host memory allocated, you can page-lock it via cudaHostRegister().
- want to allocate host memory that's page-locked, you can use cudaHostMalloc().

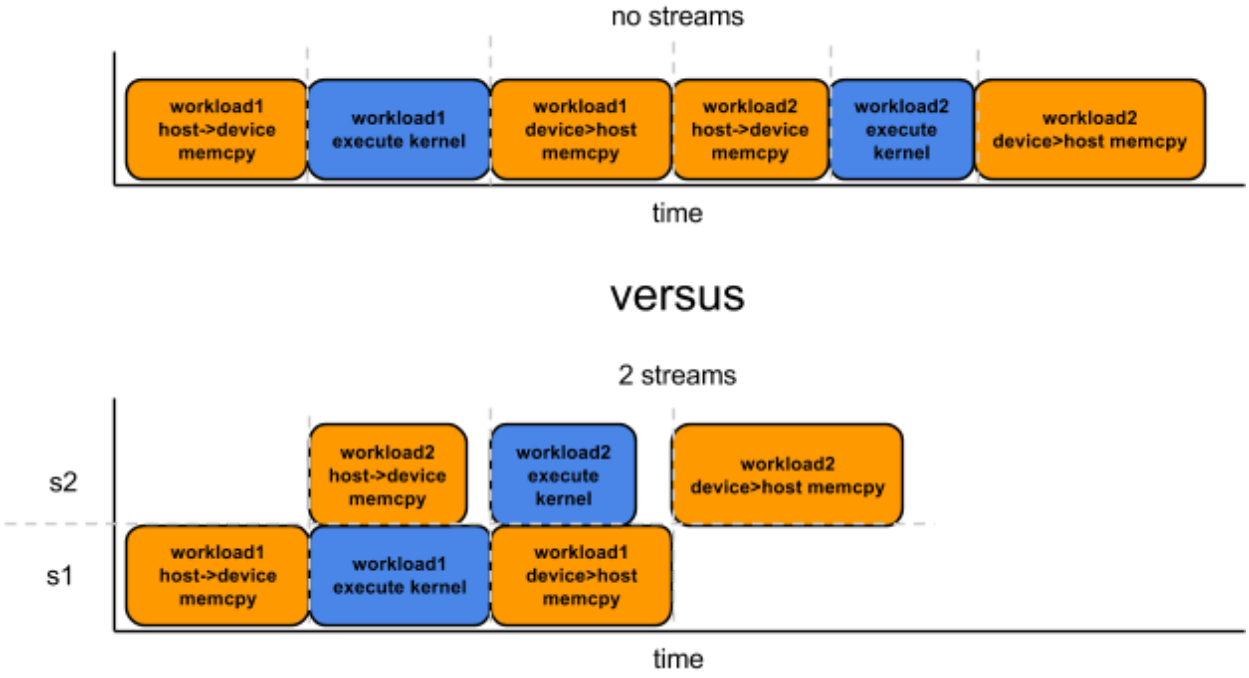
In addition to skipping the staging area, if you page-locked your host memory you can make use of cudaMemCpyAsync() to transfer the memory asynchronously -- it won't be a blocking operation, you can do other stuff while the memory transfer completes.

Streams

A stream is a sequence of CUDA operations that will execute in-order. You can have multiple streams in your program... the benefit being that the operations in each stream will execute as GPU capacity becomes available, potentially executing more than one operation at once (higher utilization).

For example, imagine you had 2 workloads. Each workload copies some data to the GPU, runs a kernel, and then copies data back. If this was implemented normally, the operations would

execute serially. If implemented using streams, the GPU takes care of when each operation is performed, potentially even running multiple operations at once...



In the above diagram, the GPU is copying over data for the 2nd workload while the kernel for the 1st workload is running, then copying over the results of the 1st workload while the kernel for the runs. These are the types of optimizations you can expect from using streams…

- not enough memory bandwidth utilized? copy data over for another stream.
- not enough SMs in use? run another stream's kernel.
- etc...

NOTE: The diagram above is a common way of getting better GPU utilization when you need to do computations on a huge amount of data (so big it won't fit into GPU memory). Instead of allocating all of the GPU's memory and running the kernel, allocate it in halves then run the same kernel in 2 separate streams. While the 1st stream is running the 2nd stream will be moving data (and vice versa). You can try maybe doing this with more than 2 streams as well?

Some things to note about using streams…

1. if using streams, operations get queued up and run asynchronously.
2. if using streams, you must pin host memory (see memory copy section above).
3. if using streams, you must use cudaMemCpyAsync (see memory copy section above).

Check out http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams for more information on streams. Below is a small example that show how to use stream with the CUDA C++ compiler.

```
cudaStream_t stream1, stream2, stream3, stream4 ;
cudaStreamCreate ( &stream1) ;
// ...
cudaMalloc ( &dev1, size ) ;
cudaMallocHost ( &host1, size ) ; // pinned memory required on host
// ...
cudaMemcpyAsync ( dev1, host1, size, H2D, stream1 ) ;
kernel2 <<< grid, block, 0, stream2 >>> ( ..., dev2, ... ) ;
kernel3 <<< grid, block, 0, stream3 >>> ( ..., dev3, ... ) ;
cudaMemcpyAsync ( host4, dev4, size, D2H, stream4 ) ;
some_CPU_method ();
```

> **NOTE**: Need to block until all your streams are done? Check out cudaDeviceSynchronize(). Need to block until a single stream is done? Check out cudaStreamSynchronize(). There are also APIs available for waiting on / firing off event: cudaEventRecord(), cudaEventSynchronize(), cudaStreamWaitEvent(), and cudaEventQuery(). You can use these event APIs directly within the kernel if you want -- look up dynamic parallelism.

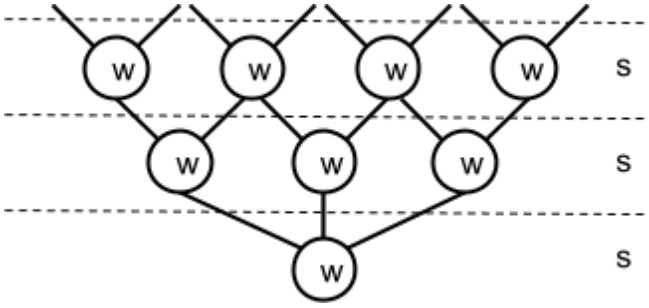
Algorithm Complexity

Work Complexity vs Step Complexity

Parallel algorithms on the GPU are calculated as step complexity and work complexity...

- step complexity → complexity of algorithm if run parallelly
- work complexity → complexity of algorithm if run serially/sequentially

Imagine the following chain of work...



We can say the example has a step complexity of 3 and a work complexity of 8. Each level/depth is a step, but each node is a piece of work. Work nodes at each step are generating some output, and the work nodes in the following steps are that output for their input.

In the real world, this is effectively just like big O notation. Instead of saying that the work/step complexity is some specific number, we refer to as a function of the size of the input. For example…

- the work efficiency is proportional to sizeof(input) / O(n)
- the work efficiency is proportional to sizeof(input)^2 / O(n^2)
- the step efficiency is proportional to log(sizeof(input)) / O(log(n)
- etc..

Work Efficiency

We can say our parallel algorithm is work efficient if it's asymptotically the same -- within a constant factor as the work complexity of its non-parallel equivalent.

All this means is that your parallel algorithm's complexity isn't more than the complexity of the serial equivalent: if in the serial version you're looping over your array once for each element, in the parallel version you don't want to be looping over the entire array once for each element: O(n) vs O(n^2). This is assuming you're doing more-or-less the same thing in the loop.

You can get a good better idea of work efficency means by reading this page https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html (sections 39.1 and 39.1.1). I've pasted the important most important part below…

> A simple and common parallel algorithm building block is the all-prefix-sums operation. In this chapter, we define and illustrate the operation, and we discuss in detail its efficient implementation using NVIDIA CUDA. Blelloch (1990) describes all-prefix-sums as a good example of a computation that seems inherently sequential, but for which there is an efficient parallel algorithm. He defines the all-prefix-sums operation as follows:
>
> > The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements
> > $[a_0, a_1, ..., a_{n-1}]$
> > and returns the array
> > $[I, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1 \oplus ... \oplus a_{n-2})]$
>
> For example, if $\oplus$ is addition, then the all-prefix-sums operation on the array
> [3 1 7 0 4 1 6 3]
> would return
> [0 3 4 11 11 15 16 22].
>
> The all-prefix-sums operation on an array of data is commonly known as *scan*. We use this simpler terminology (which comes from the APL programming language [Iverson 1962]) for the remainder of this chapter. The scan just defined is an *exclusive* scan, because each element *j* of the result is the sum of all elements up to but *not including j* in the input array. In an *inclusive* scan, all elements *including j* are summed. An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity. Likewise, an

inclusive scan can be generated from an exclusive scan by shifting the resulting array left and inserting at the end the sum of the last element of the scan and the last element of the input array (Blelloch 1990). For the remainder of this chapter, we focus on the implementation of exclusive scan and refer to it simply as "scan" unless otherwise specified.
There are many uses for scan, including, but not limited to, sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, and building histograms and data structures (graphs, trees, and so on) in parallel. For example applications, we refer the reader to the survey by Blelloch (1990). In this chapter, we cover summed-area tables (used for variable-width image filtering), stream compaction, and radix sort.

In general, all-prefix-sums can be used to convert certain sequential computations into equivalent, but parallel, computations…

```
// sequential version
out[0] = 0;
for j from 1 to n do
  out[j] = out[j-1] + f(in[j-1]);

// parallel version
forall j in parallel do
  temp[j] = f(in[j]);
all_prefix_sums(out, temp);
```

Implementing a sequential version of scan (that could be run in a single thread on a CPU, for example) is trivial. We simply loop over all the elements in the input array and add the value of the previous element of the input array to the sum computed for the previous element of the output array, and write the sum to the current element of the output array.

```
out[0] := 0
for k := 1 to n do
  out[k] := in[k-1] + out[k-1]
```

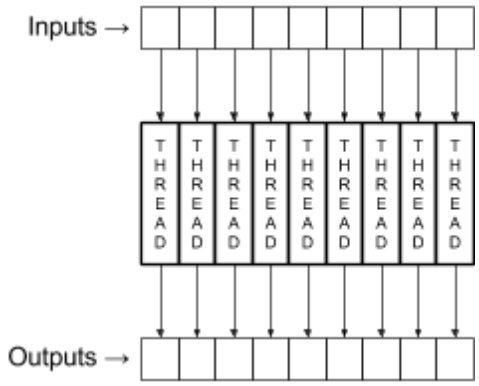
This code performs exactly *n* adds for an array of length *n*; this is the minimum number of adds required to produce the scanned array. When we develop our parallel version of scan, we would like it to be *work-efficient*. A parallel computation is work-efficient if it does asymptotically no more work (add operations, in this case) than the sequential version. In other words the two implementations should have the same *work complexity*, $O(n)$.

Algorithm Primitives

The following subsections define a common set of primitives for highly parallel algorithms.

Map

A map operation takes each data element (e.g. elements of an array, entries in a matrix, pixels in an image, etc…) and performs the same computational task on that element to produce a result. There's a one-to-one correspondence between input and output.



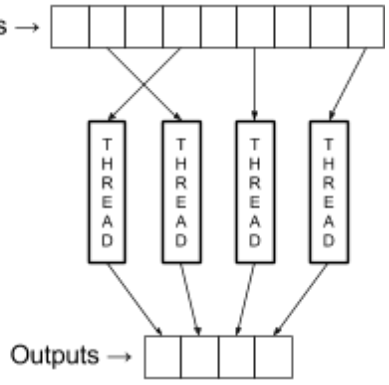
Potential use-cases...

1. transforming a number via some equation
2. turning an image to grayscale
3. blurring an image
4. transposing an 2d matrix
5. breaking up a struct → array of structures (AOS) to a structure of arrays (SOA)

Special Cases

Gather

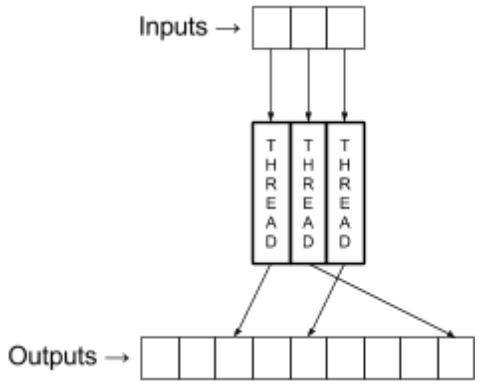
A gather operation is a map, but there's only a one-to-one relationship from output to input. Every output location has a corresponding input location, but not every input location has a corresponding output location.



> **NOTE**: See the section on "Coalesced Global Memory Access" to see why this can be a bad pattern.

Scatter

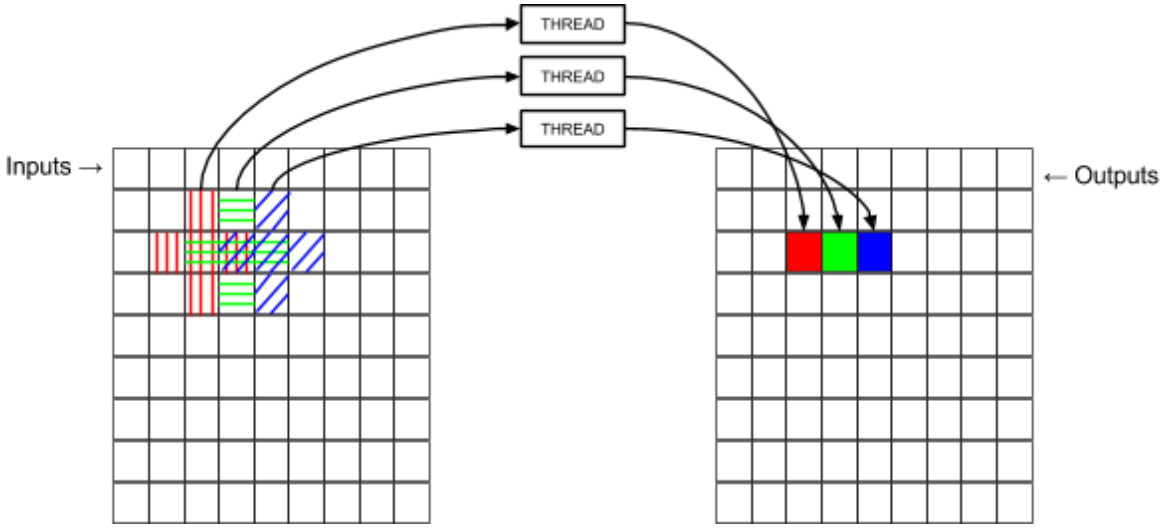
A gather operation is a map, but there's only a one-to-one relationship from input to output. Every input location has a corresponding output location, but not every output location has a corresponding input location.



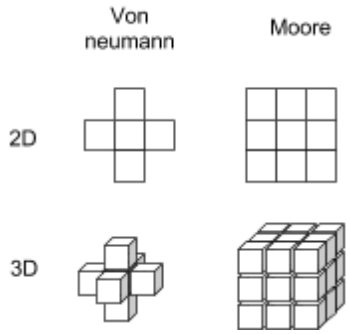
> **NOTE**: The output location isn't guaranteed to be unique. If there's a possibility for multiple threads to write to the same location (remember that all threads run at the same time), you're going to run into problems. It hasn't been clearly explained how to handle this yet.
>
> **NOTE**: See the section on "Coalesced Global Memory Access" to see why this can be a bad pattern if not implemented correctly.
>
> **NOTE**: One of the use-cases described for this was applying an image. How exactly can this be used to blur an image? We can take the input pixel and add a 1/3rd of it to each output.

Stencil

A stencil operation is where the thread reads input from a fixed neighbourhood within in array and produces some output. This is useful for things like image processing.

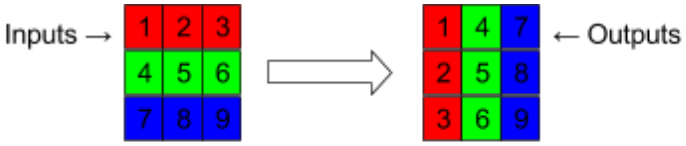


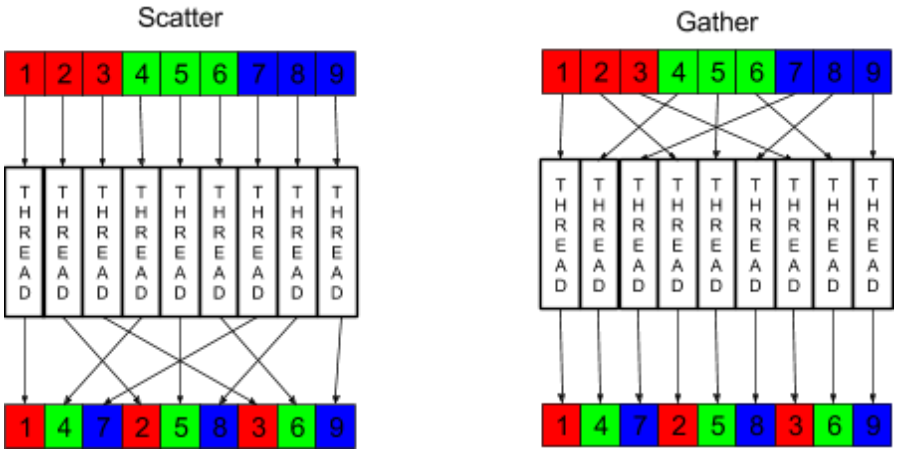
Common shapes for stencils include…



Transpose

A transpose operation is a plain old transpose operation such as ones you would do with a matrix. It can be considered a special case of either gather or scatter.





Transpose operations can also be used for reordering or breaking up a structure -- converting an array of structures (AOS) to a structure of arrays (SOA). Imagine the following struct...

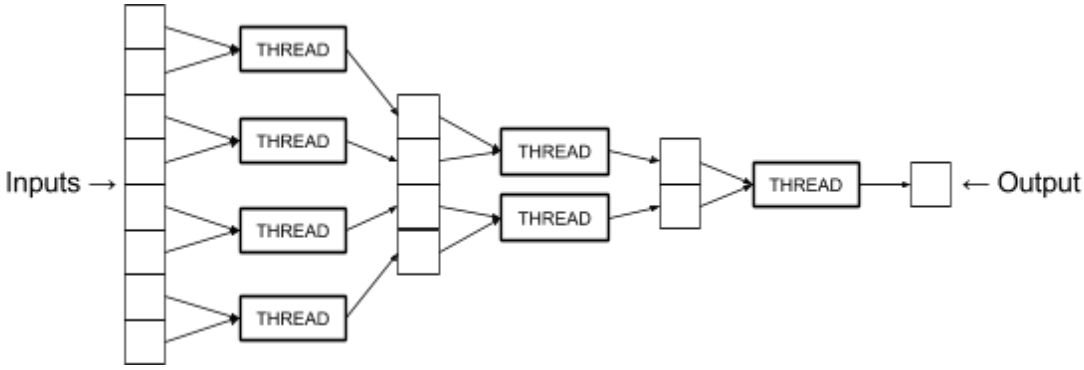
```
struct my_struct {
    float f;
    int i;
}
```

We can convert my_struct[1000] to float[1000] + int[1000]. It's useful to break up structs like this if we're doing heavy computations that mostly access certain bits (e.g. the floats). This is called the principle of locality.

> **NOTE**: We can also consider reordering the members of a struct as a transpose operation (reordering vs purely breaking them apart).

Reduce

A reduce operation is where threads aggregate a bunch of data together. This is done in multiple steps and leads to a single output.



The operator you use in your reduce must follow these rules...

- binary ← takes in 2 inputs
  The binary property essentially halves the data at each step. The operator takes in 2 inputs but gets back 1 output, so you can keep running it iteratively until you end up with your final output.

- associative ← order in which operator is applied doesn't matter
  Associative means you can choose to do your operations in any order. For example, if your operator was addition and your inputs were a, b, and c, you could add them in any order and get to the same result. If a=1, b=2, c=3…
  (1+2)+3=6
  1+(2+3)=6

  Why is this needed? It guarantees that the same result is given back regardless of the order in which threads run.

  NOTE: We make the assumption that associativity implies commutativity. Commutativity means that the operands going in can be in any order. For example, a+b=b+a.

Some ops that pass the requirements…

- addition (+)
- multiplication (*)
- bitwise and
- bitwise or
- min/max

Some ops that fail the requirements:

- subtraction (-) ← not associative... (a-b)-c != a-(b-c)
- division (/) ← not associative... (a/b)/c != a/(b/c)
- exponent ← not associative... pow(pow(a,b),c) != pow(a,pow(b,c))
- factorial (!) ← factorial takes 1 input, not 2... e.g. 5!

Potential use-cases...

1. finding min or max of a large set of numbers
2. adding a bunch of numbers together

Implementations

Serial

Below is a CPU implementation of reduce. The implementation is using addition (+) as its operator...

```
int input[4] = { 10, 4, 5, 8 };
int output = 0;
for (int i = 0; i < 4; i++) {
    output += input[i];
}
```

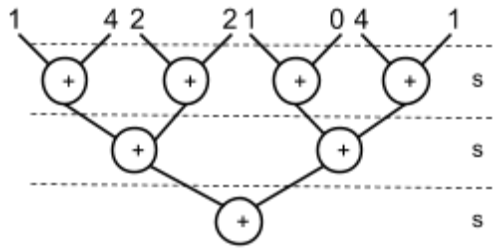
Output is…

```
27
```

Parallel

> **NOTE**: Remember the difference between work and step. Work refers to a unit of work. Step refers to multiple units of work being done in parallel.

For example, imagine you wanted to reduce 8 numbers via the addition operator. Conceptually, here's how the entire process would look…



Notice what's happening here. There are 3 steps. The work in each step is being done in parallel…

1. 4 threads will run in parallel, each will perform one of the following ops:
   1+4=5
   2+2=4
   1+0=1
   4+1=5

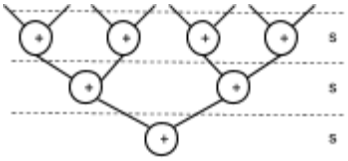
2. 2 threads will run in parallel, each will perform one of the following ops:
   5+4=9
   1+5=6
3. 1 thread will run, it will perform the following op:
   9+6=15

If you were to code this example up, it would just be 3 launches of a kernel that adds 2 numbers together…

```
__global__ void sum(float *d_in, float *d_out) {
    int idx = threadIdx.x;
    float i1 = d_in[(idx*2)];
    float i2 = d_in[(idx*2)+1];
    d_out[idx] = i1+i2;
}

void launchKernel() {
    // ...
    sum<<<1, 4>>>(d_in, d_out1);   // step1
    sum<<<1, 2>>>(d_out1, d_out2); // step2
    sum<<<1, 1>>>(d_out2, d_out3); // step3
    // ...
}
```

Essentially, this means we can reduce a large amount of inputs in a small number of steps. In our example, we summed up 8 elements in 3 steps. If we wanted to sum up 65536 elements instead, that would come out to 16 steps.

The work complexity of reduce is around O(n-1).
The step complexity of reduce is around $O(\log_2(n))$.

> **NOTE**: Having trouble reasoning out the complexity for this? Imagine we had 8 inputs and we were reducing it using the + operator…
>
> 
>
> 8 inputs (this is n)
> 7 work operations (this is n-1)
> 3 step operations (this is $\log_2(n)$)

> **NOTE**: As an optimization, you can always cluster together multiple runs of the operator in your thread. For example, instead of just adding 2 inputs together, you can just to add 10 or 20 inputs.

Practical Considerations

Number of Parallel Units

What happens if we try to reduce a massive amount of elements (e.g. a billion elements)? At each step, our GPU can only run so many threads in parallel. If we go past this cap, the GPU may internally break steps into multiple pieces that it runs sequentially. Look up “Brent’s theorem” for a deeper dive into this topic.

Floating Point Arithmetic / Non-associative Operators

What happens if you choose an operator that isn’t associative? It may still be okay to use this pattern. For example, think of floating point numbers. Arithmetic on floating point numbers is not associative. This doesn’t mean you can’t use reduce, it just means that you may not get consistent results between runs for the same inputs. This is due to the fact there is no guarantee that threads will execute in the same order. You may be processing in a different order each time.

> **NOTE**: If you’re having trouble thinking of how floating point arithmetic is not associative, try (a+b)+c != a+(b+c) -- plug into any c compiler where a=1, b=10^{10}, c=-10^{10}.

Scan

A scan operation is that transforms an array of elements such that each element becomes the aggregate of all elements preceding it.

> **NOTE**: There is no thread representation diagram here because this is a combination of other patterns and there are several implementations of it.

The operator you use for aggregation in your scan must follow these rules...

- binary ← takes in 2 inputs
- associative ← order in which operator is applied doesn’t matter
- identity ← must have some constant value I where f(I,b)=b

This rules are similar to the rules for reduce, but with the additional restriction of needing to have an identity constant. If you need more information on what binary and associativity do, see the reduce section. If you’re wondering about identity, here are the identity values of common operators…

- addition (+) → 0 e.g. 5+0=5
- multiplication (*) → 1 e.g. 5*1=5

- binary or → 0 e.g. or(1,0)=1 or(0,0)=0
- binary and → 1 e.g. and(0,1)=0 and(1,1)=1
- max → MIN_INT e.g. max(MIN_INT, 10)=10
- min → MAX_INT e.g. min(MAX_INT, 10)=10

  NOTE: MIN_INT/MAX_INT are defined by the type you're working with. For example, if you're working with an unsigned char, MIN_INT=0 and MAX_INT=0xFF.

Having trouble understanding this? Imagine that we wanted to do a scan of the array [1, 2, 3, 4] using + as the operator. Here's how we calculate each element of the output array…
[0, 1, 1+2, 1+2+3] → [0, 1, 3, 6]

For each element, we're aggregating (adding in this case) all the elements preceding it. The first element has nothing preceding it so it's set to the identity.

Potential use-cases...

1. cumulative distribution function
2. quicksort
3. etc..

Variations

There are actually 2 forms of scan: inclusive and exclusive...

- Inclusive: each element will be the aggregate of all elements before and including it.
  $[a_0, a_1,..., a_{n-1}] \rightarrow [a_0, (a_0 ? a_1),...,(a_0 ? a_1 ? ... ? a_{n-1})]$
- Exclusive: each element will be the aggregate of all elements before it.
  $[a_0, a_1,..., a_{n-1}] \rightarrow [I, a_0, (a_0 ? a_1),...,(a_0 ? a_1 ? ... ? a_{n-2})]$

Replace ? with the operator of your choice (e.g. +, *, etc..) and I with the identity for that operator. Remember that the operator must have an identity, must be binary, and must be associative+commutative.

> **NOTE**: If you wanted to be super explicit, you can write exclusive as…
> $[a_0, a_1,..., a_{n-1}] \rightarrow [I, (I ? a_0), (I ? a_0 ? a_1),...,(I ? a_0 ? a_1 ? ... ? a_{n-2})]$
> If you have trouble thinking about this, try use the operator + (identity is 0) or * (identity is 1). It should work.

For example, imagine you had the array [10, 4, 5, 8] and you wanted to scan over it using + as the operator…

- Inclusive: [10, (10+4), (10+4+5), (10+4+5), (10+4+5+8)] → [10, 14, 19, 27]
- Exclusive: [0, 10, (10+4), (10+4+5), (10+4+5)] → [0, 10, 14, 19]

> **NOTE**: Remember that the form for exclusive puts the identity for addition as the first element and doesn't take into account the last element. Remember that the identity for addition is 0: x+0=x.

What's the point of having 2 different variations of a scan? As far as I know, there are 2 different algorithms for performing a scan in parallel (discussed in later section). One of the algorithms produces an exclusive result, while the other algorithm produces an inclusive result.

It seems fairly trivial to convert one type of result to the other 2...

- I → E: shift right by 1 then add a 0 to the last element
- E → I: shift left by 1 then do a reduce over the original array to get last element

> **NOTE**: Remember that shifting a large array left/right by 1 is a super fast thing to do in parallel computing.

Implementations

Serial

Below is a CPU implementation of an inclusive scan. The implementation is using addition (+) as its operator...

```
int input[4] = { 10, 4, 5, 8 };
int output[4];
int acc = 0; // set this to the identity.. 0 for addition
for (int i = 0; i < 4; i++) {
   // flip these 2 lines to switch to exclusive scan
   acc += input[i];
   output[i] = acc;
}
```

Output is...

```
10 14 19 27
```

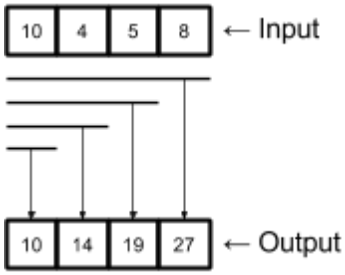
Parallel (Naive)

A naive parallel implementation for scan would be to use reduce for each element on the input array, where the reduce targets everything up to and including that element.

For example, imagine you were to perform an inclusive scan on the array [10, 4, 5, 8] using + as the operator…

- Index 0: reduce([10], +)
- Index 1: reduce([10,4], +)
- Index 2: reduce([10,4,5], +)

- Index 3: reduce([10,4,5,8], +)



Note that you can perform multiple reduces in parallel. Practically speaking, you start from index 0 of the input array and work your way up until you reach a cap for how many reduces/threads you can run. Then, you'll then have to wait for those reduces to finish before you can move on to the next group of elements. For the next group of elements from the input array, you can make use of the last element you computed in the previous group.

> **NOTE**: Having trouble visualizing this? Imagine that the 3rd element in the example was the last element we were able to do a reduce for before you ran out of GPU resources. For the next set of elements, you can directly make use of the result of that 3rd element...
> reduce([19, 8], +) vs reduce([10,4,5,8], +)

The work complexity of scan is around $O(n^2)$.
The step complexity of scan is around O(log(n))

> **NOTE**: Remember the difference between work and step. Work refers to a unit of work. Step refers to multiple units of work being done in parallel.

The step complexity for scan is the same as the step complexity for reduce because technically we're just running a whole bunch of reductions at once. The work complexity for scan is n^2 because we go over the elements of the input array once per reduce (we don't go over all elements per reduce, but we can just generalize this as n^2).

The n^2 work complexity makes this algorithm "ridiculously inefficient." As such, you should use one of the other less-trivial parallel algorithms discussed in later sections.

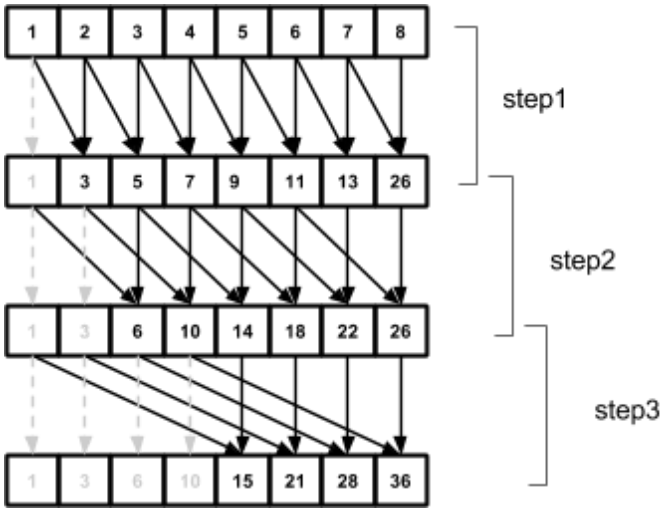
Parallel (Hillis-Steele)

Hillis-Steele is a parallel implementation of inclusive scan.

The high-level steps for Hillis-Steele scan algorithm are… at each parallel step

1. copy the first 2^{step} elements from the input array
2. add each remaining element with the element that's 2^{step} to its left

Here's a diagram showing exactly what the algorithm does for the input array [1,2,3,4,5,6,7,8]...



Note that this is much more efficient than the naive parallel implementation of scan. In the case of our example above, there are 8 elements for our scan. Using this implementation, we're doing 3 reduce-like operations. If we used our naive parallel implementation, we'd be doing 8 reduce operations.

Below is a Java implementation of the algorithm written in non-parallel form. The example takes in the array [1,2,3,4,5,6,7,8] and performs a scan on it using addition (+) as the operator...

```
int[] input = {1, 2, 3, 4, 5, 6, 7, 8};
int[] output;

int n = 1;
while (n < input.length) {
    // create new output array for this step
    output = new int[8];

    // copy the first n elements of input to output
    for (int idx = 0; idx < n; idx++) {
        output[idx] = input[idx];
    }

    // for each remaining element in input, output[i]=input[i]+input[i-n]
    for (int idx = n; idx < input.length; idx++) {
        int val1 = input[idx];
        int val2 = input[idx - n];
        output[idx] = val1 + val2;
    }
```

```
    System.out.println(Arrays.toString(output)); // print step's output

    input = output;
    n *= 2;
}
```

Output of the code is...

```
[1, 3, 5, 7, 9, 11, 13, 15]
[1, 3, 6, 10, 14, 18, 22, 26]
[1, 3, 6, 10, 15, 21, 28, 36]
```

The last line of the output is the correct result. Although the code above is non-parallel, it should be obvious what parts you can run in parallel: pretty much every iteration of the while loop can be run as a single parallel step.

> **NOTE**: Although this is an inclusive scan, it's trivial to convert it to an exclusive scan: shift the final output right by 1 and put the identity value at index 0.

The step complexity is O(log(n))
The work complexity is O(n*log(n))

> **NOTE**: Remember the difference between work and step. Work refers to a unit of work. Step refers to multiple units of work being done in parallel.

If you're having trouble visualizing this, just look at our example. We had an input size of 8 but the number of parallel steps were 3. Try it with smaller or larger step sizes, you should see a pattern like this…

- input size of 2 = 1 parallel step
- input size of 4 = 2 parallel steps
- input size of 8 = 3 parallel steps
- input size of 16 = 4 parallel steps
- etc...

You can see how the number of steps here will $\log_2(n)$ where n is the size of the input. We generalize this a log(n) when we specify the step complexity.

Parallel (Blelloch)

Blelloch is a parallel implementation of exclusive scan.

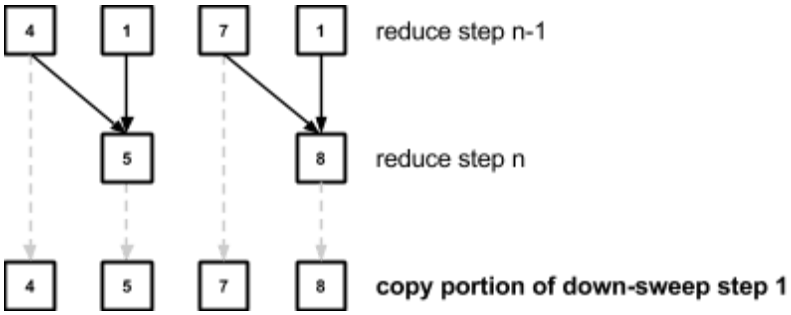
The high-level steps for Blelloch scan algorithm are…

1. perform reduce on the input array up to the point where there are only have 2 elements, saving the intermediate results at each step of the reduce
2. set the last element of the last step of the reduce to the identity
3. down-sweep the array back out to the original input size
4. perform one final down-sweep on the array without expanding it

   NOTE: Remember that the scan operator you use must have an identity value associated with it. If you don't remember what this is, read the main section of the scan section.

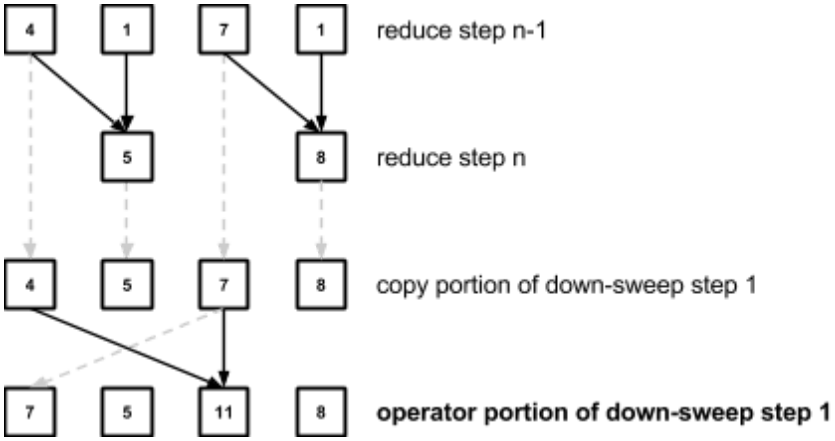
What is a down-sweep and how does it work? You can think of the initial reduce as an up-sweep. When we expand it back out to calculate the scan, we call that the down-sweep.

Remember that we saved all the intermediate values of our reduce. So when we perform a down-sweep step, we begin by doubling the size of the array and copying over the missing elements from the previous reduce step. For example….

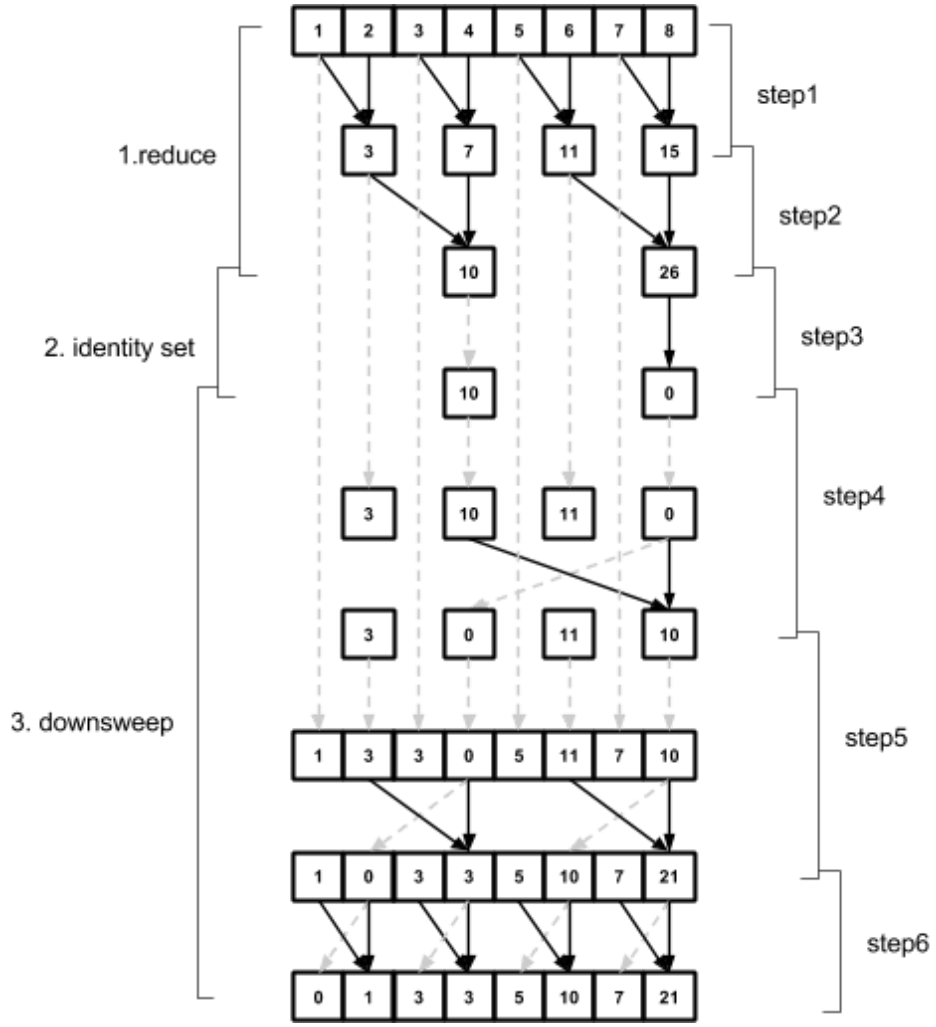


NOTE: Why do we have 2 elements at the end of our reduce? Remember that in step 1 we say that we stop our reduce when we have 2 elements.

Once that's complete, we perform a down-sweep operator on each pair of newly introduced elements. A down-sweep operator copies the right value over to the left, and sets the left value to left OP right. If we were to apply the down-sweep operator to the example above (note that we're using addition as our operator), it would look like this…



Here's a diagram showing exactly what the algorithm does for the input array [1,2,3,4,5,6,7,8] using addition (+) as the operator…



NOTE: Remember that that the identity for addition is 0.

NOTE: Look at the last step... we need to apply the down sweep operator to each pair of elements in the final array one last time before we get the result we expect for a scan.

Note that this is much more efficient than the naive parallel implementation of scan. In the case of our example above, there are 8 elements for our scan. Using this implementation, we're effectively just doing 1 reduce operation and 1 downsweep operation. That's around 5 to 6 parallel steps. If we used our naive parallel implementation, we'd be doing 8 reduce operations. In addition to that, we're doing much less work than the naive parallel implementation (the scan operator is applied way less times).

Below is a Java implementation of the algorithm written in non-parallel form. The example takes in the array [1,2,3,4,5,6,7,8] and performs a scan on it using addition (+) as the operator…

```
public static void main(String[] args) {
    int[] in = new int[] {1,2,3,4,5,6,7,8};

    System.out.println("Reducing...");
    int[][] reduce_out = reduce(in);

    System.out.println("Resetting final element to 0...");
    reduce_out[reduce_out.length-1][1] = 0;
    System.out.println(Arrays.toString(reduce_out[reduce_out.length-1]));

    System.out.println("Downsweeping...");
    downsweep(reduce_out);
}

public static int[][] reduce(int[] input) {
    List<int[]> ret = new ArrayList<>();

    ret.add(input);

    while (input.length > 2) {
        int[] output = new int[input.length / 2];

        for (int i = 0; i < output.length; i++) {
            output[i] = input[i*2] + input[i*2+1];
        }

        ret.add(output);
        input = output;
    }
```

```
    for (int i = 0; i < ret.size(); i++) {
        System.out.println(Arrays.toString(ret.get(i)));
    }

    return ret.toArray(new int[0][]);
}

public static int[] downsweep(int[][] input) {
    int[] last_output = input[input.length - 1];
    for (int i = input.length - 2; i >= 0; i--) {
        int[] reduce_res_prev = input[i];

        int[] output = Arrays.copyOf(
                reduce_res_prev,
                reduce_res_prev.length);

        // Pull down from last output
        for (int j = 0; j < last_output.length; j++) {
            output[(j*2)+1] = last_output[j];
        }

        // Sweep the values we pulled down
        for (int j = 0; j < output.length; j += 4) {
            int val1 = output[j+1];
            int val2 = output[j+3];

            output[j+1] = val2;
            output[j+3] = val1 + val2;
        }

        System.out.println(Arrays.toString(output));

        last_output = output;
    }

    // Perform one last sweep of final values
    for (int j = 0; j < last_output.length; j += 2) {
        int val1 = last_output[j];
        int val2 = last_output[j + 1];

        last_output[j] = val2;
        last_output[j + 1] = val1 + val2;
```

```
    }

    System.out.println(Arrays.toString(last_output));

    return last_output;
}
```

Output is…

```
Reducing...
[1, 2, 3, 4, 5, 6, 7, 8]
[3, 7, 11, 15]
[10, 26]
Resetting final element to 0…
[10, 0]
Downsweeping...
[3, 0, 11, 10]
[1, 0, 3, 3, 5, 10, 7, 21]
[0, 1, 3, 6, 10, 15, 21, 28]
```

The last line of the output is the correct result. Although the code above is non-parallel, it should be obvious what parts you can run in parallel.

> **NOTE**: Although this is an exclusive scan, it's trivial to convert it to an inclusive scan: shift the final results left by 1 and perform a reduce on the original input to get the value for the last element.

The step complexity is O(log(n))
The work complexity is O(n)

> **NOTE**: Remember the difference between work and step. Work refers to a unit of work. Step refers to multiple units of work being done in parallel.

Note that this is the same work and step complexity of a reduce. A downsweep is the opposite of a reduce (we're expanding out rather than reducing), but it's effectively the same amount of work with the same amount of steps. As such, we can say that, for the same amount of data, we're doing 2 reduce operations worth of steps and work. That just gets generalized down to the same work and step complexity as a reduce.

That work and step complexity may seem great, but remember that we have to store the intermediate values of the initial reduce that we do, which means that this is going to be much more memory inefficient.

Practical Considerations

Number of Parallel Units and Algorithm Choice

Given the same input array, of the two parallel implementations of scan…

- Hillis-Steele does less parallel steps but more work
- Blelloch does more parallel steps but less work

So which should you choose? It depends on the amount of work you have and the number of parallel units in your GPU. At each step, your GPU can only run so many threads in parallel. If it goes past this cap, the GPU may internally break steps into multiple pieces that it runs sequentially.

If you know that you have more than enough parallel units to handle all the work in each step, use Hillis-Steele. There will be more work being performed per parallel step (more threads per step), but there will be fewer parallel steps and each step will actually run in parallel instead of being broken up.

If you know that you don't have enough parallel units to handle all the work in each step, go for Blelloch. There will be more parallel steps, but much less work will be happening in each parallel step (less threads per step). As such, it'll be far more likely that each step will actually run in parallel instead of being broken up.

Look up "Brent's theorem" for a deeper dive into this topic.

Floating Point Arithmetic / Non-associative Operators

This is from the section on reduce, but it applies to scan as well…

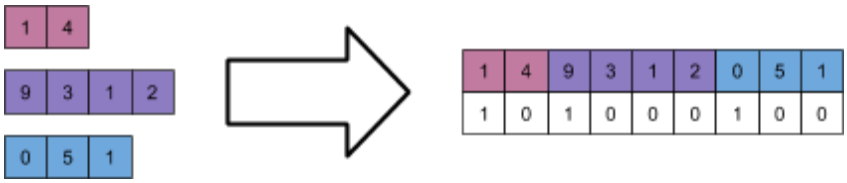
What happens if you choose an operator that isn't associative? It may still be okay to use this pattern. For example, think of floating point numbers. Arithmetic on floating point numbers is not associative. This doesn't mean you can't use scan, it just means that you may not get consistent results between runs for the same inputs. This is due to the fact there is no guarantee that threads will execute in the same order. You may be processing in a different order each time.

> **NOTE**: If you're having trouble thinking of how floating point arithmetic is not associative, try (a+b)+c != a+(b+c) -- plug into any c compiler where a=1, b=10^{10}, c=-10^{10}.

Segmented Scan

Recall that kernels run one at a time, back-to-back. In a lot of cases you'll be doing scans for many small arrays instead of a few large arrays. In those cases, you can try to implement a technique known as segment scan.

Segmented scan involves concatenating your input arrays together and then having a secondary array that identifies segment heads (where each individual array starts in the concatenated array). The kernel will use this information to scan all the arrays at once...



> **NOTE**: The assumption here seems to be that Hillis-Steele and Blelloch can be tweaked to handle this? I still need to work out how to do this.

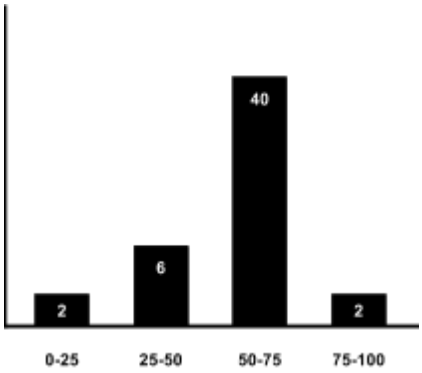
Algorithm Patterns

The following subsections discusses how common algorithms can be implemented in parallel.

Histogram

A histogram is where, for each input, you dump that input into a bucket based on the range that it sits in.

For example, let's say you have the test scores for 50 students. You want to split those scores up into 4 buckets to see how well/poorly the students did overall: 0-25, 25-50, 50-75, 75-100. For each test score, find out what bucket that test score is in and add 1 to it...



Serial

Python example...

```
input_data = [15,11,2,25,4,5,6,7,10,49,1,3,4]
histo = [0, 0, 0, 0, 0]
for e in input_data:
```

```
    idx = e // 10
    histo[idx] += 1

print(str(histo))
```

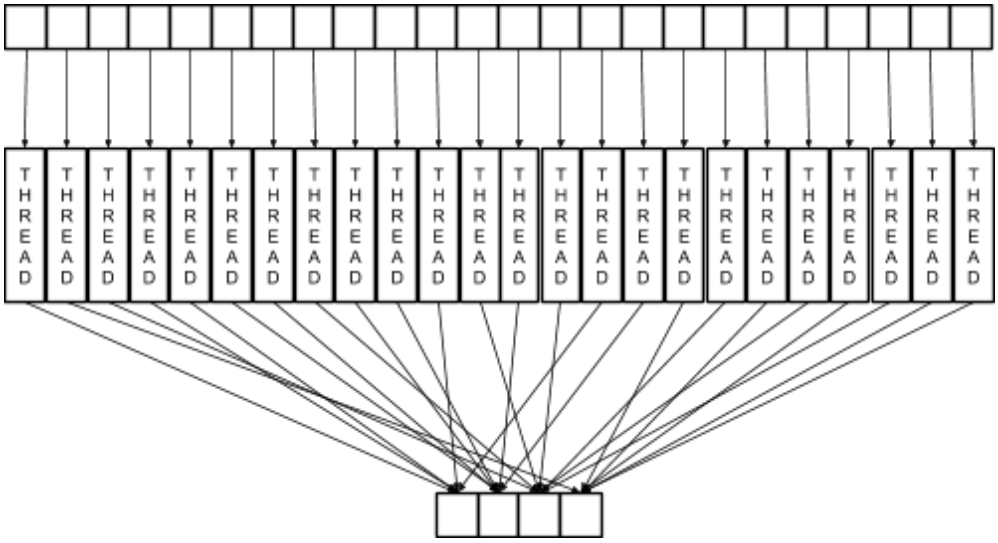
Output is…

```
[8, 3, 1, 0, 1]
```

Parallel (Atomic Add)

Launch a thread for each element and have each thread…

1. calculate which bucket the input is in
2. perform an atomicAdd on to that bucket



This is a really bad idea if you have a low number of buckets. Because many threads will be trying to add to that same bucket at the same time, threads adding to the same bucket will have to queue up and perform the add sequentially. This kills the whole point of having many parallel threads.

Python example…

```
import numpy as np
from numba import cuda


@cuda.jit
def hist(in_data, out_data):
```

```
    pos = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    val = in_data[pos]
    bucket = val // 10

    cuda.atomic.add(out_data, bucket, 1)


in_data = np.random.randint(0, 50, 10000)
out_data = np.zeros(5, dtype=np.int32)
hist[10, 1000](in_data, out_data)

print(str(out_data))
```

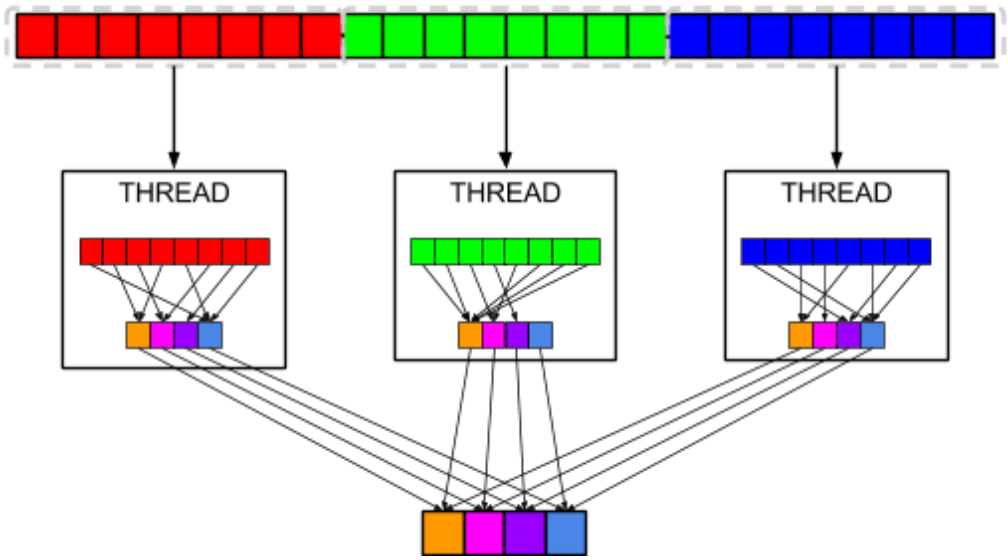
Output is…

```
[2036 1993 1928 2001 2042]
```

Parallel (Thread Local Histogram and Atomic Add)

Launch a thread for each n elements and have each thread…

1. calculate the histogram locally just for those n elements (sequentially)
2. for each bucket in the local histogram, atomicAdd it to the corresponding global histogram bucket



For a low bucket size, this is better than the first parallel method because there won't be as much contention (there are much less atomicAdds happening). On the down side, we're potentially losing some parallelism because each thread is doing a bunch of work sequentially. Depending on how you set things up (number of parallel threads the GPU can run, cost of

booting up a new thread, how much the contention from atomicAdd costs, etc..), this may not be a problem.

Python example…

```python
import numpy as np
from numba import cuda, int32


BUCKETS = 5
MAX_VAL = 50
RANGE = MAX_VAL / BUCKETS

ELEM_PER_THREAD = 100


@cuda.jit
def hist(in_data, out_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    # create local histogram array
    out_local = cuda.local.array(BUCKETS, int32)
    for i in range(0, BUCKETS):
        out_local[i] = 0

    # compute local histogram
    for i in range(0, ELEM_PER_THREAD):
        val = in_data[idx][i]
        bucket = int(val / RANGE)
        out_local[bucket] += 1

    # atomic add each element of local hist to global hist
    for i in range(0, BUCKETS):
        val = out_local[i]
        cuda.atomic.add(out_data, i, val)


np.random.seed(0)

in_data = np.random.randint(0, MAX_VAL, [65536, ELEM_PER_THREAD])
out_data = np.zeros(BUCKETS, dtype=np.int32)
hist[8, 1024](in_data, out_data)
```

```
print(str(out_data))
```

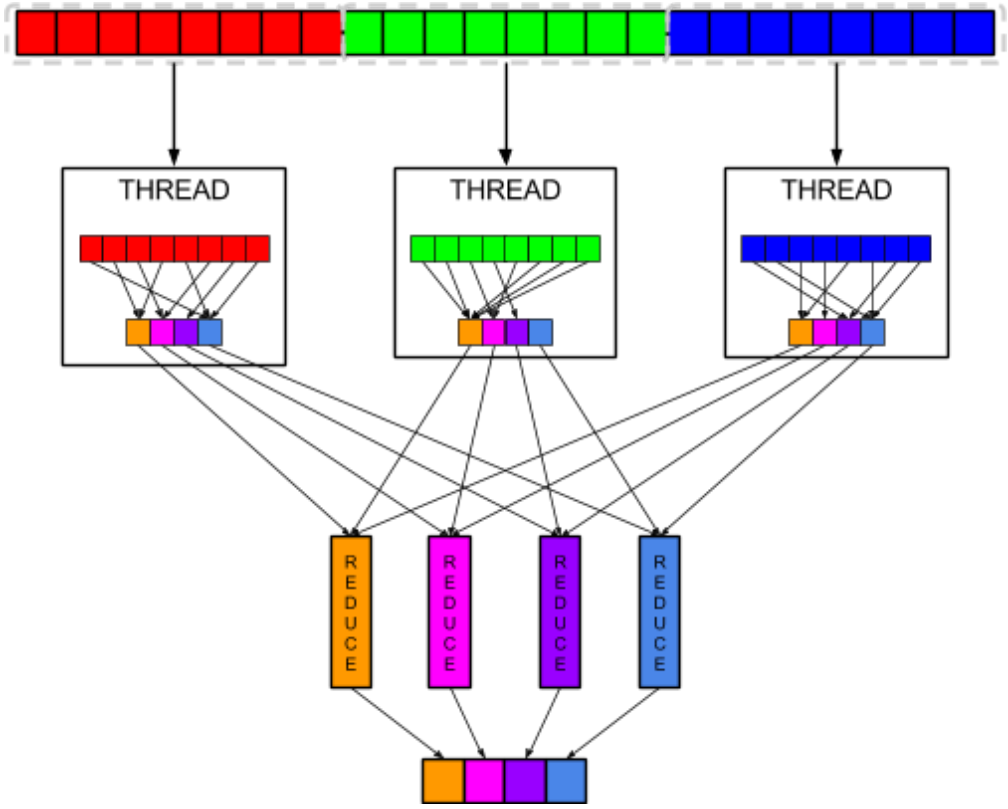
Output is…

```
[163866 163560 163849 163700 164225]
```

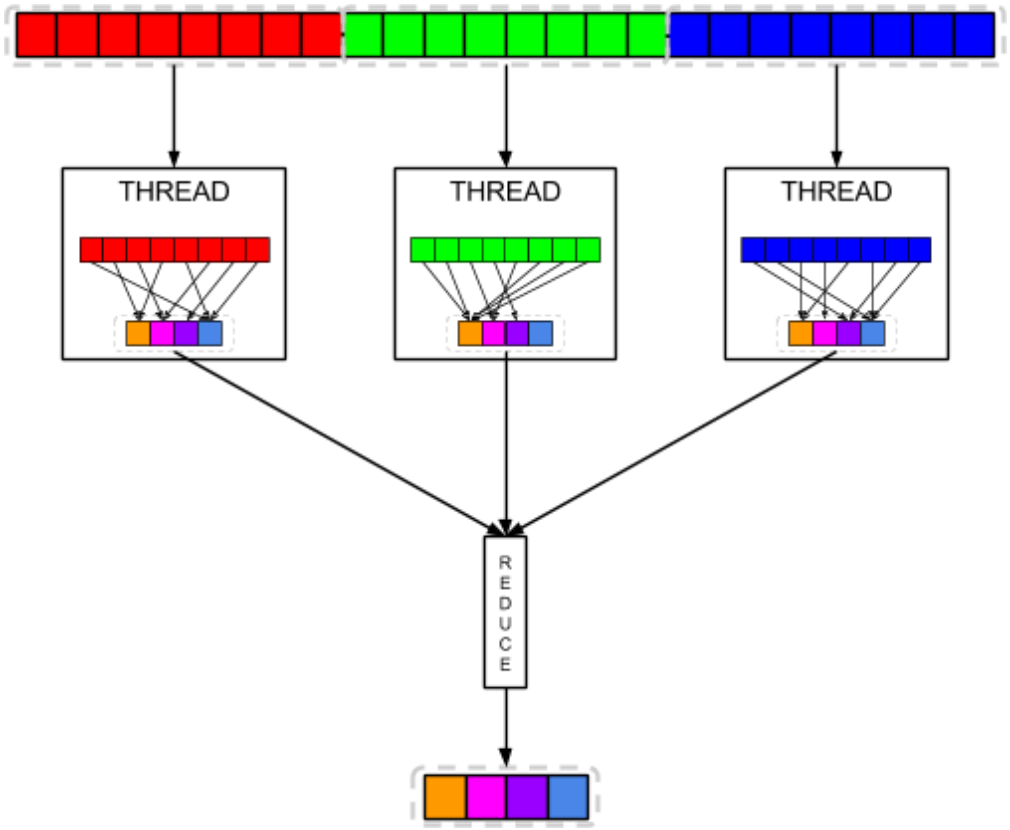
Parallel (Thread Local Histogram and Reduce)

Launch a thread for each n elements and have each thread calculate the histogram locally just for those n elements (sequentially). Then, once all the local histograms have been computed, you run them through a reduce (addition) to get the global histogram.

The trivial way to do reduce would be, for each index i, run a reduce for that element across all local histograms and put it in the corresponding index for the global histogram...



A less-trivial way to do reduce would be to treat each local histogram as a vector/matrix, and use vector/matrix addition as your reduce operator. Essentially, we're going to be doing a single reduce because we'll be treating each local histogram as an input element for the reduce. This will spawn less threads but it'll do more work per thread -- whether this is better or worse depends on your setup.



NOTE: Remember that reduce requires an operator that takes 2 inputs (binary) and is associative/commutative. The vector/matrix addition meets both criteria.

Python example…

```
import numpy as np
from numba import cuda


BUCKETS = 5
MAX_VAL = 50
RANGE = MAX_VAL / BUCKETS

ELEM_PER_THREAD = 100


@cuda.jit
def hist_local(in_data, out_data):
   idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

   # compute local histogram
```

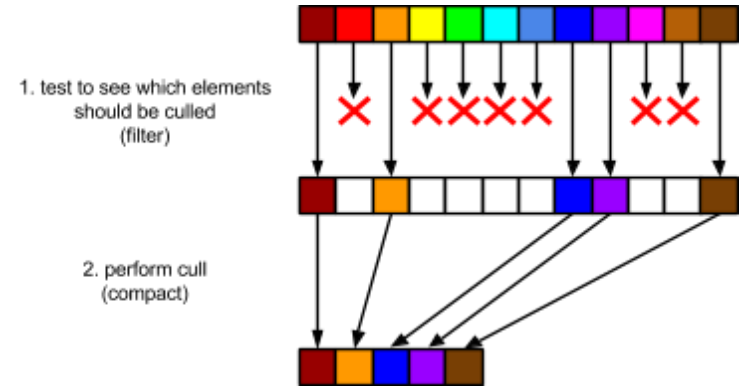
```
    for i in range(0, ELEM_PER_THREAD):
        val = in_data[idx][i]
        bucket = int(val / RANGE)
        out_data[idx][bucket] += 1


@cuda.jit
def reduce_hists(data, hop):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    in1_idx = idx * hop * 2
    in2_idx = in1_idx + hop
    out_idx = in1_idx

    for i in range(0, BUCKETS):
        data[out_idx][i] = data[in1_idx][i] + data[in2_idx][i]


np.random.seed(0)

in_data = np.random.randint(0, MAX_VAL, [65536, ELEM_PER_THREAD])
out_data = np.zeros([65536, BUCKETS], dtype=np.int32)
hist_local[64, 1024](in_data, out_data)

reduce_hists[32, 1024](out_data, 1)
reduce_hists[16, 1024](out_data, 2)
reduce_hists[8, 1024](out_data, 4)
reduce_hists[4, 1024](out_data, 8)
reduce_hists[2, 1024](out_data, 16)
reduce_hists[1, 1024](out_data, 32)
reduce_hists[1, 512](out_data, 64)
reduce_hists[1, 256](out_data, 128)
reduce_hists[1, 128](out_data, 256)
reduce_hists[1, 64](out_data, 512)
reduce_hists[1, 32](out_data, 1024)
reduce_hists[1, 16](out_data, 2048)
reduce_hists[1, 8](out_data, 4096)
reduce_hists[1, 4](out_data, 8192)
reduce_hists[1, 2](out_data, 16384)
reduce_hists[1, 1](out_data, 32768)

print(str(out_data[0]))
```

Output is…

```
[1310115 1310778 1309461 1310735 1312511]
```

Compact (Filter)

Compact is when you filter out elements from your data based on some condition/predicate. Most of the time this is referred to as just compact, because technically the dataset is being compacted by removing elements that aren't needed for further computations down the line…



This becomes a useful operation if you're going to be doing a non-trivial amount of computation on each element AND it's inexpensive (relatively) to find out which elements to avoid computations on.

Serial

Python example…

```
in_data = [1, 5, 6, 7, 0, 1, 3, 4, 2, 2, 2, 9]
print("original:  " + str(in_data))

#filter out even numbers
in_data_filtered = []
for x in in_data:
    in_data_filtered.append(x % 2)

print("filtered:  " + str(in_data_filtered))


#compact so that all even numbers are removed
in_data_compacted = []
for i in range(len(in_data)):
    if in_data_filtered[i] == 0:
```

```
        continue

    in_data_compacted.append(in_data[i])

print("compacted: " + str(in_data_compacted))
```

Output is…

```
original:  [1, 5, 6, 7, 0, 1, 3, 4, 2, 2, 2, 9]
filtered:  [1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1]
compacted: [1, 5, 7, 1, 3, 9]
```

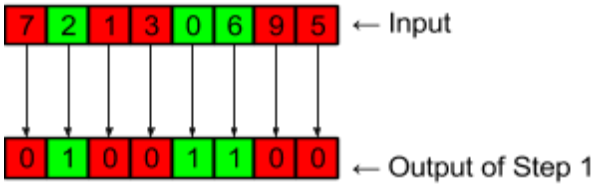
Parallel

Filter and compact is an application of several parallel primitives. The steps are as follows...

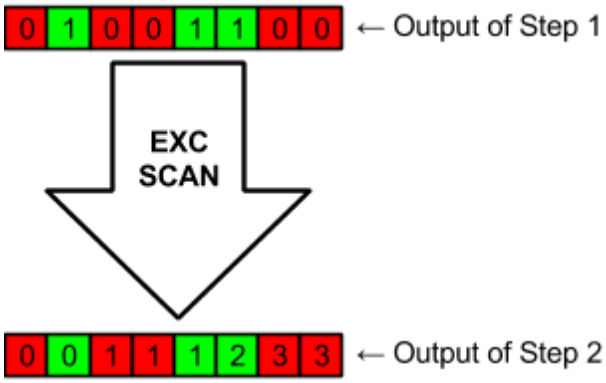
1. map through a predicate -- for each element, output 1 to keep and 0 to discard
2. exclusive scan the output of 1 -- this generates address offsets
3. scatter the data -- for each element, if we decided to keep it in step1, copy it to the address offset we computed in step 2

For example, imagine we had an input array of numbers and we wanted to get rid of all the odd numbers….

The first step is to take your input elements and map it through a predicate: generating 1s for the elements we want to keep (even numbers) and 0 for the elements we want to discard (odd numbers)…



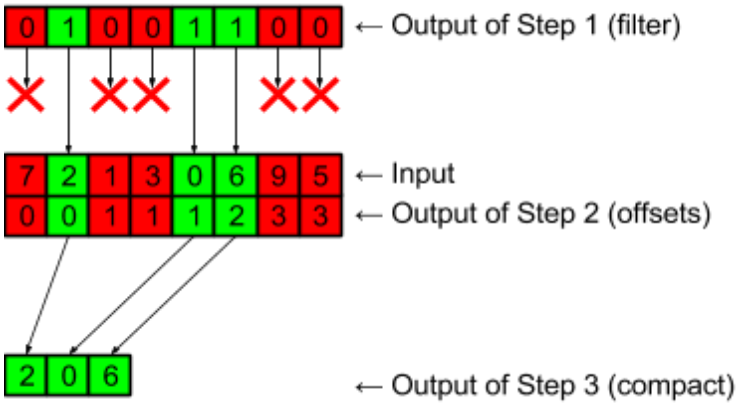
The second step is to take the output of step 1 and perform an exclusive scan on it. This will generate a unique address offset for each element that we want to keep...



NOTE: Having trouble understanding what's going on here? Look at the elements in our original input array. For each element that we determined we want to keep (is even), the corresponding location in the output of our exclusive scan points to where we should move it to. So…

- offset 0 is where we should put the 1st element (2)
- offset 1 is where we should put the 2nd element (0)
- offset 2 is where we should put the 3rd element (6)

The third step is to scatter the elements based on the outputs of step 1 and step 2. Essentially what we do is… for each elem, if we determined that we want to keep it in step 1, we move/copy it to the offset we determined in step 2…



NOTE: Remember that scatter is just a special case of map. All it means is that the each output will have a corresponding input, but not every input will have a corresponding output.

Python example…

```
import numpy as np
from numba import cuda
```

```
@cuda.jit
def filter(in_data, out_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if in_data[idx] % 2 == 0:
        out_data[idx] = 1
    else:
        out_data[idx] = 0


@cuda.jit
def hillissteele_scan_addop(in_data, out_data, step):
    skip = 2**step

    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < skip:
        out_data[idx] = in_data[idx]
    else:
        out_data[idx] = in_data[idx] + in_data[idx-skip]


@cuda.jit
def inc_scan_to_exc_scan(in_data, out_data, identity):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if idx == 0:
        out_data[0] = identity
    else:
        out_data[idx] = in_data[idx-1]


@cuda.jit
def compact(in_data, out_data, filters, offsets):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if filters[idx] == 0:
        return

    offset = offsets[idx]
    out_data[offset] = in_data[idx]


# generate input
```

```
np.random.seed(0)
in_data = np.random.randint(0, 10, 16)
print("input:           " + str(in_data))

# run filter on input (keep only even nums)
filter_out_data = np.zeros(16, dtype=np.int32)
filter[1, 16](in_data, filter_out_data)
print("filter:          " + str(filter_out_data))

# run scan on filters using hillissteele scan (inc. scan)
inc_scan_out_data = np.copy(filter_out_data)  # copy filter out
hillissteele_scan_addop[1, 16](inc_scan_out_data, inc_scan_out_data, 0)
print("inc scan step1:  " + str(inc_scan_out_data))
hillissteele_scan_addop[1, 16](inc_scan_out_data, inc_scan_out_data, 1)
print("inc scan step2:  " + str(inc_scan_out_data))
hillissteele_scan_addop[1, 16](inc_scan_out_data, inc_scan_out_data, 2)
print("inc scan step3:  " + str(inc_scan_out_data))
hillissteele_scan_addop[1, 16](inc_scan_out_data, inc_scan_out_data, 3)
print("inc scan step4:  " + str(inc_scan_out_data))

# convert inc. scan to exc. scan (required by compact)
exc_scan_out_data = np.zeros(16, dtype=np.int32)
inc_scan_to_exc_scan[1, 16](inc_scan_out_data, exc_scan_out_data, 0)
print("conv to exc scan:" + str(exc_scan_out_data))

# run compact
compact_out_data = np.full(16, -1, dtype=np.int32)
compact[1, 16](in_data,
               compact_out_data,
               filter_out_data,
               exc_scan_out_data)
print("compact:         " + str(compact_out_data))
```

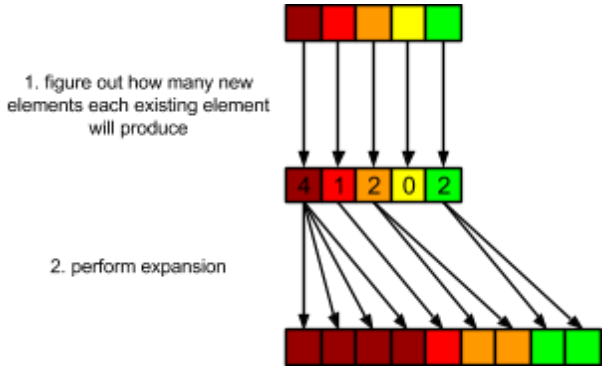
Output is…

```
input:           [5 0 3 3 7 9 3 5 2 4 7 6 8 8 1 6]
filter:          [0 1 0 0 0 0 0 0 1 1 0 1 1 1 0 1]
inc scan step1:  [0 1 1 0 0 0 0 0 1 2 1 1 2 2 1 1]
inc scan step2:  [0 1 1 1 1 0 0 0 1 2 2 3 3 3 3 3]
inc scan step3:  [0 1 1 1 1 1 1 1 2 2 2 3 4 5 5 6]
inc scan step4:  [0 1 1 1 1 1 1 1 2 3 3 4 5 6 6 7]
conv to exc scan:[0 0 1 1 1 1 1 1 1 2 3 3 4 5 6 6]
```

```
compact:        [ 0  2  4  6  8  8  6 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

Allocation

Allocation is for when each element in your dataset might get split up into many elements and the number of elements isn't fixed or known beforehand. It figures out your final destination array size and where each new element will sit.

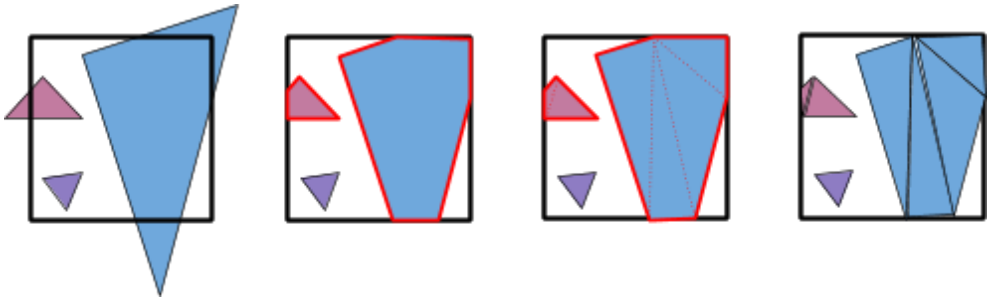


> **NOTE**: If each element was split into some fixed size or the size, you could simply preallocate the space. Also, you could always preallocate the maximum possible size of your destination array (assuming you knew what that was), but that's considered wasteful.

For example, imagine that each element in your dataset was a sentence. You want to split each sentence up by space such that your dataset becomes the individual words that make up the sentences. This would be an allocation operation.

Another example is culling triangles. During the 3D rendering process, you need to clip triangles that intersect with bounds of your viewport. The problem is that when you clip a triangle, you

might end up with shapes that aren't triangles (e.g. quads and n-gons). Those non-triangle shapes need to be split up back into triangles so they can be drawn using efficient algorithms.



Serial

Python example…

```
in_data = [
    "Hi Steve, this is Bob",
    "The movie was great!",
    "My cake was made with buttercream"]
print("original...")
[print("  " + x) for x in in_data]

# for each element N... if N is an even number, copy it N times
out_data = []
for x in in_data:
    out_data += x.split()

print("expanded...")
[print("  " + x) for x in out_data]
```

Output is…

```
original...
  Hi Steve, this is Bob
  The movie was great!
  My cake was made with buttercream
expanded...
  Hi
  Steve,
  this
  is
  Bob
  The
  movie
```

```
was
great!
My
cake
was
made
with
buttercream
```

Parallel

Allocation is an application of several parallel primitives. The steps are as follows...

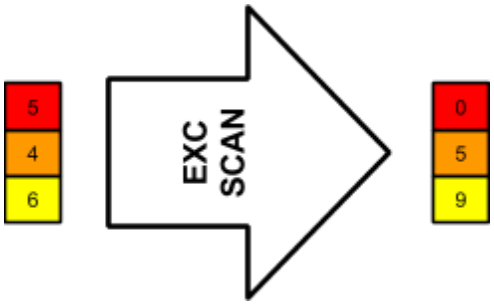
1. map to split counts -- for each element, output how many elements it splits into
2. exclusive scan the output of 1 -- this generates address offsets
3. scatter the data -- for each element, if we decided to keep it in step1, copy it to the address offset we computed in step 2

Notice what's happening here. This is almost exactly the same as compact. The only difference is that, in step 1 we're finding out the number of elements to create instead of testing to see if we should keep the element.

For example, imagine we had an input array of strings. We want to split each string by space...

The first step is to take your input elements and map it through a kernel that generates how many outputs each element will have. In this case, it will have the number of words in the sentence…

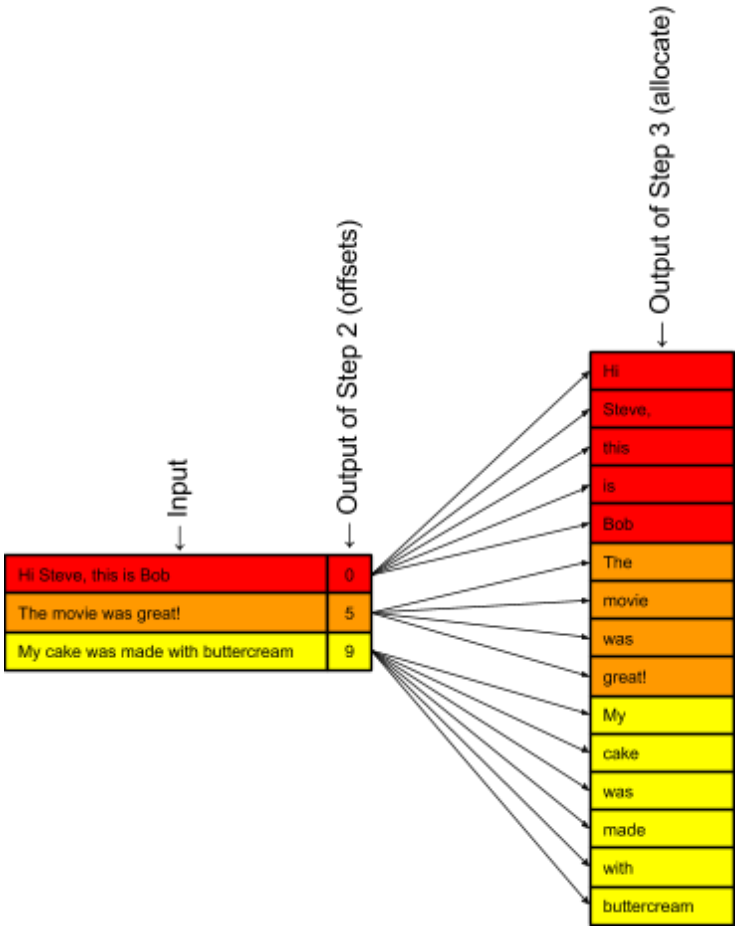
The second step is to take the output of step 1 and perform an exclusive scan on it. This will generate a unique address offset for the new elements (address offsets for the words in our sentences)…



NOTE: Having trouble understanding what's going on here? Look at the output of our exclusive scan. It tell us where we should start putting the words for each sentence. So the words from…

- sentence 1 (5 words) will start at offset 0 and end at offset 4
- sentence 2 (4 words) will start at offset 5 and end at offset 8
- sentence 3 (6 words) will start at offset 9 and end at offset 14

The third step is to scatter the elements based on the outputs of steps 1 and 2. This is where we perform the actual splitting. From step 2, we know how many elements our final destination array will have. We can allocate the correct number of elements, then run a kernel splits up each sentence copies the results to the destination array (at the offset determined in step 2)…



NOTE: Remember that scatter is just a special case of map. All it means is that the each output will have a corresponding input, but not every input will have a corresponding output.

Python example…

NOTE: This is an awful example, but there's enough to show the different steps of allocate.

```
import numpy as np
from numba import cuda


@cuda.jit
def wordcount(in_data, out_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if in_data[idx][0] == 0x00:
        return
```

```
    cnt = 1
    wordlen = 0
    for ch in in_data[idx]:
        if ch == 0x20 and wordlen > 0:
            cnt += 1
            wordlen = 0
        elif ch != 0x00 and ch != 0x20:
            wordlen += 1

    out_data[idx] = cnt


@cuda.jit
def hillissteele_scan_addop(in_data, out_data, step):
    skip = 2**step

    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < skip:
        out_data[idx] = in_data[idx]
    else:
        out_data[idx] = in_data[idx] + in_data[idx-skip]


@cuda.jit
def inc_scan_to_exc_scan(in_data, out_data, identity):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if idx == 0:
        out_data[0] = identity
    else:
        out_data[idx] = in_data[idx-1]


@cuda.jit
def split(in_data, out_data, offsets):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    in_arr = in_data[idx]
    in_offset = offsets[idx]

    wordcnt = 0
```

```
    wordlen = 0
    for i in range(len(in_arr)):
        ch = in_arr[i]
        if ch == 0x20 and wordlen > 0:
            wordlen = 0
            wordcnt += 1
        elif ch != 0x00 and ch != 0x20:
            out_data[in_offset + wordcnt][wordlen] = ch
            wordlen += 1


# helper methods
def as_uint8(s):
    ret = np.zeros([128], dtype=np.uint8)
    for idx in range(len(s)):
        ret[idx] = ord(s[idx])
    return ret
def as_str(a):
    ret = ''
    for idx in range(len(a)):
        if a[idx] != 0:
            ret += chr(a[idx])
    return ret

# create input
in_data = np.zeros([8, 128], dtype=np.uint8)
in_data[0] = as_uint8("Hi Steve, this is Bob")
in_data[1] = as_uint8("The movie was great!")
in_data[2] = as_uint8("My cake was made with buttercream")
print("input...")
for a in in_data: print("  " + as_str(a))

# run filter on input (keep only even nums)
wordcount_out_data = np.zeros(8, dtype=np.int32)
wordcount[1, 8](in_data, wordcount_out_data)
print("wordcount:       " + str(wordcount_out_data))

# run scan on filters using hillissteele scan (inc. scan)
inc_scan_out_data = np.copy(wordcount_out_data)
hillissteele_scan_addop[1, 8](inc_scan_out_data, inc_scan_out_data, 0)
print("inc scan step1:  " + str(inc_scan_out_data))
hillissteele_scan_addop[1, 8](inc_scan_out_data, inc_scan_out_data, 1)
```

```
print("inc scan step2:  " + str(inc_scan_out_data))
hillissteele_scan_addop[1, 8](inc_scan_out_data, inc_scan_out_data, 2)
print("inc scan step3:  " + str(inc_scan_out_data))

# convert inc. scan to exc. scan (required by allocate)
exc_scan_out_data = np.zeros(8, dtype=np.int32)
inc_scan_to_exc_scan[1, 8](inc_scan_out_data, exc_scan_out_data, 0)
print("conv to exc scan:" + str(exc_scan_out_data))

# split to individual words
final_size = exc_scan_out_data[-1]  # hack, last sentence must have 0 words
final_out_data = np.zeros([final_size, 128], dtype=np.uint8)
split[1, 8](in_data,
            final_out_data,
            exc_scan_out_data)
print("final...")
for a in final_out_data: print("  " + as_str(a))
```

Output is…

```
input...
  Hi Steve, this is Bob
  The movie was great!
  My cake was made with buttercream




wordcount:       [5 4 6 0 0 0 0 0]
inc scan step1:  [ 5  9 10  6  0  0  0  0]
inc scan step2:  [ 5  9 15 15 10  6  0  0]
inc scan step3:  [ 5  9 15 15 15 15 15 15]
conv to exc scan:[ 0  5  9 15 15 15 15 15]
final...
  Hi
  Steve,
  this
  is
  Bob
  The
  movie
```

```
was
great!
My
cake
was
made
with
buttercream
```

Sparse Matrix Vector Multiplication (SpMV)

Sparse Matrix Vector Multiplication (SpMV) is when you multiply a sparse matrix with a vector.

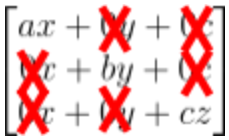
What is a sparse matrix? A sparse matrix is a matrix where the majority of elements are 0...

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & b & 0 & c \\ 0 & 0 & 0 & 0 & d & 0 \end{bmatrix}$$

Typically, if you were do a matrix vector multiplication, it would look something like this...

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + 0y + 0c \\ 0x + by + 0c \\ 0x + 0y + cz \end{bmatrix}$$

Notice that in the example above, the matrix is sparse: many of the elements are 0. The elements that are 0 still require the same amount of work to compute but ultimately contribute nothing to the final result...



We can avoid holding onto and computing these 0s by using a structure called compressed sparse row (CSR) to represent our matrix. CSR is comprised of 3 arrays…

- values → non-zero elements of the matrix
- columns → which matrix column each of elements in values are in
- row pointers → index in values of the starting element of each matrix row

So for example, here is the CSR representation of the following matrix…

$$\begin{bmatrix} a & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & b & 0 & c \\ 0 & 0 & 0 & 0 & d & 0 \end{bmatrix}$$

values → [a, b, c, d]
columns → [0, 3, 5, 4]
row pointers → [0, 1, 3]

With CSR, we have quick access to the non-zero matrix elements along with their location in the matrix. CSRs are important for both parallel implementation and serial implementation of SpMV. This will be discussed further in the parallel subsection.

There are many cases in the wild where you'll need to deal with sparse matrices. Some examples are…

- recommender systems (e.g. Google's pagerank or NetFlix's recommendations)
- simulations
- graphics
- machine learning

Serial

Python example…

```
import numpy as np


class CSR:
    def __init__(self, mat):
        self.values = []
        self.columns = []
        self.row_ptrs = []
        self.shape = mat.shape
        for x in range(mat.shape[0]):
            first = True
            for y in range(mat.shape[1]):
                if mat[x][y] != 0:
                    self.values.append(mat[x][y])
                    self.columns.append(y)
                    if first:
                        idx = len(self.values) - 1
                        self.row_ptrs.append(idx)
                        first = False

    def __str__(self):
        return "values=  " + str(self.values) + "\n"\
               + "columns= " + str(self.columns) + "\n"\
               + "row_ptrs=" + str(self.row_ptrs)
```

```
mat = np.array([[3, 0, 0, 0, 0, 0],
                [0, 0, 0, 1, 0, 2],
                [0, 0, 0, 0, 4, 0]], dtype=np.int32)
vec = np.array([[1], [2], [3], [4], [5], [6]])
mat_csr = CSR(mat)
print(mat_csr)


res = np.zeros((mat.shape[0], 1), dtype=np.int32)
row_boundaries = set(zip(mat_csr.row_ptrs, mat_csr.row_ptrs[1:]))
row_boundaries.add((mat_csr.row_ptrs[-1], len(mat_csr.values)))
row = 0
print("row bounds: " + str(row_boundaries))
for row_bounds in row_boundaries:
    for idx in range(row_bounds[0], row_bounds[1]):
        col = mat_csr.columns[idx]
        mat_val = mat_csr.values[idx]
        vec_val = vec[col][0]
        res[row] += mat_val * vec_val

    row += 1

# uncomment this to verify that the results are correct
print("expected...")
print(np.dot(mat, vec))
print("actual...")
print(res)
```

Output is…

```
values=  [3, 1, 2, 4]
columns= [0, 3, 5, 4]
row_ptrs=[0, 1, 3]
row bounds: {(0, 1), (1, 3), (3, 4)}
expected...
[[ 3]
 [16]
 [20]]
actual...
[[ 3]
```

```
 [16]
 [20]]
```

Parallel

SpMV is an application of several primitives. The steps are as follows...

1. map the CSR column array to the corresponding values in the vector
2. map the CSR value array such that each element is multiplied by the corresponding element in step 1's output
3. for each CSR row segment, reduce the corresponding part of step 3's output by addition

For example, imagine that we wanted to multiply the following matrix and vector together…

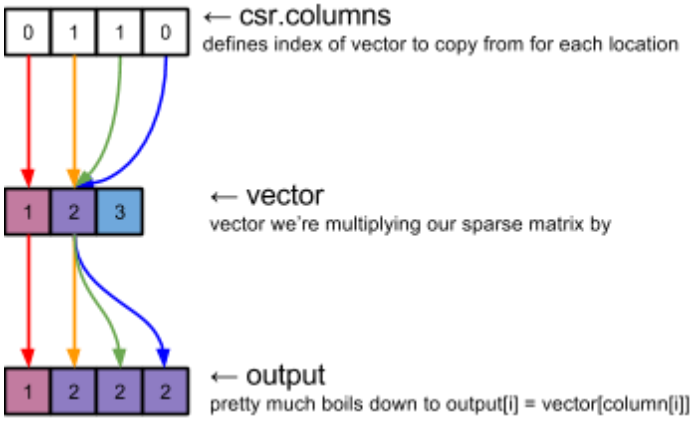
$$\begin{bmatrix} 3 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Before we do anything, we need to get our matrix into compressed sparse row (CSR) format. The high-level steps to convert a matrix to CSR was covered in the main section, so I won't go over it again. Once we convert the matrix to CSR format, it should look like this...

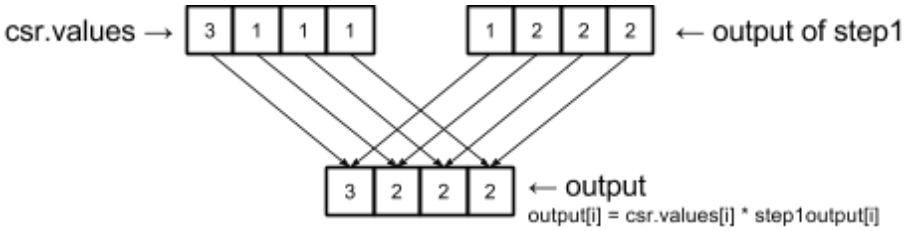
csr.values → [3, 1, 1, 1]
csr.columns → [0, 1, 1, 0]
csr.rowpointers → [0, 2, -1, 3]

> **NOTE**: I'm not going to cover it here, but it's fairly simple to convert a matrix to CSR in parallel. It boils doing to doing map(s) + compact. It's fairly simple to figure out.
>
> **NOTE**: Why is the 2nd element of csr.rowpointers -1? Because the entire 3rd row is 0s, which means there is no index in csr.values for row 3.

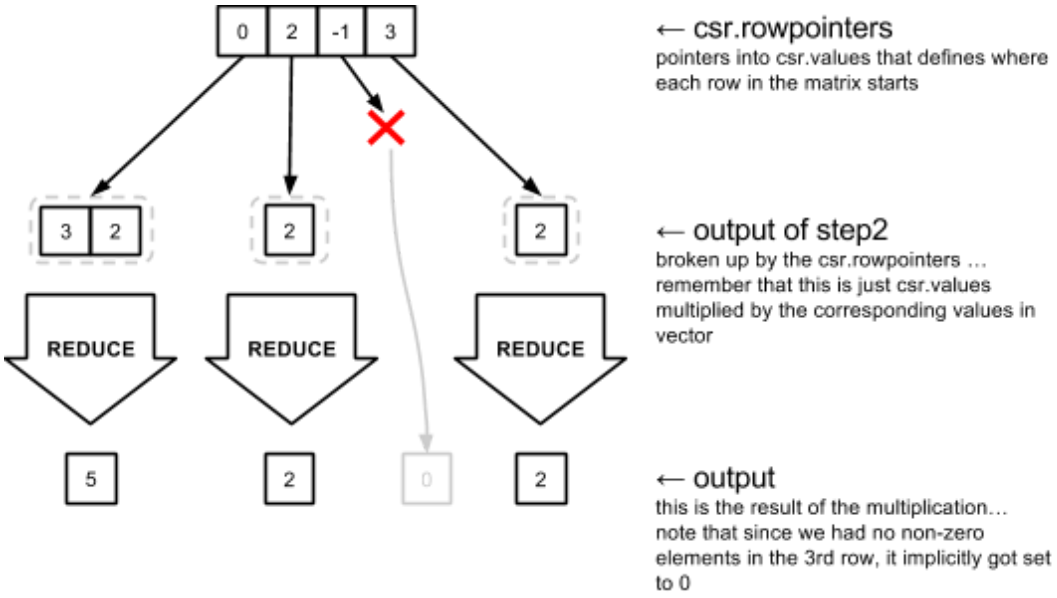
The first step is to map the csr.columns to the corresponding values in the vector. For each element in the csr.columns (c), we're mapping it to vector[c]...



The second step is to map csr.values such that each element is multiplied by the corresponding element in step 1's output…



The third step is to break up the output of step 2 based on csr.rowpointer, and perform an reduce (using add operator) on each of those segments…



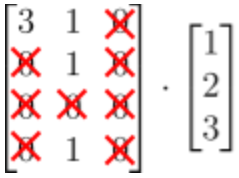
NOTE: For this step… instead of reducing each segment independently, you can try using a segmented reduce or a segmented scan (if you use scan you would use the final

element of the result). Remember that kernels run one a time back-to-back. Using a segmented reduce/scan means less kernel launches (potentially faster results). The reason I didn't do this is because I don't know how to implement segmented reduce/scan yet. It would probably complicate the example as well. For more information see the segmented scan section.

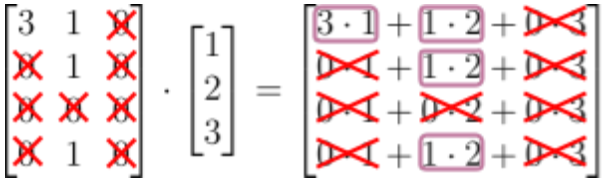
Notice how this all fits together. The entire multiplication operation is computed like this...

$$\begin{bmatrix} 3 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 \\ 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 \\ 0 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 \\ 0 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 0 \\ 2 \end{bmatrix}$$

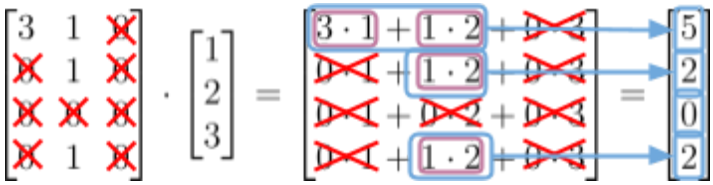
The zeros in the matrix contribute nothing to the final result, but still require work to compute. By converting the matrix to CSR format, we're avoiding all the zeros in the matrix…



Steps 1 and 2 does all the multiplications required to compute the final results...



Step 3 adds up the multiplications done in steps1 and 2, by row, to get the final result…



Python example…

```
from math import log, ceil
from numba import cuda
import numpy as np
```

```
class CSR:
    def __init__(self, mat):
        values = []
        columns = []
        row_ptrs = []
        for x in range(mat.shape[0]):
            last_len = len(values)
            for y in range(mat.shape[1]):
                if mat[x][y] != 0:
                    values.append(mat[x][y])
                    columns.append(y)
            row_ptrs.append((last_len, len(values)))
        self.shape = mat.shape
        self.values = np.array(values, dtype=np.int32)
        self.columns = np.array(columns, dtype=np.int32)
        self.row_ptrs = row_ptrs

    def __str__(self):
        return "values=  " + str(self.values) + "\n"\
               + "columns= " + str(self.columns) + "\n"\
               + "row_ptrs=" + str(self.row_ptrs)


@cuda.jit
def map_out_vector(in_cols, in_vec, out_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    col = in_cols[idx]
    out_data[idx] = in_vec[0][col]


@cuda.jit
def map_mult(in_val, in_expanded_vec, out_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    out_data[idx] = in_val[idx] * in_expanded_vec[idx]


@cuda.jit
def reduce_add(data, hop):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    out_idx = idx * hop
    in1_idx = idx * hop
```

```
    in2_idx = idx * hop + (hop//2)

    if in2_idx < len(data):
        data[out_idx] = data[in1_idx] + data[in2_idx]


# create mat and vec for spmv
mat = np.array([[3, 1, 0],
                [0, 1, 0],
                [0, 0, 0],
                [0, 1, 0]], dtype=np.int32)
vec = np.array([[1], [2], [3]], dtype=np.int32)
print("matrix and vector being used for SpMV...")
print(" mat")
print(mat)
print(" vec")
print(vec)

# convert mat to csr format
print("\nconvert matrix to CSR format...")
mat_csr = CSR(mat)  # serial, but easy to do in parallel: compact
nonzero_len = mat_csr.values.shape[0]
print(mat_csr)

# step 1
print("\nstep 1. expand out vec based on csr.columns...")
vec_expand_res = np.zeros(nonzero_len, dtype=np.int32)
map_out_vector[1, nonzero_len](mat_csr.columns, vec, vec_expand_res)
print(vec_expand_res)

# step 2
print("\nstep 2. multiply expanded vec by csr.values...")
mult_res = np.zeros(nonzero_len, dtype=np.int32)
map_mult[1, nonzero_len](mat_csr.values, vec_expand_res, mult_res)
print(mult_res)

# step 3
print("\nstep 3. breakup and reduce...")
res = np.zeros((mat.shape[0], 1), dtype=np.int32)
for row in range(len(mat_csr.row_ptrs)):
    region = mat_csr.row_ptrs[row]
    start = region[0]
```

```
    stop = region[1]
    segment = mult_res[start:stop]

    if len(segment) == 0:
        continue
    elif len(segment) == 1:
        res[row][0] = segment[0]
        continue

    # reduce segment -- result will be in segment[0]
    hops = 2
    threads = len(segment) / hops
    threads = 2 ** ceil(log(threads, 2))  # round up to nearest pow of 2
    while threads > 0:
        blocks = (threads // 1024) + 1
        threads_per_block = 1024

        # run reduce step
        reduce_add[blocks, threads_per_block](segment, hops)

        hops *= 2
        threads //= 2
    res[row][0] = segment[0]

print(res)
```

Output is…

```
matrix and vector being used for SpMV...
 mat
[[3 1 0]
 [0 1 0]
 [0 0 0]
 [0 1 0]]
 vec
[[1]
 [2]
 [3]]

convert matrix to CSR format...
values=  [3 1 1 1]
```

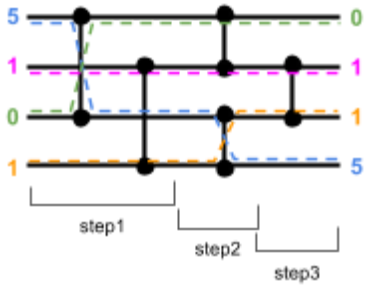
```
columns= [0 1 1 1]
row_ptrs=[(0, 2), (2, 3), (3, 3), (3, 4)]

step 1. expand out vec based on csr.columns...
[1 2 2 2]

step 2. multiply expanded vec by csr.values...
[3 2 2 2]

step 3. breakup and reduce...
[[5]
 [2]
 [0]
 [2]]
```

Sorting Networks

Sorting networks are networks that take a fixed number of inputs, perform a fixed number of compare-and-swap operations on those inputs, and ultimately output the inputs as a sorted list.

For example, here's a sorting network being used to sort a 2 element list. This is the simplest case of a sorting network...

Here's another example where 4 elements are being sorted. Note that as long as the compare-and-swaps aren't hitting the same element, they can be grouped into the same parallel step. In the example below, we can perform everything in 3 parallel steps.



> **NOTE**: Look what's happening in the network. It isn't limited to the input [5,1,0,1]. It will sort any input of size 4.

Sorting networks are often referred to as oblivious. This is because, unlike other more complex sorting algorithms, they don't make decisions based off of the data they encounter (e.g.

quicksort). Instead, they're making fixed decisions as to what to compare-and-swap at each parallel step.

This property makes them a popular choice for implementing in hardware and on GPUs. While there may be a high number of steps, there is very little in terms of thread divergence: no complex loops/switches/ifs/etc (see optimizations section on why this is a good thing).

Sorting networks may not be the best choice for sorting over a very large array. They may however be a good choice for sorting smaller arrays that fit in shared memory (maybe as part of a larger sorting algorithm). If you choose to do this, remember that you can use thread barriers to sync between the threads in the block which the shared memory is for vs launching a new kernel for each step.

> **NOTE**: The lessons recommended that if you want to use as a sort network to sort an array in shared memory, it's better to have 1 thread per input. A compare-and-swap will take place on 2 threads, but both those threads will generate the same results. The thread doesn't end after a parallel step. Use a thread barrier to wait until all threads are done for that parallel step before moving to the next parallel step.

There are two main types of a sorting networks: brick/odd-even sort and bitonic sort. There's an independent section available for each.

Serial

This is the 2nd example from the parent section (the 4 input example).

Python example…

```
def compare_and_swap(in_data, idx1, idx2):
    if in_data[idx1] > in_data[idx2]:
        temp = in_data[idx1]
        in_data[idx1] = in_data[idx2]
        in_data[idx2] = temp


data = [5, 1, 0, 1]
compare_and_swap(data, 0, 2)
compare_and_swap(data, 1, 3)
compare_and_swap(data, 0, 1)
compare_and_swap(data, 2, 3)
compare_and_swap(data, 1, 2)
print(str(data))
```

Output is…

```
[0, 1, 1, 5]
```

Parallel

This is the 2nd example from the parent section (the 4 input example).

Python example…

```
from numba import cuda
import numpy as np


@cuda.jit('void(int32[:], int32, int32)', device=True, inline=True)
def compare_and_swap(in_data, idx1, idx2):
    if in_data[idx1] > in_data[idx2]:
        temp = in_data[idx1]
        in_data[idx1] = in_data[idx2]
        in_data[idx2] = temp


@cuda.jit
def sort_network_pass(in_data, net_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    idx1 = net_data[0][idx]
    idx2 = net_data[1][idx]
    compare_and_swap(in_data, idx1, idx2)


step1indices = np.array([[0,2], [1,3]], dtype=np.int32)
step2indices = np.array([[0,1], [2,3]], dtype=np.int32)
step3indices = np.array([[1], [2]], dtype=np.int32)

data = np.array([5,1,0,1], dtype=np.int32)
sort_network_pass[1,2](data, step1indices)
sort_network_pass[1,2](data, step2indices)
sort_network_pass[1,1](data, step3indices)
print(str(data))
```

Output is…

```
[0 1 1 5]
```

Brick/Odd-Even Sort (Sorting Network)

One common sorting algorithm that can easily be made parallel is bubble sort. The parallel version of bubble sort is called brick sort or odd-even sort.

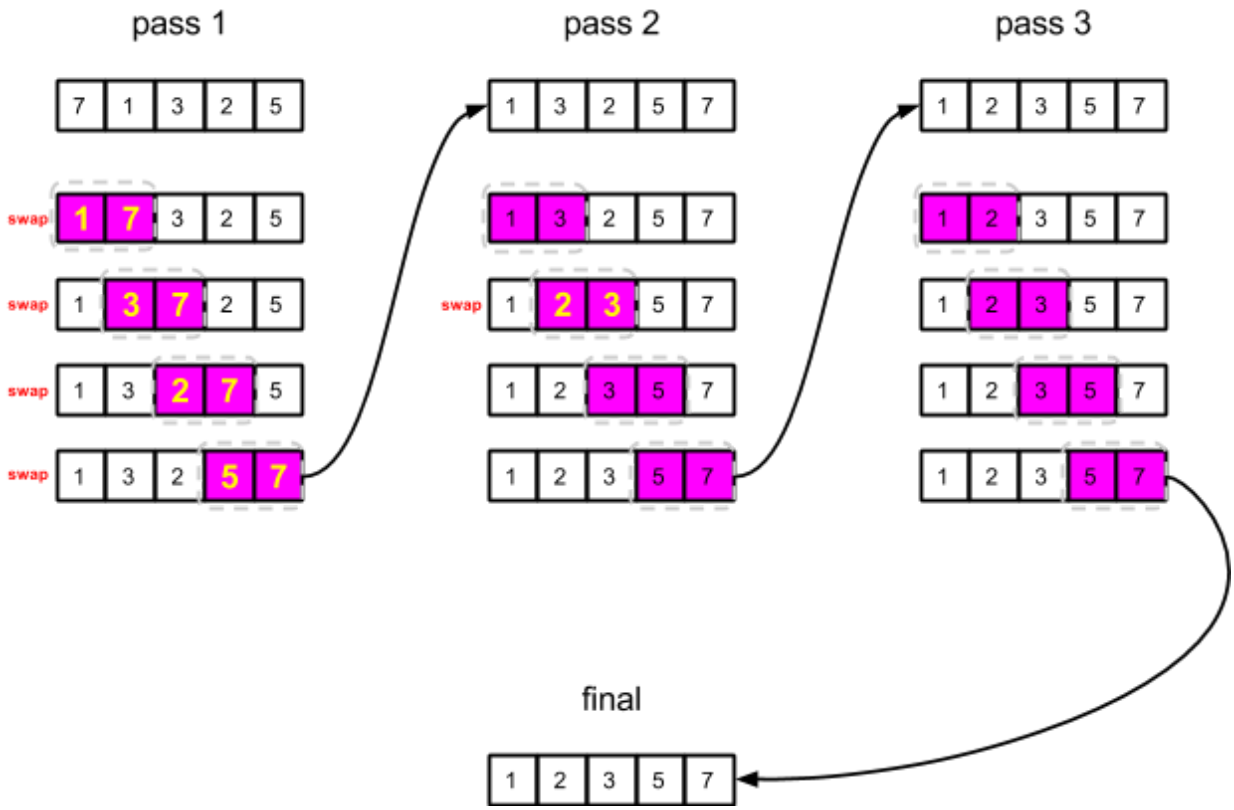
When it comes to algorithm step/work complexity, brick/odd-even sort isn't a particularly efficient parallel algorithm. But, it is an algorithm that can easily exploit a lot of parallelism -- it can be expressed as a sorting network. This will be discussed further in the parallel subsection.

There are better GPU sorts available, but there are also cases where you'll want to use brick/odd-even sort instead. For example, if you want to sort a small list in shared memory, a brick/odd-even sorting network may be the way to go.

Serial

If you don't remember what bubble sort is, it's basically an algorithm that iterates over a list and compares each element to the element next to it, swapping if they're out-of-order. It does this multiple times, until it comes across a pass where no elements were swapped (there were no out-of-order elements).

For example, this is what happens when bubble sort is used to sort the list [7,1,3,2,5]...



Python example…

```
data = [7, 1, 3, 2, 5]

def bubble_sort_pass(in_data):
    swapped = False
    for i in range(len(in_data) - 1):
        if in_data[i] > in_data[i + 1]:
            temp = in_data[i]
            in_data[i] = in_data[i+1]
            in_data[i + 1] = temp
            swapped = True

    return swapped


idx = 1
keep_going = True
while keep_going:
    print("before pass " + str(idx) + ": " + str(data))
    keep_going = bubble_sort_pass(data)
    print(" after pass " + str(idx) + ": " + str(data))
    print("")
    idx += 1

print("        final: " + str(data))
```

Output is…

```
before pass 1: [7, 1, 3, 2, 5]
 after pass 1: [1, 3, 2, 5, 7]

before pass 2: [1, 3, 2, 5, 7]
 after pass 2: [1, 2, 3, 5, 7]

before pass 3: [1, 2, 3, 5, 7]
 after pass 3: [1, 2, 3, 5, 7]

        final: [1, 2, 3, 5, 7]
```

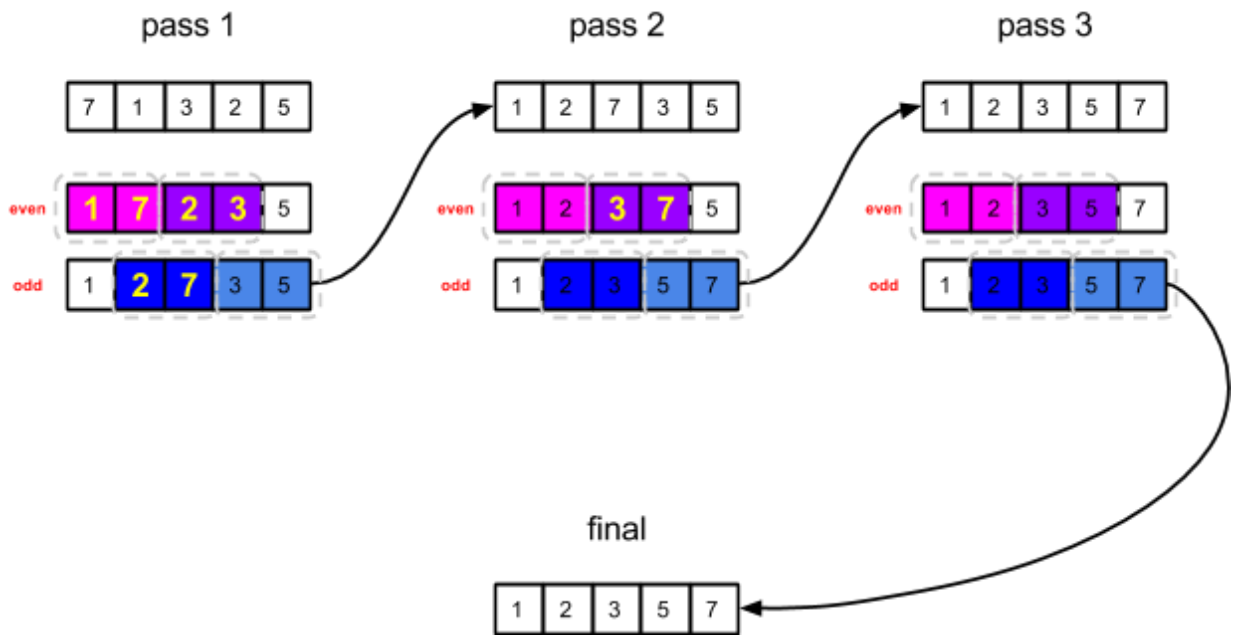
Parallel

Brick/odd-even sort is almost the same as regular bubble sort. It boils down to doing several passes over 2 map primitives. For each pass...

1. map even indices such that they swap data[i] with data[i+1] if out-of-order
2. map odd indices such that they swap data[i] with data[i+1] if out-of-order

Repeat the above 2 maps until all elements are in order.

For example, this is what happens when brick/odd-even sort is used to sort the list [7,1,3,2,5]...



The step complexity of this sort is O(n). This is worst case complexity, which happens when an element that should be at the end of the array is at the beginning of the array. Notice how our example fits this case. We have 7 in the first element when we start, but 7 is all the way at the end when we finish. There were 5 elements overall and we did 6 map operations in total (6 parallel steps).

The work complexity of this sort is $O(n^2)$. Our step complexity is O(n) and we're pretty much going over every element of the array at each parallel step, so the overall work complexity essentially is $O(n*n) \rightarrow O(n^2)$.

This is a poor work/step complexity, but note what's happening in the algorithm. Each parallel step is just a bunch of fixed compare-and-swap operations. This is pretty much a sorting network. The fact that it is a sorting network means that it fits well with one of the main tenets of writing code for the GPU: low thread divergence.

You may not want to use brick/odd-even sort for a large overall sort, but it may be good to use for sorting a local list in shared memory. If you choose to do this, remember that you can use thread barriers to sync between the threads in your block vs launching a new kernel for each step.

> **NOTE**: The lessons recommended that if you want to use as a sort network to sort an array in shared memory, it's better to have 1 thread per input. A compare-and-swap will take place on 2 threads, but both those threads will generate the same results. The thread doesn't end after a parallel step. Use a thread barrier to wait until all threads are done for that parallel step before moving to the next parallel step.

> **NOTE**: See the sorting networks section if you forgot what a sorting network is.

Python example…

```python
from numba import cuda
import numpy as np


@cuda.jit
def even_sort(in_data, swapped_ptr):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    idx *= 2

    if idx+1 >= len(in_data):
        return

    if in_data[idx] > in_data[idx + 1]:
        temp = in_data[idx]
        in_data[idx] = in_data[idx + 1]
        in_data[idx + 1] = temp
        swapped_ptr[0] = 1


@cuda.jit
def odd_sort(in_data, swapped_ptr):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    idx = (idx*2) + 1

    if idx+1 >= len(in_data):
        return
```

```
    if in_data[idx] > in_data[idx + 1]:
        temp = in_data[idx]
        in_data[idx] = in_data[idx + 1]
        in_data[idx + 1] = temp
        swapped_ptr[0] = 1


sort_pass = 1
data = np.array([7, 1, 3, 2, 5], dtype=np.int32)
even_swapped_ptr = np.zeros(1, dtype=np.int32)
odd_swapped_ptr = np.zeros(1, dtype=np.int32)
threads = len(data) // 2
# this is a crappy algo
#   allocates the current num of blocks but all threads per block
#   can be updated to allocate only as many as needed
blocks = (threads // 1024) + 1
threads_per_block = 1024
while True:
    print("sort pass " + str(sort_pass))
    print("----------------")

    even_sort[blocks, threads_per_block](data, even_swapped_ptr)
    print("after even: " + str(data))

    odd_sort[blocks, threads_per_block](data, odd_swapped_ptr)
    print("after odd:  " + str(data))

    print("")

    if even_swapped_ptr[0] == 0 and odd_swapped_ptr[0] == 0:
        break

    even_swapped_ptr[0] = 0
    odd_swapped_ptr[0] = 0

    sort_pass += 1

print("final:      " + str(data))
```

Output is…

```
sort pass 1
```

```
----------------
after even: [1 7 2 3 5]
after odd:  [1 2 7 3 5]

sort pass 2
----------------
after even: [1 2 3 7 5]
after odd:  [1 2 3 5 7]

sort pass 3
----------------
after even: [1 2 3 5 7]
after odd:  [1 2 3 5 7]

final:      [1 2 3 5 7]
```

Bitonic Sort (Sorting Network)

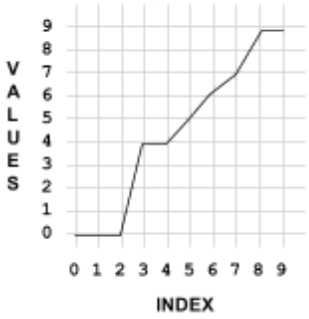
A bitonic sort is a sorting network that expects the input array to be a bitonic or monotonic sequence.

What is a monotonic/bitonic sequence?

- monotonic sequence → sequence of ever increasing OR ever decreasing numbers.
- bitonic sequence → sequence that's made up of 2 monotonic sequences attached together: it either increases then decrease or decreases then increases.

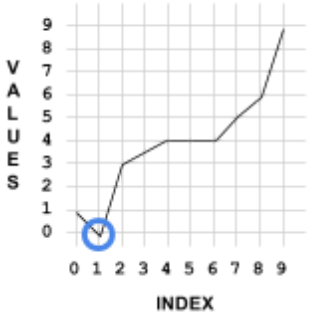
For example…

Example1: This never changes direction -- it's always increasing, so it's a monotonic sequence.



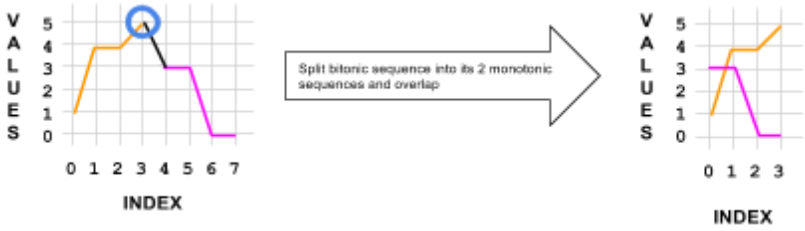
Example2: This changes direction once (highlighted), so it's a bitonic sequence.

Example3: This changes direction once (highlighted), so it's a bitonic sequence.

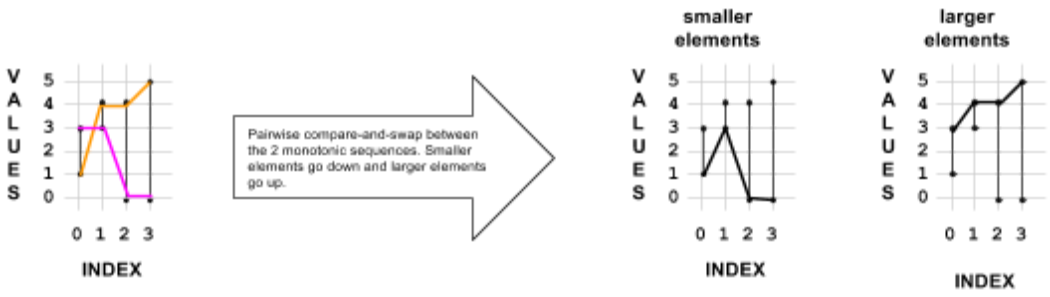


Example4: This changes direction more than once (highlighted), so it is not a bitonic or monotonic sequence.

How does a bitonic sort work? It turns out that if a sequence is bitonic, there's a relatively simple recursive process to get it sorted. The process begins by taking the 2 monotonic sequences that make up the bitonic sequence and overlapping them together...



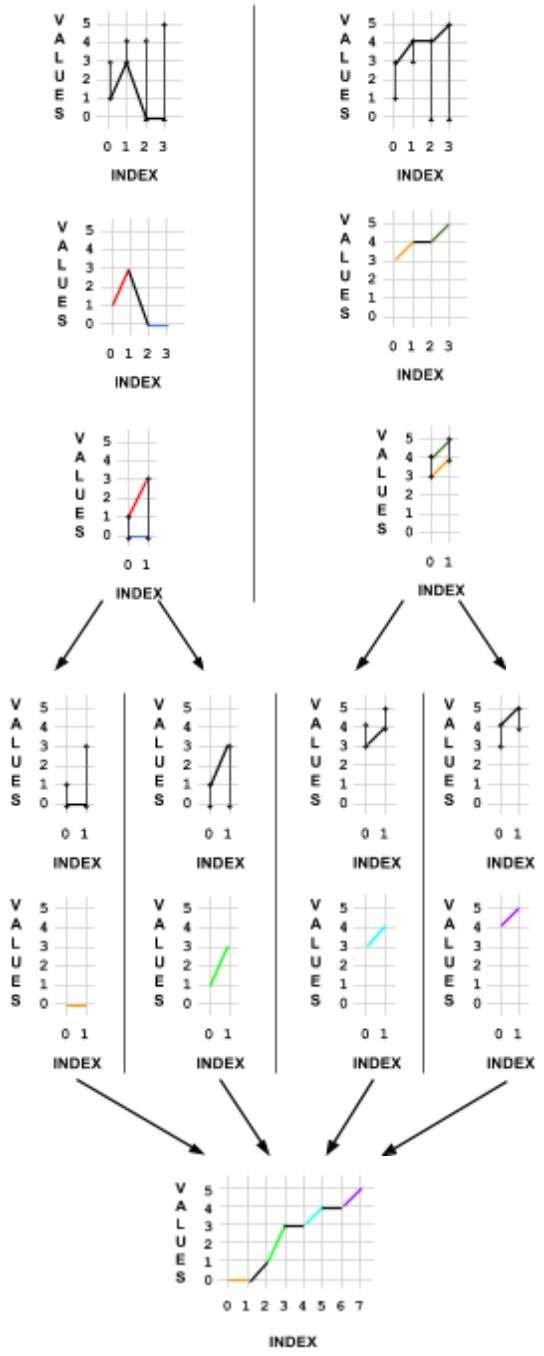
Then, do a compare-and-swap for each of the overlapping pairs. All the smaller elements should get moved to the first sequence and all the higher elements should get moved to the second sequence...



Notice that…

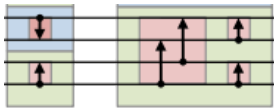
- all values in “smaller elements” are <= the values in “larger elements”.
- both “smaller elements” and “larger elements” are bitonic/monotonic.

Keep doing this overlap+compare-and-swap process for each output until you can’t break up the sequence any further. Ultimately you’ll end up with the input being fully sorted...



The example above showed how a bitonic sequence gets sorted. However, in the actual sorting network for a bitonic sort, your input sequence does not have to be bitonic. The sorting network will make it bitonic.

For example, here's a bitonic sorting network with 4 inputs…



The arrows define the direction of the compare-and-swap. Any inputs going through a …

- blue box will have descending compare-and-swaps (always have down arrows).
- green box will have ascending compare-and-swaps (always have up arrows).

Notice what's happening in the stacks…

The 1st stack...

- compare-and-swap(0, 1, desc)
- compare-and-swap(2, 3, asc)

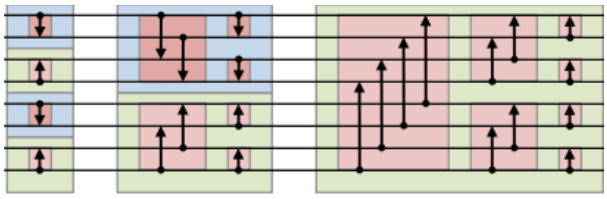
We end up with 1 bitonic sequence after the 1st stack: 0-3.

The 2nd stack…

- perform asc bitonic sort on 0-3

We end up with a fully sorted sequence after the 2nd stack.

The network scales out easily. Here's the network extended to 8 inputs…



The 1st stack…

- compare-and-swap(0, 1, desc)
- compare-and-swap(2, 3, asc)
- compare-and-swap(4, 5, desc)
- compare-and-swap(6, 7, asc)

We end up with 2 bitonic sequences after the 1st stack: 0-3 and 4-7.

The 2nd stack…

- perform desc bitonic sort on 0-3
- perform asc bitonic sort on 5-7

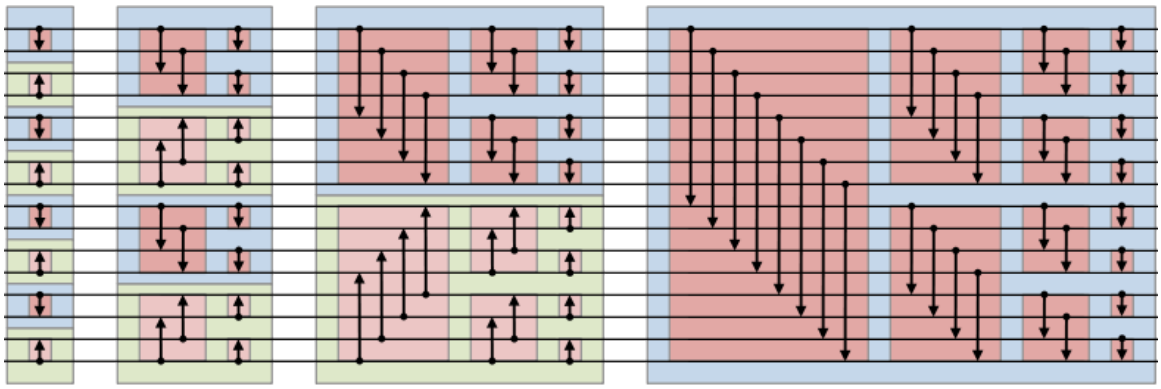
We end up with 1 bitonic sequences after the 2nd stack: 0-7.

The 3rd stack…

- perform asc bitonic sort on 0-7

We end up with a fully sorted sequence after the 3rd stack.

Here's the network extended to 16 inputs. Process is pretty much the same, but this time the last stack is switched to perform a sort in descending order (if you want an ascending sort you can switch this back)...



> **NOTE**: The block diagram came from wikipedia. The wikipedia article shows another version of the block diagram where ascending/descending doesn't matter, but I didn't have the time to comprehend how it worked so I left it out.

> **NOTE**: In each of the block diagrams, for the 1st stack I mention compare-and-swap. Be aware that a compare-and-swap on to adjacent elements is essentially the same thing as doing a bitonic sort on an input of size 2. This is important because it makes the algorithm simpler to implement.

Serial

Here are 2 serial examples of bitonic sort.

The first example is a hardcoded sorting network that will sort any input of size 4. It uses the same compare-and-swaps shown in the 4 input example in the parent section.

Python example...

```
def compare_and_swap(in_data, idx1, idx2):
    if in_data[idx1] < in_data[idx2]:
        temp = in_data[idx1]
        in_data[idx1] = in_data[idx2]
        in_data[idx2] = temp


data = [7,4,6,5]  # len must be power of 2
compare_and_swap(data, 0, 1)
```

```
compare_and_swap(data, 3, 2)

compare_and_swap(data, 0, 2)
compare_and_swap(data, 1, 3)

compare_and_swap(data, 0, 1)
compare_and_swap(data, 2, 3)

print(str(data))
```

Output is…

```
[7, 6, 5, 4]
```

The second example is the same pretty much the same thing as the first example, except that it's been generalized into an algorithm. It will take in any input (length must be power of 2) and apply the necessary compare-and-swaps for a bitonic sort.

Note that the methods in this example map to the diagram shown in the parent section…

- bitonic_swap_step does the computations shown as red blocks
- bitonic_swap_sweep does the computations shown in the blue/green blocks
- bitonic_sort does the entire diagram

Python example...

```
from enum import Enum


class Direction(Enum):
    ASC = 0
    DESC = 1

    def flip(self):
        if self.value == 0:
            return Direction.DESC
        else:
            return Direction.ASC


def compare_and_swap(in_data, idx1, idx2):
    if in_data[idx1] < in_data[idx2]:
```

```
        temp = in_data[idx1]
        in_data[idx1] = in_data[idx2]
        in_data[idx2] = temp


def bitonic_swap_step(in_data, offset, size, direction):
    swap_count = size // 2

    if direction == Direction.DESC:
        elem_ptr1 = offset
        elem_ptr2 = elem_ptr1 + swap_count
        elem_ptr_inc = 1
    else:
        elem_ptr1 = offset + size - 1
        elem_ptr2 = offset + size - 1 - swap_count
        elem_ptr_inc = -1

    for i in range(swap_count):
        compare_and_swap(in_data, elem_ptr1, elem_ptr2)
        elem_ptr1 += elem_ptr_inc
        elem_ptr2 += elem_ptr_inc


def bitonic_swap_sweep(in_data, offset, size, direction):
    swap_size = size
    while swap_size > 0:
        swap_start = offset
        while swap_start < offset + size:
            bitonic_swap_step(
                in_data,
                swap_start,
                swap_size,
                direction)
            swap_start += swap_size
        swap_size //= 2


def bitonic_sort(in_data, direction):
    step_size = 2

    while step_size <= len(in_data):
        for step_offset in range(0, len(in_data), step_size):
```

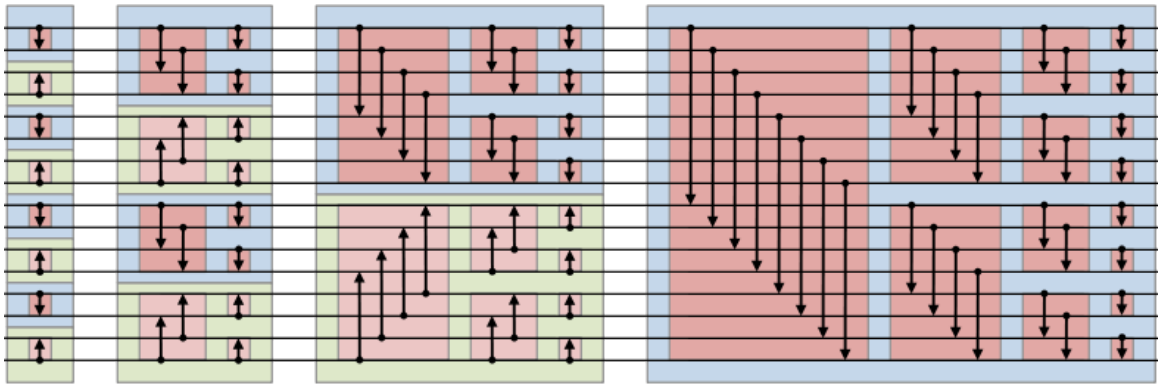
```
            bitonic_swap_sweep(
                in_data,
                step_offset,
                step_size,
                direction)
            direction = direction.flip()
        step_size *= 2


data = [7,4,6,5]  # len must be power of 2
bitonic_sort(data, direction=Direction.DESC)
print(str(data))
```

Output is…

```
[7, 6, 5, 4]
```

Parallel

The block diagram for 16 input bitonic sort network is below. Notice how each vertical row of red boxes is essentially 1 parallel step (each line is being touched by only 1 compare-and-swap).



We can see that for an input size of 16, we have 10 parallel steps. We can generalize that to O(logn) step complexity.

At each parallel step, we can see that we're doing 8 compare-and-swaps. That's n/2 compare-and-swaps. We can generalize that as O(n) work complexity per step. Since our step complexity is O(logn), our overall work complexity will be O(logn*n) → O(nlogn).

Like brick/odd-even sort, the fact that it is a sorting network means that it fits well with one of the main tenets of writing code for the GPU: low thread divergence. You may not want to use bitonic sort for a large overall sort, but it may be good to use for sorting a local list in shared memory. If

you choose to do this, remember that you can use thread barriers to sync between the threads in your block vs launching a new kernel for each step.

> **NOTE**: The lessons recommended that if you want to use as a sort network to sort an array in shared memory, it's better to have 1 thread per input. A compare-and-swap will take place on 2 threads, but both those threads will generate the same results. The thread doesn't end after a parallel step. Use a thread barrier to wait until all threads are done for that parallel step before moving to the next parallel step.

For the parallel implementation, instead of trying to write a generic algorithm, the indices of the sort network are hardcoded at each step. I think this is how sorting networks are intended to be written for GPUs. If implemented as a generic algorithm, I'd imagine that we would have much higher thread divergence?

> **NOTE**: Sort networks are suppose to be efficient because they're "oblivious," so we should hardcode the indices at each step? See the sort network section if you need a refresher on sort networks. For this implementation, I took the 2nd example from the serial version and changed it to dump out the indices for each compare-and-swap, and used those indices for the hardcoded version.

Python example…

```python
from numba import cuda
import numpy as np


@cuda.jit('void(int32[:], int32, int32)', device=True, inline=True)
def compare_and_swap(in_data, idx1, idx2):
    if in_data[idx1] > in_data[idx2]:
        temp = in_data[idx1]
        in_data[idx1] = in_data[idx2]
        in_data[idx2] = temp


@cuda.jit
def sort_network_pass(in_data, net_data):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    idx1 = net_data[0][idx]
    idx2 = net_data[1][idx]
    compare_and_swap(in_data, idx1, idx2)
```

```
step1indices = np.array([[0,3,4,7], [1,2,5,6]], dtype=np.int32)
step2indices = np.array([[0,1,7,6], [2,3,5,4]], dtype=np.int32)
step3indices = np.array([[0,2,5,7], [1,3,4,6]], dtype=np.int32)
step4indices = np.array([[0,1,2,3], [4,5,6,7]], dtype=np.int32)
step5indices = np.array([[0,1,4,5], [2,3,6,7]], dtype=np.int32)
step6indices = np.array([[0,2,4,6], [1,3,5,7]], dtype=np.int32)

data = np.array([7,4,6,5,3,2,1,0], dtype=np.int32)
sort_network_pass[1,4](data, step1indices)
sort_network_pass[1,4](data, step2indices)
sort_network_pass[1,4](data, step3indices)
sort_network_pass[1,4](data, step4indices)
sort_network_pass[1,4](data, step5indices)
sort_network_pass[1,4](data, step6indices)

print(str(data))
```

Output is…

```
[0 1 2 3 4 5 6 7]
```

Merge Sort

> **NOTE**: Merge sort is an efficient sort to do on a GPU, but confusing to implement correctly. If you can, opt for using radix sort -- it's a much simpler sort to implement and works very well on the GPU.

Merge sort is a sort that, given enough contortions to the logic, can also be done in parallel on the GPU. The trick to making merge sort work efficiently on GPUs is to perform a different algorithm at each "tier" of data.

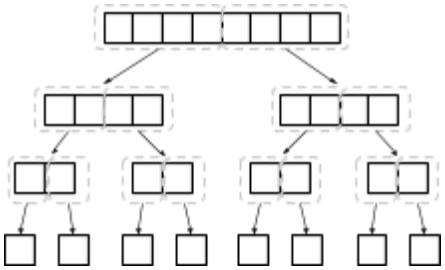
That means that you choose a different algorithm when the lists being merged are small vs when they're medium sized vs when they're large. These different algorithms make sure that the threads being launched are kept busy / doing lots of work, so we aren't wasting GPU resources.

This will be discussed further in the parallel subsection.

Serial

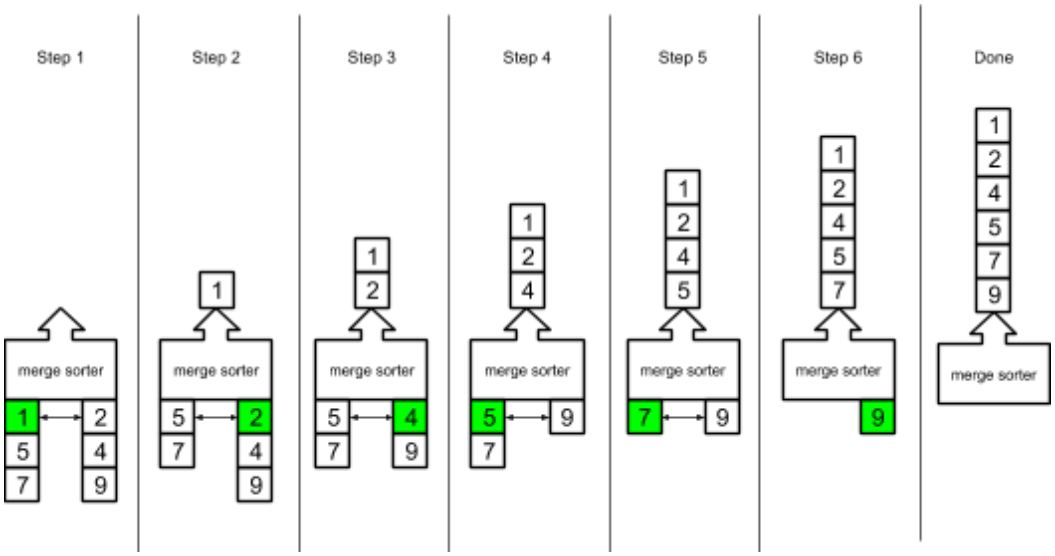
If you forgotten how merge sort works, the core of the sort revolves around merging together 2 sorted lists.

The input array is recursively broken down until each sublist is of size 1…



NOTE: Remember that a list of size 1 is implicitly sorted. There is only 1 element.

Then, those sublists are built back up as you walk out of the recursion by merging the lists. Merging in this case means that the head element of both sorted lists are compared and the smallest/largest one is taken. This is done iteratively until the lists are fully merged...



Python example…

```
def sort(data):
    if len(data) <= 1:
        return data

    split_idx = len(data)//2
    half1 = data[:split_idx]
    half2 = data[split_idx:]

    half1_sorted = sort(half1)
    half2_sorted = sort(half2)

    ret = []
    half1_idx = 0
```

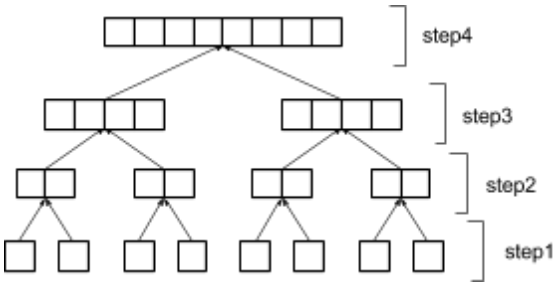
```
    half2_idx = 0
    while half1_idx < len(half1_sorted) or half2_idx < len(half2_sorted):
        if half1_idx < len(half1_sorted) and half2_idx < len(half2_sorted):
            val1 = half1_sorted[half1_idx]
            val2 = half2_sorted[half2_idx]
            if val1 < val2:
                val = val1
                half1_idx += 1
            else:
                val = val2
                half2_idx += 1
        elif half1_idx < len(half1_sorted):
            val = half1_sorted[half1_idx]
            half1_idx += 1
        elif half2_idx < len(half2_sorted):
            val = half2_sorted[half2_idx]
            half2_idx += 1

        ret.append(val)

    return ret


out = sort([5,1,2,4,3,3,2,0,9])
print(str(out))
```

Output is…

```
[0, 1, 2, 2, 3, 3, 4, 5, 9]
```

Parallel

The naive way to do merge sort in parallel would be to pretty much do the same thing as the serial version, except that serial merge operations at each level would happen in parallel…



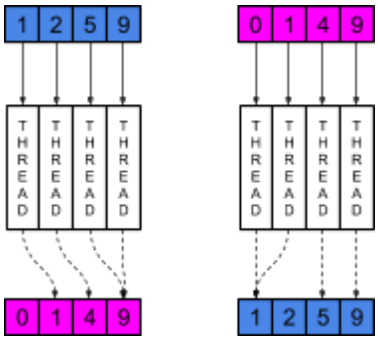
Apparently, doing it this way is a poor match for the GPU (launching too many kernels? thread divergency issues? etc..). A better way to merge 2 sorted lists together on the GPU would be to launch a thread for each element of each list, where each thread will...

1. determine its elements position in its own list
2. determine where its element should be in the other list (via binary search)
3. scatter its value to ownIdx + otherIdx

   NOTE: You almost always want to do these steps in shared memory, which means you can't deal with merging large arrays. A strategy for merging large arrays is discussed further below.

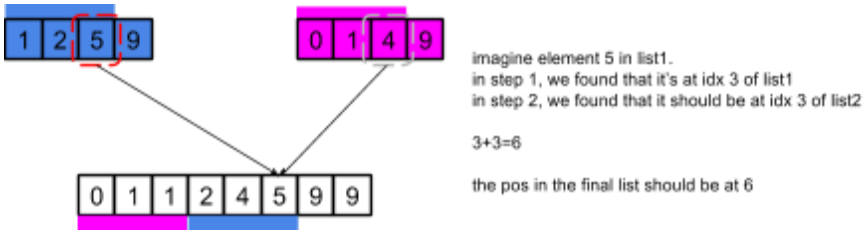
The first step is pretty basic. You can use threadIdx to determine your position in your own list.

The second step is a bit more tricky. You basically need to find where your element would be in the other list. Typically, the quickest way to do this is by running a binary search through the other list…



NOTE: Think of how Collections.binarySearch() works in Java -- if it can't find it, it'll return the index of where it would be if it existed.

The third step determines the index of the element in the final list and scatters the element to that location. You take where you are in your own list (step1 result) and where you should be in the second list (step2 result) and add them together to find out where to place the element in the final list…



NOTE: This works because both lists are sorted. If you're having trouble thinking about it, just play through some small examples in your head.

NOTE: The diagram above is not entirely accurate. Note that the element 1 exists in both lists. It looks like if the same value exists in both lists, they will get scattered to the same location. Maybe check to see if you got an exact hit in the binary search and if so, just move over some extra spaces if you're in list1 vs list2?

Python example…

```
from numba import cuda
import numpy as np


@cuda.jit('int32(int32[:], int32)', device=True)
def search(in_data, val):
    low = 0
    high = len(in_data) - 1
    while low <= high:
        mid = (low + high) // 2
        if in_data[mid] == val:
            return mid
        elif in_data[mid] < val:
            low = mid + 1
        else:
            high = mid - 1

    return mid if in_data[mid] > val else mid+1


# this example uses global memory...
# you SHOULD be using shared memory when you do this, otherwise it
# is inefficient.
@cuda.jit
def merge(in_data1, in_data2, out_data):
    pos = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    len_ = len(in_data1)

    if pos >= len_:
        pos -= len_
        temp = in_data1
        in_data1 = in_data2
        in_data2 = temp

    idx1 = pos
```

```
    val1 = in_data1[idx1]

    idx2 = search(in_data2, val1)

    out_data[idx1 + idx2] = val1


# run gpu merge on 2 sorted lists
in1 = np.array([1, 2, 5, 9], dtype=np.int32)
in2 = np.array([0, 4, 7, 8], dtype=np.int32)
out = np.zeros(8, dtype=np.int32)
merge[1,8](in1, in2, out)
print(str(out))
```

Output is...

```
[0 1 2 4 5 7 8 9]
```

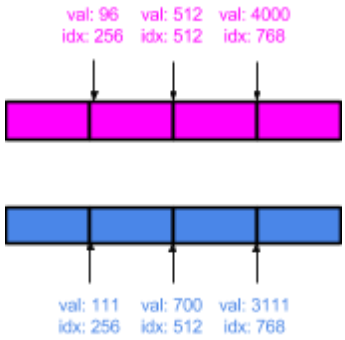
There are a couple of things to watch out for here...

It's typical to replace the early steps of our merge sort with a sorting network. So, instead of running merges on lists of a small size (e.g. 1, 2, 4, 8, …), we can have each thread load up 1024 element chunks of the input data into shared memory and use a sorting network to get those chunks sorted. Then, we can start off the actual merge sort threads with 1024 element lists.

> **NOTE**: Remember that sorting networks are super efficient when implemented on a GPU and when the data being sorted is loaded into shared memory. See the sorting network section for more info.

In the latter steps of our merge sort, if we're working only with shared memory we won't be able to merge larger lists. It turns out that there's a way to split up the larger lists and distribute chunks for merging to different thread blocks such that everything comes out properly sorted in the end…

1. take both lists and split them up by n (e.g. 256)



2. take the values at each split point and sort them

3. we can use the splitter list to determine which block a merge should go to.

   For example, imagine if we wanted to merge the 2nd and 3rd block designated by the splitter list. We use binary search to calculate where 2nd's value (512) would be in 3rd block and vice versa. The range defined would be what would get merged by an independent thread block.

   NOTE: I don't fully understand how this works. The lesson isn't clear. Need to try implementing at some point.

Because there may be many things happening here if we implemented it properly for the GPU, it's tough to get an overall work/step complexity.

Radix Sort

Radix sort is the most efficient sorting algorithm when it comes to sorting on the GPU. A large part of the reason for this is that it's simple and brute-force, just like sorting networkings. Unlike sorting networks, you can use radix for sorting in global memory (super larger arrays).

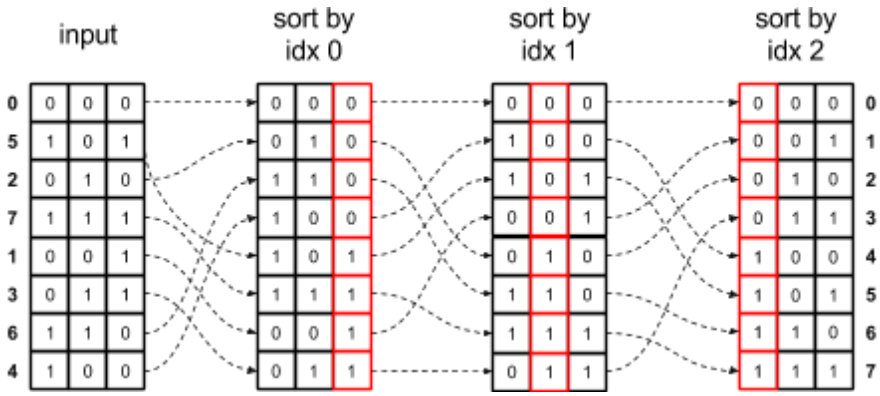
One thing to keep in mind about radix sort is that it requires keys. The other sorts in previous sections of this document all worked around the idea of comparison. Radix sort works around the idea of having actual data for the key (e.g. integers), where pieces of this data (e.g. bits in the integer) get split up and compared.

Serial

If you forgot how a radix sort works, you basically break up the key you're sorting by into symbols. Starting from the least significant symbol, you sort the list by that symbol. Then, move up to the next symbol and repeat. Keep going until there are no more symbols.

What is a symbol? You can think of a symbol as the components that make up the key you're sorting by. So for example, your key can be of type integer and the symbols you're sorting by can be the individual bits that make up those integers. Another example is having a key of type string and the symbols you're sorting by can be the letters/numbers/punctuation that make up the string.

Here's an example of sorting a 3-bit integer...



NOTE: Remember that when you're sorting by symbol to maintain the order in which each element appears. For example, if you're sorting by idx 0, all the 0-bit items should appear in the same order (the 1st 0-bit item in the original list with should get moved to the 1st element, the 2nd 0-bit item in the original list should get moved to the 2nd element, etc…).

Python example…

```
def sort(data, bit_pos):
    ret0 = []
    ret1 = []

    bit_mask = 1 << bit_pos
    for val in data:
        val_bit_isolate = val & bit_mask

        if val_bit_isolate == 0:
            ret0.append(val)
```

```
        else:
            ret1.append(val)

    return ret0+ret1


def debug_print(data):
    for i in range(len(data[::])):
        print("{0:03b}={0}".format(data[i]))
    print("------")


print('input')
data = [0, 5, 2, 7, 1, 3, 6, 4]
debug_print(data)

print('bit pos 0')
data = sort(data, 0)
debug_print(data)

print('bit pos 1')
data = sort(data, 1)
debug_print(data)

print('bit pos 2')
data = sort(data, 2)
debug_print(data)
```

Output is…

```
input
000=0
101=5
010=2
111=7
001=1
011=3
110=6
100=4
------
bit pos 0
000=0
```

```
010=2
110=6
100=4
101=5
111=7
001=1
011=3
------
bit pos 1
000=0
100=4
101=5
001=1
010=2
110=6
111=7
011=3
------
bit pos 2
000=0
001=1
010=2
011=3
100=4
101=5
110=6
111=7
------
```

Parallel

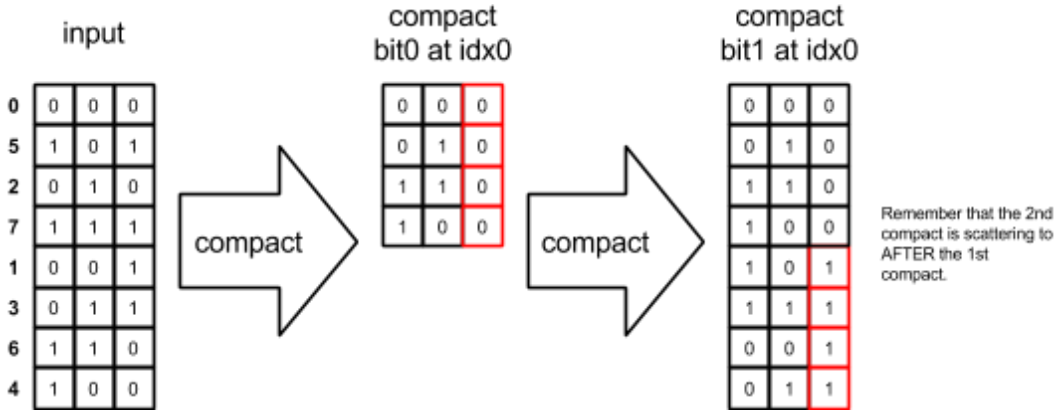
The parallel implementation of radix sort involves boils down to doing multiple compact operations per symbol. For example, imagine you're dealing with bits. Starting at the least significant bit, for each bit…

1. compact where bit is 0
2. compact where bit is 1, but append the results to step 1's output

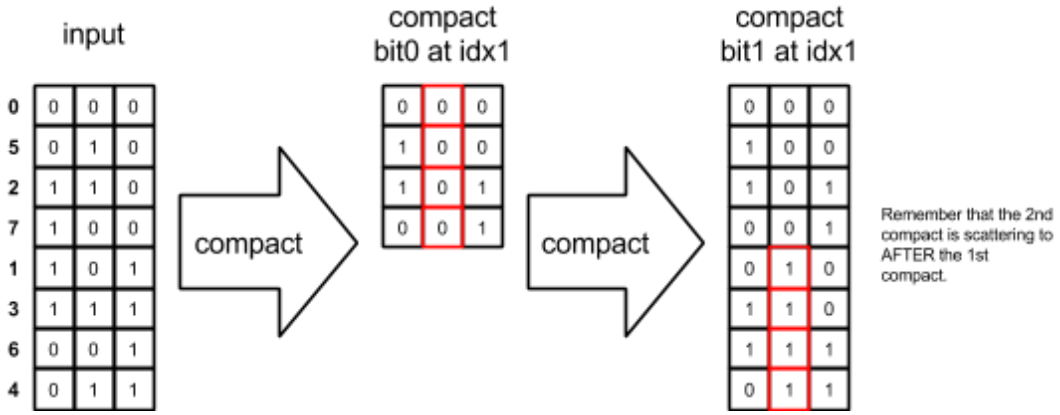
> **NOTE**: Typically, for radix sort on the GPU, you should group by a larger symbol size. For example, instead of sorting by 1 bit at a time, it's typically to sort by 4 bits at a time if you're sorting on a GPU.

For example, imagine you're sorting a list of 3-bit integers and your symbol size is 1 bit...

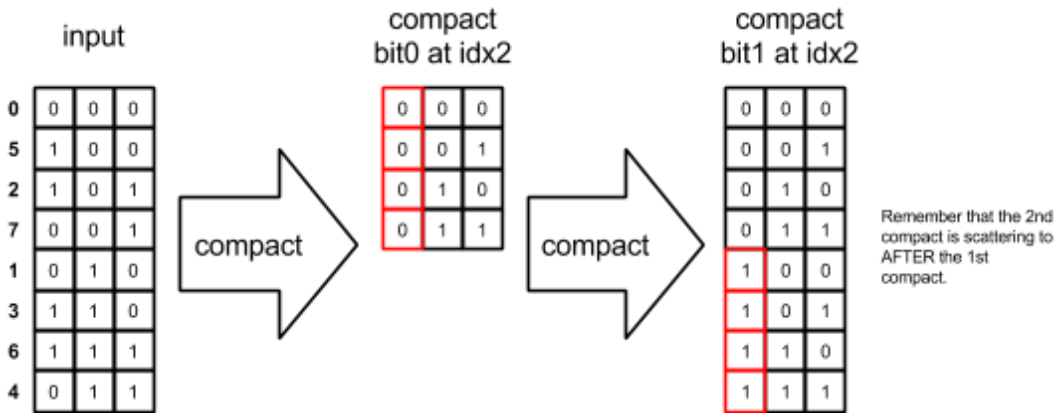
Begin by doing the compacts for the least significant bit...



Follow up by doing the compacts for the 2nd bit…



Then finally the 3rd bit…



Python example…

```
import numpy as np
```

```
from numba import cuda


@cuda.jit
def filter(in_data, out_data, bit_idx, bit_val):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    bit = (in_data[idx] >> bit_idx) & 0x01
    if bit == bit_val:
        out_data[idx] = 1
    else:
        out_data[idx] = 0


@cuda.jit
def hillissteele_scan_addop(in_data, out_data, step):
    skip = 2**step

    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if idx < skip:
        out_data[idx] = in_data[idx]
    else:
        out_data[idx] = in_data[idx] + in_data[idx-skip]


@cuda.jit
def inc_scan_to_exc_scan(in_data, out_data, identity):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if idx == 0:
        out_data[0] = identity
    else:
        out_data[idx] = in_data[idx-1]


@cuda.jit
def scatter(in_data, out_data, filters, offsets, start_from):
    idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    if filters[idx] == 0:
        return

    offset = start_from + offsets[idx]
    out_data[offset] = in_data[idx]
```

```
def run_predicate(in_data, bit_idx, bit_val):
    out_data = np.zeros(10, dtype=np.int32)
    filter[1, 10](in_data, out_data, bit_idx, bit_val)

    print("filter" + str(bit_val) + ":  " + str(out_data))

    return out_data


def run_scan(pred, bit_val):
    # run scan on filters using hillissteele scan (inc. scan)
    inc_scan_out_data = np.copy(pred)  # copy filter out
    hillissteele_scan_addop[1, 10](inc_scan_out_data, inc_scan_out_data, 0)
    hillissteele_scan_addop[1, 10](inc_scan_out_data, inc_scan_out_data, 1)
    hillissteele_scan_addop[1, 10](inc_scan_out_data, inc_scan_out_data, 2)
    hillissteele_scan_addop[1, 10](inc_scan_out_data, inc_scan_out_data, 3)

    # convert inc. scan to exc. scan (required by compact)
    exc_scan_out_data = np.zeros(10, dtype=np.int32)
    inc_scan_to_exc_scan[1, 10](inc_scan_out_data, exc_scan_out_data, 0)
    print("scan" + str(bit_val) + ":    " + str(exc_scan_out_data))
    print("scanlen" + str(bit_val) + ": " + str(inc_scan_out_data[-1]))

    # return exc. scan and the length of the output
    return exc_scan_out_data, inc_scan_out_data[-1]


def run_scatter(out_data, pred, scan, start_at, bit_val):

    scatter[1, 10](in_data,
                   out_data,
                   pred,
                   scan,
                   start_at)
    print("scatter" + str(bit_val) + ": " + str(out_data))


# generate arrays
bit_size = 3
max_val = 1 << bit_size
```

```
np.random.seed(0)
in_data = np.random.randint(0, max_val, 10)
out_data = np.full(10, -1, dtype=np.int32)

for bit_idx in range(bit_size):
    print("----sorting bit_idx " + str(bit_idx))
    print("input:    " + str(in_data))

    # run predicate on input
    bit0_pred = run_predicate(in_data, bit_idx, 0)
    bit1_pred = run_predicate(in_data, bit_idx, 1)

    # run scan on filters using hillissteele scan (inc. scan)
    bit0_scan, bit0_scanlen = run_scan(bit0_pred, 0)
    bit1_scan, bit1_scanlen = run_scan(bit1_pred, 1)

    # run scatter
    out_data[:] = -1  # clear out data
    run_scatter(out_data, bit0_pred, bit0_scan, 0, 0)
    run_scatter(out_data, bit1_pred, bit1_scan, bit0_scanlen, 1)

    print("output:   " + str(out_data))
    in_data[:] = out_data
```

Output is…

```
----sorting bit_idx 0
input:    [4 7 5 0 3 3 3 7 1 3]
filter0:  [1 0 0 1 0 0 0 0 0 0]
filter1:  [0 1 1 0 1 1 1 1 1 1]
scan0:    [0 1 1 1 2 2 2 2 2 2]
scanlen0: 2
scan1:    [0 0 1 2 2 3 4 5 6 7]
scanlen1: 8
scatter0: [ 4  0 -1 -1 -1 -1 -1 -1 -1 -1]
scatter1: [4 0 7 5 3 3 3 7 1 3]
output:   [4 0 7 5 3 3 3 7 1 3]

----sorting bit_idx 1
input:    [4 0 7 5 3 3 3 7 1 3]
filter0:  [1 1 0 1 0 0 0 0 1 0]
```

```
filter1:  [0 0 1 0 1 1 1 1 0 1]
scan0:    [0 1 2 2 3 3 3 3 3 4]
scanlen0: 4
scan1:    [0 0 0 1 1 2 3 4 5 5]
scanlen1: 6
scatter0: [ 4  0  5  1 -1 -1 -1 -1 -1 -1]
scatter1: [4 0 5 1 7 3 3 3 7 3]
output:   [4 0 5 1 7 3 3 3 7 3]

----sorting bit_idx 2
input:    [4 0 5 1 7 3 3 3 7 3]
filter0:  [0 1 0 1 0 1 1 1 0 1]
filter1:  [1 0 1 0 1 0 0 0 1 0]
scan0:    [0 0 1 1 2 2 3 4 5 5]
scanlen0: 6
scan1:    [0 1 1 2 2 3 3 3 3 4]
scanlen1: 4
scatter0: [ 0  1  3  3  3  3 -1 -1 -1 -1]
scatter1: [0 1 3 3 3 3 4 5 7 7]
output:   [0 1 3 3 3 3 4 5 7 7]
```

Algorithm Optimization

The following subsections detail optimization patterns for GPU program detailed in Stratton's taxonomy paper: http://impact.crhc.illinois.edu/shared/papers/optimization2012.pdf.

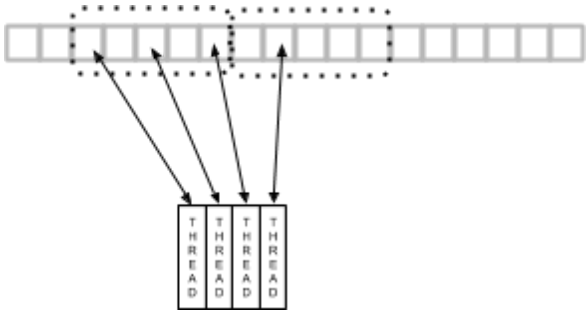
Data Layout Transformation

If the memory access pattern is strided, you can reorganize your input data such that it becomes contiguous. The transpose communication pattern (read the map/transpose section) covers this. Specifically, the part that covers converting an array of structures (AOS) to structure of arrays (SOA)...

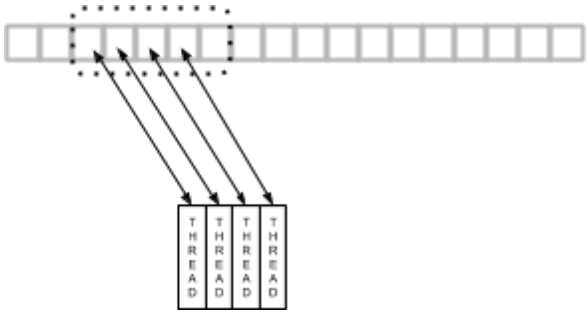
```
// performing an AOS to SOA conversion serially
// doing this for illustration -- way more efficient if done as cuda kernel
struct soa_struct {
    float f;
    int j;
};

soa_struct var1[1024];
```

```
struct aos_struct {
    float f[1024];
    int j[1024]
};

aos_struct var2;

for (int i = 0; i < 1024; i++) {
    var2.f[i] = var1[i].f;
    var2.j[i] = var1[i].j;
}
```

Remember from the thread divergence section that threads execute in lock-step. As such, if all threads are at the point of accessing soa_struct[i].f, it will be a strided access pattern (they'll be skipping over j)...



If aos_struct.f[i] were accessed instead, the memory being accessed by each thread will be closer together (contiguous), so it's much more likely that the threads will hit the same chunk of global memory being pulled in by the GPU. This means less global memory fetches overall…



Tiling

Tiling is a fancy word for manually caching data that'll be worked on by the threads in a block into shared memory. Once the processing is done, the data will be written back out to main memory.

> **NOTE**: I think the reason why this is called tiling is because it takes a “tile” from a larger grid of 2D data into shared memory, processes it in shared memory, and writes it out back to global memory. Since you’re working with tiles, it’ll be reading from and writing to “coalesced” global memory (see optimization section for more info).

Privatization

Privatization is a fancy word for keeping and working on a local output instead of hitting a global output. Once work is complete on the local output, combine it into the global output (if applicable).

> **NOTE**: Take a look at the local histogram example earlier in this doc. You can do mini-histograms local to each thread block, and then combine to a global histogram using atomics.

Partitioning (Binning)

Binning is a fancy word for spatial partitioning.

The easiest example to think of here is calculating the distance from every city to every other city in some map, and only keep those city pairs that are within 500km of each other. The brute force way of doing this would be to pair up every city with every other city and launch a thread for each pair.

Instead of doing that, you can partition the map into 500km x 500km bins. The cities would go into their respective bin and the cities would only pair up with cities within their own bin or an adjacent bin. As such, you would be launching much fewer threads…

> **NOTE**: Doesn’t really need to be said but this isn’t limited to 2D.