

# 1 Background

This document aims to give an introduction to CUDA C programming for students who are new to the language. Provided alongside these notes are fully documented example programs, as well as exercises that usually involve modifying these programs.

Section 2 will describe how to access the CUDA environment that's available on the SECS servers. Section 3 will summarize a few key features of CUDA programming model. Section 4 will show off more advanced optimization techniques available to a CUDA programmer. Finally, Section 5 will present applications of CUDA to several classes of problems that are well-suited for the GPU.

## 2 Setting up a CUDA Environment

This section will describe how to configure a development environment that's suitable for compiling and executing CUDA programs. By default, this assignment will assume the use of the SECS Linux server `yoko`, which is installed with a Tesla K20Xm graphics card. An additional subsection at the end will point to some resources on how to harness a CUDA-enabled GPU on a personal computer.

### 2.1 Accessing the SECS servers

- For the following steps, you will need an SECS account. If you don't already have one, an SECS account can be requested here:  
`https://oakland.edu/secs/technology-office/`
- The SECS servers can be accessed when off-campus with the use of a VPN. SECS provides VPN software that is compatible with Windows, MacOS, and Linux. The official SECS instructions on how to set up the VPN can be found here:  
`http://www.secs.oakland.edu/docs/pdf/vpn.pdf`  
(Local copy saved to `resources/secs-vpn.pdf`)
- Once the VPN is installed, configured, and enabled, the SECS servers can be accessed via SSH from a terminal. Connect to the server `yoko.secs.oakland.edu` using your SECS username and password using the command:

```
ssh USERNAME@yoko.secs.oakland.edu
```

You will be prompted for your SECS password before the connection is accepted.

- Further documentation on the SECS servers, including those which have CUDA-enabled GPUs, can be found here:

<https://www.secs.oakland.edu/docs/pdf/linuxServers.pdf>

(Local copy saved to `resources/secs-linuxServers.pdf`)

- If you already had files saved on the SECS network, you can run the `ls` command to view them after connecting to the server.

## 2.2 The CUDA environment

The primary tool when developing a CUDA program is the Nvidia CUDA compiler, `nvcc`. On the yoko server, `nvcc` should have already been made available by default, which you can verify by running `which nvcc`. If successful, this command will show the location of `nvcc` on the filesystem.

`nvcc` is a compiler for the language *CUDA C/C++*, which is a *superset* of both the standard C and C++ languages. That is, `nvcc` is still able to compile any C or C++ program that a normal compiler for those respective languages could. The remainder of this assignment will only use CUDA C.

- As a first test of `nvcc`, run the command `nano hello-world.c` to open the terminal's text editor and enter the following simple C program:

```
#include <stdio.h>

void main() {
    printf("Hello world!\n");
}
```

- Save and close the file. (Recall that in the `nano` text editor, `Ctrl+X` attempts to close the current file, after first prompting you to save the file if it has changed.)
- Compile the program with `nvcc hello-world.c -o hello-world`. The command-line argument `-o hello-world` tells the compiler to write the executable output to the file `hello-world`. If the `-o` flag isn't given, `nvcc` will write the executable to the file `a.out` by default.
- Run the executable with the command `./hello-world`. You should see the output `Hello world!`, just as you would with when using a normal C compiler.

## 2.3 Installing CUDA locally

If you have a relatively new Nvidia GPU installed in a personal computer, it's likely CUDA-enabled, meaning you would be able to follow these assignments without needing to use the remote SECS server. This will require substantially more configuration than is necessary on an SECS server, but may be worthwhile for an interested student who plans to use CUDA beyond the scope of this class.

In order to follow the remainder of the assignment on a local environment, you will need to install the CUDA Toolkit. Instructions are provided by Nvidia and will depend on your operating system.

- A list of CUDA-enabled GPUs is provided by Nvidia here:  
<https://developer.nvidia.com/cuda-gpus>
- CUDA installation instructions for Windows:  
<https://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html>
- CUDA installation instructions for Linux:  
<https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html>

Once the CUDA Toolkit is installed, add the `bin` subdirectory from the installation directory to your `PATH` variable. This will make the newly installed CUDA executables available from the terminal. If this has been done successfully, the terminal command `which nvcc` should correctly display the location of the CUDA compiler.

## 3 The CUDA Programming Model

The goal of the CUDA programming model is to provide a high-level interface to the highly parallelizable computational power of a GPU. This section will summarize the underlying structure of each CUDA program by example, as well as give a method of profiling CUDA programs that will be used to compare them to their serial counterparts.

### 3.1 Thread hierarchy

A function that is to be executed on the GPU is generally called a *kernel*. When a CUDA program calls a kernel function, the program must also allocate that kernel a certain number of threads. Each thread will then execute its own copy of that function, either in parallel or in sequence, according to the kernel's definition.

The threads assigned to a kernel are divided into blocks, where each block has the same number of threads. Thus, each thread is associated to both a block index, as well as a thread index within that block. This data can be accessed by device code using the following CUDA-specific global variables:

- `blockDim.x` contains the number of threads in each block.
- `blockIdx.x` contains the executing thread's block number.
- `threadIdx.x` contains the executing thread's index within its block.

If  $N$  threads are allocated, then each thread can use the above three values to compute its own globally unique *thread ID* (written `tid` in this assignment) between 0 and  $N - 1$ . This pattern is used frequently in CUDA programs:

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

**Program 1.** Compile and execute `print-tid.cu` in your CUDA environment. You should get output similar to the following:

```
nfireman@yoko:~/cuda$ nvcc print-tid.cu -o print-tid
nfireman@yoko:~/cuda$ ./print-tid
block index=1, thread index=0, tid=3
block index=1, thread index=1, tid=4
block index=1, thread index=2, tid=5
block index=0, thread index=0, tid=0
block index=0, thread index=1, tid=1
block index=0, thread index=2, tid=2
```

There are several language features of CUDA C to note about this program:

- The `__global__` keyword indicates that the marked function executes on the device but is called from the host (in other words, the marked function is a kernel function). A similar keyword, `__device__`, means the marked function both executes on and must be called by the device.
- The triple angle bracket notation seen in `print_tid<<2,3>>()` is known as a *kernel launch*. Calling a kernel function in this way from the host means that 2 thread blocks that contain 3 threads will each call that kernel function (allocating 6 threads altogether).
- The `cudaDeviceSynchronize()` function blocks the program's process until each device thread has finished executing. Without this synchronization call, the host process may terminate before one or more threads has a chance to print its statement to the console.
- Finally, note that each `tid` value printed to the console is unique and between 0 and 5, as expected.

**Exercise 1.** Change the `print_tid` kernel launch to use 4 blocks of 3 threads each, recompile and execute the program, and verify that the thread ID values are again unique from 0 to 11.

## 3.2 Memory model

In the context of a CUDA program, the CPU is known as the *host* and the CUDA-enabled GPU is known as the *device*. The host cannot directly access the device's memory (and vice versa); the memory of one must be copied to the memory of the other by calling a CUDA library function. Therefore, a CUDA program will often broadly consist of the following steps:

- Program 2.** The program `parallel-add.cu` illustrates these steps in a minimal CUDA program. Run and compile the `parallel-add.cu` program and compare your observed output to the following:

The program allocates `N` threads on a single block, and performs one addition per thread in the kernel function `kernel_add`. To verify that the program is working correctly, it prints the resulting sums after copying them from the device back to the host.

- The function `cudaMalloc` is the standard way to allocate device memory. Consider the call:

- The second argument `N * sizeof(int)` denotes the number of bytes of device memory to allocate.
- The value `&dev_a` is the address of the pointer `dev_a`. Its address is passed instead of the value itself since `cudaMalloc` will write the pointer to allocated device memory into the address of `dev_a`, changing its value. This is known as a *pass by reference*.
- Since `dev_a` is of type `int*`, `&dev_a` will be of type `int**`, as `&dev_a` is a pointer to `dev_a`. However, since `cudaMalloc` is written to allocate any amount (that is, any type) of memory, it expects its pointer argument to be of type `void**`, which signifies a pointer to *typeless* memory. To accommodate this, `&dev_a` must be cast to `void**`.
- Once `dev_a` has been set to a device memory pointer by `cudaMalloc`, it is an error to attempt to read its value on the host. In order to read device memory from the host, a suitable function from the CUDA API must be used.

- 5



- The type of a CUDA event is `cudaEvent_t`.
- Each CUDA event must be *created* by passing it to `cudaEventCreate` before it can be used.
- Passing an event to the function `cudaEventRecord` saves a timestamp in that event. This is used both before and after the kernel is executed to save timestamps in the events `start` and `stop`, respectively.
- The call `cudaEventSynchronize(stop)` blocks the host until the `stop` event has been recorded (that is, passed to `cudaEventRecord`). This serves the same purpose as `cudaDeviceSynchronize()`, which was used previously.
- The duration between the two recorded events is computed with the call `cudaEventElapsedTime(&milliseconds, start, stop)`. Note that the computed duration is returned by reference using the first `float` argument, rather than as the return value of the function call itself.
- The function `cudaEventDestroy` is used to free an event's memory after the event isn't needed anymore. While this is an important step in large, real-world programs to avoid memory leaks, in simple example programs like this one it's ultimately unnecessary.

**Exercise 3.** Note that the addition in the kernel function of `parallel-add-timed` is nested in a for loop, where the number of loop iterations is equal to `TRIALS`. By default, `TRIALS` is set to 1. Recompile and execute the program again where `TRIALS` is set to 10, 1,000, and 100,000, and record the resulting duration in each case. Does the duration increase at the same rate as the number of trials?

**Exercise 4.** The program `serial-add-timed.c` has been provided to perform a comparable task as `parallel-add-timed.cu`, except without the use of CUDA. Use the command

```
gcc serial-add-timed.c -o serial-add-timed
```

to compile this program with the `gcc` C compiler. Compare its execution time to that of `parallel-add-timed.cu` where `TRIALS` is set to 10, 1,000, and 100,000.

**Exercise 5.** The constant `N` in both `parallel-add-timed.cu` and `serial-add-timed.c` determines the length of the arrays being added. Compare the execution times after changing the value of `N` in both programs to 10,000, 100,000, and 1,000,000 (while `TRIALS` is still set to 1, as in the original programs).

**Note:** you should also remove the for loop at the end of the `main` function that prints every element of the result array for large values of `N`.

## 4 Optimization techniques

### 4.1 Shared block memory

As you may know, modern CPU memory is organized in a *memory hierarchy*, which ranks a system's memory by its access speed. Processors can take advantage of even a small amount of fast memory by copying data to it, manipulating the faster memory, then copying results back to the slower, larger memory.

The CUDA platform also adheres to this principle by way of *shared memory*, which is a small amount of fast memory that is shared by the threads in a thread block. At a high level, a thread block's shared memory is located on the same chip as the GPU cores acting as the threads in that block; this is known as *on-chip memory*. For this reason, the shared memory has higher *bandwidth* (rate of transfer) and lower *latency* (time for a transfer to complete) than the *global memory* that we have used until now. In particular, when `cudaMemcpy` is used in the `cudaMemcpyHostToDevice` mode, the host memory is being copied into the device's global memory.

To demonstrate the effectiveness of shared memory, we will compare the execution times of two programs that complete the same task, where one program's kernel uses shared memory and the other uses only global memory. Both programs compute the *dot product* of two arrays `a` and `b` of length `N`, which is defined as:

$$\text{dot\_prod} = (a[0] * b[0]) + (a[1] * b[1]) + \dots + (a[N-1] * b[N-1])$$

Since each of the `N` terms of this sum can be computed independently of each other, this computation can be parallelized in a straightforward yet highly effective way, as follows. Each allocated thread will be assigned certain indices between 0 and `N`. For each such index `i` assigned to it, that thread will compute `a[i] * b[i]`, writing the result to another array at `c[i]`. Once all products have been computed, the elements of the array `c` can be summed by the host to arrive at the desired dot product. This will be the approach used by the kernel that only uses global memory.

To take advantage of shared memory in a second approach, each thread will maintain its own *partial sum* equal to the sum of the `a[i] * b[i]` terms that thread was assigned to compute. Each thread will then write its partial sum into shared memory, and one thread in each block will be assigned the additional task of summing these per-thread partial sums to compute a per-block partial sum, which can finally be written to global memory. The host will sum the block-wide partial sums to arrive at the desired dot product as before.

Note that the main improvement in the latter shared memory approach comes from decreasing the number of writes made to global memory. The first approach needs to write once to global memory for each index of the input arrays, while the second approach only needs to write to global memory once per thread block.

**Program 4.** The above dot product algorithms are implemented in



`dot-product-globalmem.cu` and `dot-product-sharedmem.cu`. The latter program uses shared memory, introducing the following new features of CUDA C:

- The `__shared__` keyword is used in a variable declaration to denote its allocation in shared memory. Thus, the net effect of the statement

```
__shared__ float threadPartials[THREADS_PER_BLOCK];
```

is to allocate one such array for each thread block associated to the kernel launch.

- Since shared memory is shared by *every thread in a thread block*, and is therefore vulnerable to race conditions, should one thread write to a location in shared memory that another thread can possibly read or write from in a nondeterministic order.
- The `__syncthreads()` function is another example of a synchronization barrier, as we have seen previously with `cudaEventSynchronize` and `cudaDeviceSynchronize`. Calling `__syncthreads()` from the device will force the caller's thread to wait until every other thread in its block has also called `__syncthreads()`. Thus, `__syncthreads` is an essential tool when using shared memory to avoid race conditions as described above.

In `dot-product-sharedmem.cu`, the call to `__syncthreads()` is used to wait until the shared memory array `threadPartials` has received every thread's partial sum, at which point it is safe to compute the per-block partial sum.

**Exercise 6.** Currently, `dot-product-sharedmem.cu` computes the dot product of two `float` arrays. Modify the program so that it computes the dot product of two `int` arrays. In the modified program, the kernel function should have the signature:

```
__global__ void kernel_dot(int* a, int* b, int* blockPartials)
```

After modifying the program and confirming that it still correctly computes the dot product, compare its execution time to the original program that used `float` arrays.

#### Additional resources

- Section 3.2.4 (Shared Memory) of the CUDA C Programming Guide:  
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory>  
(Local copy saved to `resources/CUDA_C_Programming_Guide.pdf`)
- A brief NVIDIA blog post introducing the performance benefits of CUDA shared memory:  
<https://developer.nvidia.com/blog/using-shared-memory-cuda-cc>  
(Local copy saved to `resources/Using_Shared_Memory.pdf`)

- Section 5.3 of "CUDA by Example: An Introduction to General-Purpose GPU Programming" by Jason Sanders Edward Kandrot

## 4.2 Atomic operations

In a multithreaded program, an operation on a value is *atomic* if only one thread is allowed access to that value while the atomic operation is in progress. Atomic operations are necessary when many concurrent threads are trying to read or write to the same value. The CUDA environment provides atomic versions of several arithmetic operations that can be used to perform atomic operations in a CUDA program.

To demonstrate the use of atomic operations in CUDA, two programs are provided whose kernel functions compute a *histogram* representing random numerical data. That is, given the input array of integers `values`, both kernels compute a histogram array `histo`, where at the end of the kernel execution, `histo[i]` is equal to the number of times the integer `i` appears in `values`. Histograms are best known as a tool to summarize data in a human-readable way, but they also have algorithmic applications in image processing and computer vision.

To compute the histogram of the input array, each thread iterates over its own share of input values read from `values`. For each input value, the thread must record its presence in the histogram `histo` by performing the following steps:

1. Read the current value of `histo[i]`.
2. Compute the value `histo[i] + 1`.
3. Overwrite `histo[i]` with the value `histo[i] + 1`.

If these steps aren't guaranteed to be atomic, then two threads may read `histo[i]` before the other can write the incremented value. This results in both threads setting the value to `histo[i] + 1`, even though two increments should have taken place. Thus, the histogram kernel needs to be able to perform atomic additions to the integer values currently stored in `histo`.

Atomic addition in CUDA is performed with the call:

```
atomicAdd(int* address, int value)
```

which atomically adds `value` to the integer located at the address `address`. Note that differently-typed versions of `atomicAdd` exist for `float`, `double`, and so on, but they are all named `atomicAdd`.

**Program 5.** The programs `histogram-globalmem.cu` and `histogram-sharedmem.cu` are provided to demonstrate the use of atomic operations in CUDA.

In `histogram-globalmem.cu`, each thread performs an `atomicAdd` operation on an element of the histogram `histo` for each input value that thread processes.

Thus, the number of concurrent `atomicAdd` operations at any one time is limited to the number of entries in `histo`, since each entry supports at most one atomic addition at a time.

The program `histogram-sharedmem.cu` resolves this issue with the aid of shared memory. Each block of threads computes its own per-block histogram `histo_block`, summarizing the values processed by the threads in that block. Each `histo_block` array supports as many concurrent atomic additions as it has entries as before, and there are as many such arrays as there are allocated thread blocks. Thus, the maximum possible concurrency is increased by a factor of the number of allocated blocks when compared to `histogram-globalmem.cu`.

Of course, once a thread block finishes computing its own `histo_block`, it must atomically add those values to the output histogram `histo` in global memory.

**Exercise 7.** Compile and execute `histogram-globalmem.cu` and `histogram-sharedmem.cu` and compare their execution times.

**Exercise 8.** Modify both `histogram-globalmem.cu` and `histogram-sharedmem.cu` to use an input array of size 10,000 by changing the value of `N` in both programs. Compare the resulting execution times of both programs. Next, repeat the same process using the value 1,000,000 for `N`.

**Exercise 9.** In the kernel function of `histogram-sharedmem.cu`, several steps are currently performed by the one thread in each block with `threadIdx.x == 0` (namely, the first step that initializes `histo_block` and the last step that adds `histo_block` values to the output `histo` array in global memory).

Modify the program so that the work done in those steps is evenly divided between the threads in each block, in the same way that the computation of the histogram itself is divided evenly between all threads in each block. Compare the execution time of your modified program to the original version of `histogram-sharedmem.cu`.

### Additional resources

- Section B.14 (Atomic Operations) of the CUDA C Programming Guide:  
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>  
(Local copy saved to `resources/CUDA_C_Programming_Guide.pdf`)
- Chapter 9 of "CUDA by Example: An Introduction to General-Purpose GPU Programming" by Jason Sanders, Edward Kandrot.
- Chapter 9 of "Programming Massively Parallel Processors: A Hands-on Approach", 3rd Edition by David B. Kirk, Wen-mei W. Hwu.

### 4.3 Pinned host memory

So far, we have allocated host memory using the standard `malloc` function in the C standard library. As a reminder, this memory has been allocated and copied to the device as follows:

```
int *a, *dev_a;
a = (int*)malloc(N * sizeof(int));
cudaMalloc((void**)&dev_a, N * sizeof(int));
cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
```

The memory allocated by `malloc` is known as *pageable memory*. This refers to the operating system's ability to *page out* this data from RAM to the disk when it hasn't been recently used, which is done automatically by every modern operating system for performance reasons. One consequence of this is that the location of pageable memory in physical RAM may change, meaning it does not have a fixed address that can be referenced by other programs (such as a device kernel).

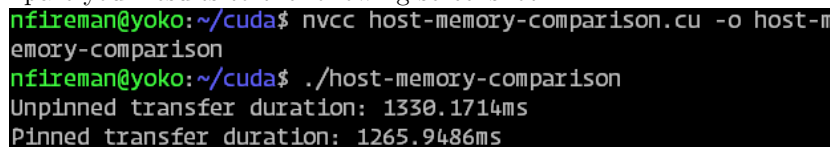
As a result of this, the above code snippet will not cause the data on the host in the array `a` to be directly copied to the device. Instead, CUDA will first allocate temporary *pinned memory* (also known as *page-locked memory*), which cannot be paged out by the operating system to the disk and thus has a consistent address. CUDA will then copy the array `a` twice: first from its original location allocated by `malloc` to the temporary pinned memory, and second from the pinned memory to the device.

In order to avoid data being copied from pageable memory to pinned memory as described above, one can manually allocate pinned memory using the CUDA function `cudaMallocHost` instead of `malloc`:

```
int *a_pinned, *dev_a;
cudaMallocHost((void**)&a_pinned, N * sizeof(int));
cudaMalloc((void**)&dev_a, N * sizeof(int));
cudaMemcpy(dev_a, a_pinned, N * sizeof(int), cudaMemcpyHostToDevice);
```

This will result in the array `a_pinned` being copied only once: from its original allocation site on the host to the device. Note, however, that pinned memory that is manually allocated in this way must also be manually freed using the function `cudaFreeHost`. Since pinned memory will (by definition) remain in RAM until it is freed, pinned memory leaks can be very costly.

**Program 6.** The program `host-memory-comparison.cu` is provided to demonstrate the runtimes when using `cudaMemcpy` to copy data from both unpinned host memory and pinned host memory. Compile and execute the program. Compare your results to the following screenshot:



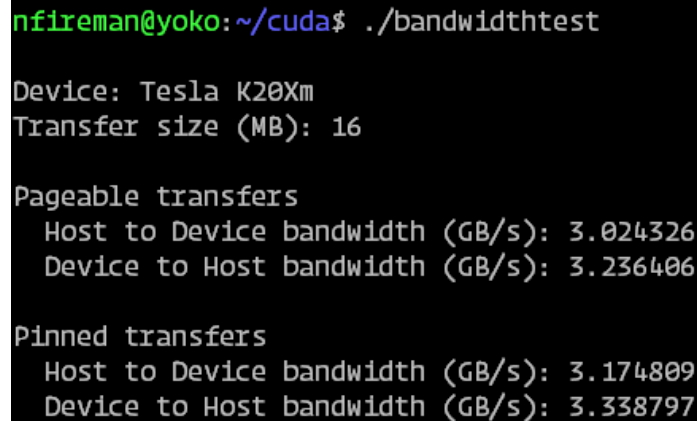
```
nfireman@yoko:~/cuda$ nvcc host-memory-comparison.cu -o host-memory-comparison
nfireman@yoko:~/cuda$ ./host-memory-comparison
Unpinned transfer duration: 1330.1714ms
Pinned transfer duration: 1265.9486ms
```

**Exercise 10.** Modify `host-memory-comparison.cu` so that both arrays stored in unpinned and pinned host memory are initialized with random integers before they're copied to the device (for an example of how to generate random integers, refer to the histogram programs from an earlier section). Record the resulting runtimes. Are they noticeably different than the original program?

**Exercise 11.** Currently, the two tests in `host-memory-comparison.cu` only measure the time it takes for memory to be copied from the host to the device. Modify the program so that both tests measure the time it takes to copy the memory from host to device, then copy the same memory back from the device to the host. Record the resulting runtimes.

**Exercise 12.** The NVIDIA blog post referenced in the resources at the end of this section provides its own program that is similar to `host-memory-comparison.cu`. Their program also compares the transfer times of pinned and unpinned host memory, but goes one step further by also computing the bandwidth of a large memory transfer from each type of host memory to the device.

Their program is provided as `nvidia-bandwidthtest.cu`. Compile and execute this program. Compare your observed output to the following screenshot:

A terminal window showing the output of the `nvidia-bandwidthtest.cu` program. The prompt is `nfireman@yoko:~/cuda$ ./bandwidthtest`. The output shows the device is a Tesla K20Xm with a transfer size of 16 MB. It then displays bandwidth results for pageable and pinned transfers, both in GB/s.

```
nfireman@yoko:~/cuda$ ./bandwidthtest

Device: Tesla K20Xm
Transfer size (MB): 16

Pageable transfers
  Host to Device bandwidth (GB/s): 3.024326
  Device to Host bandwidth (GB/s): 3.236406

Pinned transfers
  Host to Device bandwidth (GB/s): 3.174809
  Device to Host bandwidth (GB/s): 3.338797
```

#### Additional resources

- A brief NVIDIA blog post summarizing pinned host memory:  
<https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/>  
(Local copy saved to `resources/Optimize_Data_Transfers.pdf`)
- Section 10.2 of "CUDA by Example: An Introduction to General-Purpose GPU Programming" by Jason Sanders, Edward Kandrot.

## 5 Applications

### 5.1 Convolution filters

A *convolution* is a general mathematical operation that "blends" two functions together in some way to produce a third function. Convolutions have seen wide application in many fields, including image processing, where applying a *convolution filter* means transforming each pixel of an image based on what the image looks like near that pixel. For simplicity, we will be applying a convolution filter to a 1D array of pixel values, rather than the usual 2D array of pixels that comprises an image.

A convolution filter is characterized by its *mask*, which is an array of values that define how much influence nearby pixels should have when applying the filter. As an example, suppose we have the following input array and mask:

```
int input_array[8] = {3, 1, 0, 2, 6, 3, 4, 1};
int mask[5] = {1, 2, 3, 2, 1};

// will hold the result from applying 'mask' to 'input_array'.
int output_array[8];
```

Note that `output_array` will always have the same size as `input_array`. The  $i$ -th value of the output array `output_array[i]` is computed as the dot product (defined in Section 4.1) of `mask` and a `mask`-sized subarray of `input_array` that is *centered* at index  $i$ . For example, the following line computes `output_array[4]`:

```
output_array[4] = 1*0 + 2*2 + 3*6 + 2*3 + 1*4;
```

In this example, we have computed the dot product of `mask` with `{0,2,6,3,4}`, which is the `mask`-sized subarray of `input_array` centered at index 4.

One complication with this definition is when computing a value like `output_array[0]`, which is too close to the array boundary to fit a full `mask`-sized subarray centered there. In cases like these, we simply ignore the extra values of `mask` and `input_array` that don't fit:

```
output_array[0] = 3*3 + 1*2 + 0*1;
output_array[1] = 3*2 + 1*3 + 0*2 + 2*1;
output_array[6] = 6*1 + 3*2 + 4*3 + 1*2;
output_array[7] = 3*1 + 4*2 + 1*3;
```

**Program 7.** The program `convolution1.cu` implements the convolution filter algorithm described above. It uses a new feature of CUDA C, *constant memory*, indicated by the `__constant__` modifier on the declaration of `dev_mask`:

```
__constant__ int dev_mask[MASK_SIZE];
```

This memory is "constant" in the sense that it cannot be modified from the device. Constant memory is initialized using `cudaMemcpyToSymbol`, which has

an identical signature to `cudaMemcpy` that we have already used. The following line of code initializes the constant memory `dev_mask` by copying the contents of `mask` into it:

```
cudaMemcpyToSymbol(dev_mask, mask, MASK_SIZE * sizeof(int));
```

Note that there isn't an equivalent of `cudaMalloc` that allocates constant memory. The above variable declaration of `dev_mask` performs this role.

Compile and execute `convolution1.cu`. Compare your observed output to the following screenshot:

```
nfireman@yoko:~/cuda$ nvcc convolution1.cu -o convolution1
nfireman@yoko:~/cuda$ ./convolution1
Input array size: 100000
Convolution duration: 835.143ms (100000 trials)
```

**Exercise 13.** Modify `convolution1.cu` so that the `mask` array is copied to global device memory and passed as an argument to `convolution_kernel`, instead of using constant memory. Compare the performance of the modified program to the original.

**Program 8.** As shown in Section 4.1, shared memory can greatly improve the performance of a CUDA application when multiple threads in a block need to access the same data. The provided program `convolution2.cu` gives a second implementation of a convolution filter using shared memory.

To take advantage of shared memory, one thread in each thread block is assigned to copy a so-called *tile* of the input array into shared memory. The tile will contain all values in the input array that each thread in the thread block will need to compute its own output value. Because of how convolution filters are defined, two adjacent threads will access many of the same input values, and so almost all of the input values copied to shared memory will be accessed multiple times. By copying these values to shared memory *once* before the convolution takes place, this program gains the benefit of shared memory *multiple times*; once for each additional time the value is read.

You should find that the shared memory version applies convolution filters of the same size noticeably faster compared to the previous implementation:

```
nfireman@yoko:~/cuda$ nvcc convolution2.cu -o convolution2
nfireman@yoko:~/cuda$ ./convolution2
Input size: 100000
Convolution duration: 490.455ms (100000 trials)
```

**Exercise 14.** The size of the mask in both convolution programs is defined by `MASK_SIZE`. Record the runtimes when this value is modified to 11, 15, and 21 in both programs. Which runtime increases faster? Try to explain your findings.

**Exercise 15.** Currently, the shared memory array `tile` is initialized solely by the thread in each block where `threadIdx.x` is 0. Modify `convolution2.cu` so that the shared memory is initialized concurrently by all threads in each block.

**Exercise 16.** Currently in `convolution2.cu`, every input array value needed by a thread block is copied to the shared memory array `tile`. However, the values on the boundary of the tile won't be used as often as those not on the boundary. In fact, the first and last elements of `tile` will only be used by one thread in the block each (the first and last threads, respectively), so it's a net performance loss to copy these elements into shared memory.

Modify `convolution2.cu` so that `tile` has width equal to `THREADS_PER_BLOCK`, the number of threads in each thread block. In doing so, you should aim to maximize performance by reading input array values from global memory when they won't be needed by the maximum number of threads.

### Kernel profiling with nvprof

We have seen in several example programs how to use CUDA events to measure the execution durations of kernels. This section will briefly show how the profiling tool `nvprof`, provided as part of the CUDA environment, can be used to measure a kernel's execution duration automatically. By passing a compiled program to `nvprof`, the profiler will run the program and monitor its execution, producing a report that lists the share of execution time spent on each GPU activity and CUDA API call.

The following screenshot demonstrates the usage of `nvprof` to profile `convolution1.cu` (omitting the profiler's report for brevity). Note that the original program's output is printed, since `nvprof` executes the program to obtain its measurements.

```
nfireman@yoko:~/cuda$ nvcc convolution1.cu -o convolution1
nfireman@yoko:~/cuda$ nvprof ./convolution1
==7007== NVPROF is profiling process 7007, command: ./convolution1
Input array size: 100000
Convolution duration: 878.032ms (100000 trials)
==7007== Profiling application: ./convolution1
==7007== Profiling result:
```

The report should list three rows with the type GPU Activities. One is named `convolution_kernel(int*, int*, int, int)`, corresponding to the kernel function that applies the convolution filter. The other two rows are named `[CUDA memcpy HtoD]` and `[CUDA memcpy DtoH]`, which refer to host-to-device and device-to-host memory transfers, respectively.

**Exercise 17.** Use `nvprof` to measure the execution time of the kernels in `convolution1.cu` and `convolution2.cu`. Compare the results to the execution times printed by the programs themselves that were computed using CUDA events.

### Additional resources

- Chapter 7 of "Programming Massively Parallel Processors: A Hands-on Approach", 3rd Edition by David B. Kirk, Wen-mei W. Hwu.
- "Image Convolution with CUDA" by Victor Podlozhnyuk.



[http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_64\\_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf)

(Local copy saved to `resources/convolutionSeparable.pdf`)

- NVIDIA documentation for `nvprof`:

<https://docs.nvidia.com/cuda/profiler-users-guide/index.html>

(Local copy saved to `resources/CUDA_Profiler_Users_Guide.pdf`)

## 5.2 Prefix sum

Given an array of numbers `data` of length `N`, the *prefix sum* operation computes a new array `prefixes` such that `prefixes[i]` is the sum of `data[0]` through `data[i]`. In other words, the prefix sum operation computes the following sums:

```
prefixes[0] = data[0];
prefixes[1] = data[0] + data[1];
...
prefixes[N] = data[0] + data[1] + ... + data[N];
```

A serial prefix sum algorithm can therefore be implemented as follows:

```
prefixes[0] = data[0];
for (int i = 1; i < N; i++) {
    prefixes[i] = prefixes[i - 1] + data[i];
}
```

At first glance, it may not seem possible to parallelize this loop, since each loop iteration computes `prefixes[i]` using the previous iteration's computation of `prefixes[i - 1]`. This section will develop a parallel prefix sum algorithm inspired by the *Kogge-Stone adder*, which is a class of circuit that performs binary addition.

Using the Kogge-Stone technique, our implementation will use `N` threads (one per input element) in a single thread block. Each thread will perform  $\log_2(N)$  additions, which means that overall, this implementation should outperform the serial algorithm above, which has to perform `N` sequential additions. Note that if each of the `N` threads performs  $\log_2(N)$  additions, then overall there will be  $N \log_2(N) > N$  additions, meaning that while the parallel algorithm takes less time per thread than the serial algorithm, the parallel version must do more work overall.

Two programs implementing the Kogge-Stone prefix sum are provided, where the first implementation is intentionally buggy. Thus, another objective of this section will be to demonstrate the use of the `cuda-memcheck` tool provided by NVIDIA in order to diagnose and fix the buggy first program.

**Program 9.** The first implementation of the Kogge-Stone prefix sum is provided in `kogge-stone1.cu`. As Exercise 18 will show, this implementation has a race condition for large values of `N` that will be resolved in the next implementation, but the logic of the prefix sum itself is sound and will be documented here. The program should work for the provided value `N = 32`.

The kernel function `kogge_stone_scan` first copies the input array `in` into the shared memory array `partials` to improve performance, as we’ve seen before. The `partials` array is copied to the output array at the end of the kernel, so the purpose of the kernel’s main loop is to compute the prefix sum array in `partials`.

The loop variable of this loop is `stride`, and the loop invariant of interest is that at the beginning of the iteration when `stride = k`, the values `partials[0]`, `partials[1]`, ..., `partials[k-1]` are each equal to the appropriate element of the prefix sum. Note that this holds before the first iteration, since `partials[0]` is initialized to `in[0]`, which is also the first element of the prefix sum array. The loop invariant is then maintained by adding a suitable prior element to each unfinished element of `partials`, motivated by the similar process in the serial version of the program. Note the use of the `__syncthreads()` function to guarantee that all threads start each loop iteration together.

**Exercise 18.** Compile and execute the program. Then recompile the program where the input array size `N` is set to 128, 256, 512, and 1024. Execute the program several times for each new value of `N`. The program should start producing incorrect results; record the lowest value of `N` where you find that incorrect results occur.

As Exercise 18 showed, `kogge-stone1.cu` has a bug and produces incorrect results when `N` is large. To debug this program, we will use the `cuda-memcheck` tool provided by NVIDIA as part of the CUDA development environment. It can be used to debug race conditions, memory leaks, and several other classes of bugs. A program is analyzed for a particular class of bug using the `-tool` command-line argument.

**Exercise 19.** Use `cuda-memcheck` to check `kogge-stone1.cu` for memory access errors by using the `memcheck` tool. Note that `cuda-memcheck` analyzes executables (as opposed to source code), so a compiled version of `kogge-stone1.cu` should be passed to `cuda-memcheck` instead of the source code itself.

```
nfireman@yoko:~/cuda$ cuda-memcheck --tool memcheck kogge-stone1
===== CUDA-MEMCHECK
Results valid: yes
Input array size: 32
Elapsed time: 1.516896ms
===== ERROR SUMMARY: 0 errors
```

**Exercise 20.** Recompile `kogge-stone1.cu` using the `-lineinfo` command-line argument (this will cause `cuda-memcheck` to attach source code line numbers to

the errors it finds). Then use `cuda-memcheck` to check for race conditions with the `racecheck` tool.

You should be prompted with several race condition warnings. Record the program line(s) referenced by each warning message.

```
nfireman@yoko:~/cuda$ nvcc -lineinfo kogge-stone1.cu -o kogge-stone1
nfireman@yoko:~/cuda$ cuda-memcheck --tool racecheck kogge-stone1
===== CUDA-MEMCHECK
===== WARN: Race reported between Read access at 0x000000b0 in /S
ECS/home/n/nfireman/cuda/kogge-stone1.cu:20:kogge_stone_scan(int*, in
t*)
===== and Write access at 0x000000c8 in /SECS/home/n/nfireman
/cuda/kogge-stone1.cu:20:kogge_stone_scan(int*, int*) [112 hazards]
=====
```

The `racecheck` tool should point to the following line of `kogge-stone1.cu` as the source of the race condition:

```
partials[tid] += partials[tid - stride];
```

Since the value of `partials[tid - stride]` may also be updated by a different thread, whether that update occurs before or after the update to `partials[tid]` will change the effect of the above line of code. This is the source of the race condition detected by `racecheck`.

To resolve the race condition, we *buffer* the updates to the `partials` array until the end of each loop iteration. That is, we maintain two arrays, `partials` and `partials_next`, and instead of applying changes directly to `partials`, we store the new desired value of `partials[tid]` in `partials_next[tid]`. Thus, the above line of code is transformed to the following:

```
partials_next[tid] = partials[tid] + partials[tid - stride];
```

At the end of each loop iteration, we copy `partials_next` into `partials` all at once. This guarantees that the value `partials[tid - stride]` has a consistent value when it is read by thread `tid`, regardless of the order in which the threads execute.

**Exercise 21.** Run the `racecheck` tool on `kogge-stone2.cu` to verify that a race condition is not detected in this version of the program.

### Additional resources

- "Parallel Prefix Sum (Scan) with CUDA" by Mark Harris.  
[http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/scan/doc/scan.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/scan/doc/scan.pdf)  
(Local copy saved to `resources/scan.pdf`)
- Chapter 8 of "Programming Massively Parallel Processors: A Hands-on Approach", 3rd Edition by David B. Kirk, Wen-mei W. Hwu.

- Official documentation for `cuda-memcheck`:  
<https://docs.nvidia.com/cuda/cuda-memcheck/index.html>  
(Local copy saved to `resources/CUDA_Memcheck.pdf`)

### 5.3 Mergesort

Mergesort is a fundamental general-purpose sorting algorithm that is both widely used and relatively simple to implement. In its simplest form, the algorithm first recursively splits an unsorted input array into two equal-sized groups repeatedly until each group has only one element. The second phase of the algorithm is to *merge* two sorted groups into a single larger sorted group, until only one sorted group containing all of the original input values remains. Note that at the start of the second phase, each "group" only has one element each, meaning it is technically sorted.

Figure 1 demonstrates this procedure on the input array  $\{56, 29, 35, 42, 15, 41, 75, 21\}$ . The top half of the figure shows the splitting phase and the bottom half shows the merging phase.

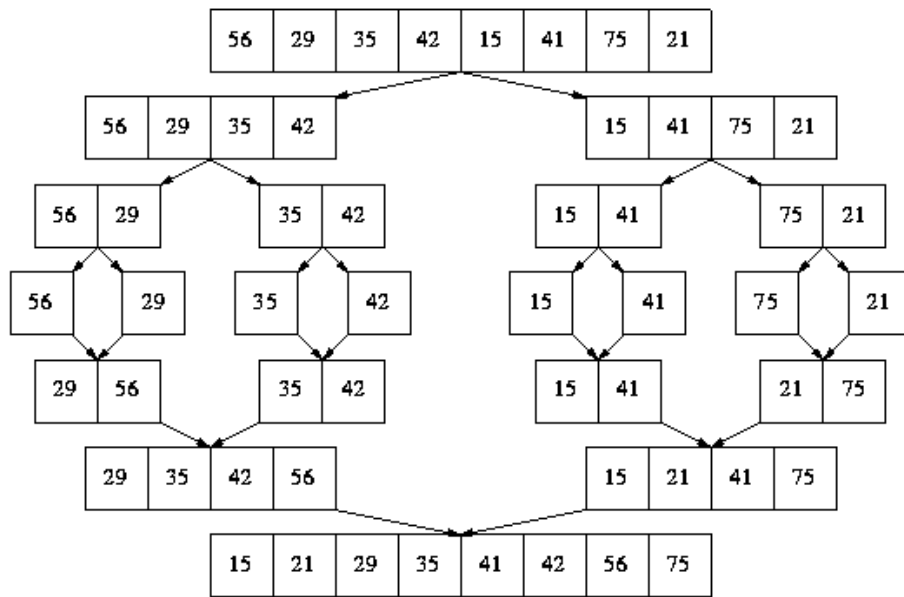


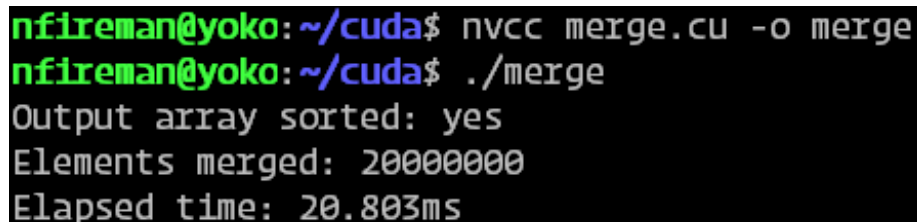
Figure 1: (source: <https://webdocs.cs.ualberta.ca/~holte/T26/merge-sort.html>)

**Program 10.** This section presents the program `merge.cu`, which demonstrates a parallel algorithm for the merging phase of mergesort. The algorithm we will implement was proposed in 2012 for use in the MPI (Message Passing Interface) framework, but we will adapt it accordingly for the CUDA environment. The

paper that originally introduced the algorithm is referenced at the end of this section.

First, compile and execute the program. Compare your results to the following screenshot. The remainder of this section will document how this program works.

**Note:** the justification below is quite complicated and understanding it is not required. It's summarized here as an example of how parallelism can still be applied in practice to nontrivial problems, even when it doesn't seem like it's possible to do so.



```
nfireman@yoko:~/cuda$ nvcc merge.cu -o merge
nfireman@yoko:~/cuda$ ./merge
Output array sorted: yes
Elements merged: 20000000
Elapsed time: 20.803ms
```

### 5.3.1 Serial merge

The first subroutine of our merging kernel will be a function that performs a *serial merge* of two sorted input arrays **a** (of length **m**) and **b** (of length **n**) into a third output array **out**:

```
__device__ void serial_merge(int *a, int m, int *b, int n, int *out)
```

The serial merge maintains values **i** and **j**, which are the smallest unmerged indices of **a** and **b**, respectively. The index **k** of **out** that will be merged into next is also maintained. It first merges the smallest of **a[i]** and **b[j]** into **out[k]** (incrementing indices where appropriate) until all values of one input array have been merged. Then, the remaining unmerged values in the other input array are merged to complete the algorithm.

Note the `__device__` keyword in the declaration to signify that this function will be called from device threads. Our aim in the remainder of the algorithm will be to correctly divide the work of merging two input arrays between many threads, each of which will serially merge distinct subarrays of the inputs into a subarray of the output array.

In particular, we will divide the work of computing the **m + n** output elements evenly among the allocated threads (recall that the number of threads allocated to a kernel can be computed from the device as `blockDim.x*gridDim.x`). Each thread can thus determine which output indices it is responsible for. Knowing this, it remains to compute which subarrays of the two input arrays should be merged into each thread's slice of the output array. In other words, we need to be able to work backwards to find the input subarrays that will be merged into a given output subarray.

### 5.3.2 Co-rank

The `co_rank` function is the second subroutine in our merging kernel, and is responsible for determining which subarrays of the input will be merged into a given subarray of the output.

We first observe that for any subarray  $\{out[0], out[1], \dots, out[k-1]\}$  of the output array, its elements will be obtained from merging subarrays  $\{a[0], a[1], \dots, a[i-1]\}$  and  $\{b[0], b[1], \dots, b[j-1]\}$  of the input arrays for some  $i$  and  $j$ , since input elements are merged from left to right into the output array. In this situation, we say that the input indices  $i$  and  $j$  are the *co-ranks* of the output index  $k$ . Note that  $i + j$  must equal  $k$ , since subarrays of lengths  $i$  and  $j$  are merged into a subarray of length  $k$ .

Next, suppose we have two output indices  $k_0$  and  $k_1$  such that  $k_0 < k_1$ ,  $k_0$  has co-ranks  $i_0$  and  $j_0$ , and  $k_1$  has co-ranks  $i_1$  and  $j_1$ . Then since  $\{a[0], \dots, a[i_0-1]\}$  and  $\{b[0], \dots, b[j_0-1]\}$  are merged into  $\{out[0], \dots, out[k_0-1]\}$ , the remaining subarrays  $\{a[i_0], \dots, a[i_1-1]\}$  and  $\{b[j_0], \dots, b[j_1-1]\}$  must be merged into  $\{out[k_0], \dots, out[k_1-1]\}$ .

To see why this fact is useful, we need to apply it to the case where a CUDA thread is assigned to merge into the output indices  $k_0$  to  $k_1 - 1$ . In this case, the co-ranks of  $k_0$  and  $k_1$  tell us exactly which subarrays of  $a$  and  $b$  our thread needs to merge into  $\{out[k_0], \dots, out[k_1-1]\}$ .

We can now define the behavior of the `co_rank` function:

```
__device__ int co_rank(int k, int *a, int m, int *b, int n)
```

If  $k$  has co-ranks  $i$  and  $j$ , then the above call to `co_rank` will return  $i$ . Note that since  $k=i+j$  (as we noted above),  $j$  can be computed as  $k-i$ , so we only need to explicitly return  $i$ .

It remains to develop an algorithm to find the co-ranks of a given index  $k$ . In our implementation of `serial_merge`, the co-ranks of  $k$  will be the values of  $i$  and  $j$  when `out[k]` is written to (since at that point, we have merged  $\{a[0], a[1], \dots, a[i-1]\}$  and  $\{b[0], b[1], \dots, b[j-1]\}$  into  $\{out[0], out[1], \dots, out[k-1]\}$ ). Hence, one way of computing the co-ranks of  $k$  would be to "simulate" our serial merge procedure on the entire array.

However, this would be very inefficient for the threads responsible for merging into the end of `out`. To compute the co-ranks of large output indices in this way, the serial merge would have to process almost all of the elements in both input arrays, one at a time. This work would need to be repeated by every thread assigned to merge into high indices of the output array.

Instead, we will use the fact that both input arrays are sorted to more quickly compute co-ranks without needing to process every input element. We will also use the following characterization of the co-ranks  $i$  and  $j$  of  $k$ : they are indices so that  $i+j == k$ ,  $a[i-1] \leq b[j]$ , and  $b[j-1] < a[i]$ .

Using an approach inspired by binary search, the main loop of the `co_rank` function finds the co-ranks of  $k$  by maintaining candidate indices  $i$  and  $j$  such that  $i+j$  always equals  $k$ . At each loop iteration, if  $a[i-1] > b[j]$ , then  $j$

is increased and `i` is decreased by the same amount. Otherwise, if `b[j-1] >= a[i]`, then `i` is increased and `j` is decreased by the same amount. Once neither of these inequalities holds, `i` and `j` must be the co-ranks of `k`, and `i` is returned.

### 5.3.3 Parallel merge kernel

The `parallel_merge_kernel` function ties together the `serial_merge` and `co_rank` subroutines we have just developed. Each thread computes a unique subarray of the output based on its `tid` value. The thread then computes the co-ranks `i` and `j` of its lowest output index `k`, as well as the co-ranks `i_next` and `j_next` of the next thread's lowest output index `k_next`. Finally, the thread performs a serial merge of `{a[i], ..., a[i_next-1]}` and `{b[j], ..., b[j_next-1]}` into `{out[k], ..., out[k_next-1]}`.

**Exercise 22.** Try different values of `THREADS_PER_BLOCK` and `NUM_BLOCKS` while keeping the original value of `N=10000000` fixed, and record the runtimes along with the values used. Aim to improve the performance of the unmodified program.

### Additional resources

- "Efficient MPI Implementation of a Parallel, Stable Merge Algorithm" by Christian Siebert, Jesper Larsson Träff (EuroMPI 2012).  
[https://link.springer.com/chapter/10.1007%2F978-3-642-33518-1\\_25](https://link.springer.com/chapter/10.1007%2F978-3-642-33518-1_25)  
(Local copy saved to `resources/MPI_Mergesort.pdf`)
- Documentation for the `merge` operation in the `moderngpu` CUDA library:  
<https://moderngpu.github.io/merge.html>  
(Local copy saved to `resources/moderngpu_merge.pdf`)
- Chapter 11 of "Programming Massively Parallel Processors: A Hands-on Approach", 3rd Edition by David B. Kirk, Wen-mei W. Hwu.
- "Designing efficient sorting algorithms for manycore GPUs" by N. Satish, M. Harris and M. Garland (2009 IEEE International Symposium on Parallel & Distributed Processing).  
<https://mgarland.org/files/papers/gpusort-ipdps09.pdf>  
(Local copy saved to `resources/gpusort-ipdps09.pdf`)

## 5.4 Sparse matrix-vector multiplication

This section will make extensive use of *matrices*, which is the mathematical name for two-dimensional arrays. The following C code defines a 3 by 4 matrix `A` (meaning that `A` has 3 rows and 4 columns):

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

This section will present a solution for the *matrix-vector multiplication* problem, which is defined as follows. Suppose we have an  $m$  by  $n$  matrix  $A$  and a vector  $x$  of length  $n$ . Then the matrix-vector product  $A * x$  is a vector  $y$  of length  $m$ , where  $y[i]$  is the dot product of the  $i$ -th row of  $A$  with  $x$ . In other words, for  $0 \leq i < m$ :

```
y[i] = A[i][0]*x[0] + A[i][1]*x[1] + ... + A[i][n-1]*x[n-1];
```

In this section, we will also additionally assume that the input matrix  $A$  is *sparse*; a matrix is said to be sparse if most of its values are zero. There's no precise cutoff for when a matrix is sparse or not, but in practice it generally means that only a small (constant) number of elements are nonzero per row or column.<sup>1</sup>

Storing a sparse matrix as a two-dimensional array (like in the above code) is often inefficient, since the vast majority of the memory is going to be used to store zero-valued elements. To take better advantage of sparsity, *sparse matrix encodings* have been developed to store sparse matrices in memory more efficiently than with a two-dimensional array.

This section will define the *Compressed Sparse Row* (CSR) encoding for sparse matrices. For a given sparse matrix  $A$ , the aim of the CSR scheme is to encode the positions and values of just the nonzero elements of  $A$ ; any element left unencoded is guaranteed to be 0. Assuming that  $A$  is sufficiently sparse, this will save significant memory over the traditional array encoding of  $A$ .

Suppose we have the following matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 4 & 7 & 5 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 2 & 0 \end{bmatrix}$$

The CSR compression scheme uses three arrays to encode the matrix  $A$ . The first array `data` will simply be the nonzero elements of  $A$  listed in order, where we read the top row left-to-right, and continue through the rows top-to-bottom:

```
float data[7] = {2, 1, 4, 7, 5, 3, 2};
```

The second array `col_index` will be the same size as `data`, and `col_index[i]` will be the column index of the corresponding nonzero value `data[i]`. For example, `data[2]` references the 4 belonging to column index 1 of  $A$ , so `col_index[2]` should be 1.

```
int col_index[7] = {0, 3, 1, 2, 3, 0, 2};
```

---

<sup>1</sup>If you're familiar with asymptotic notation, then assuming an  $m$  by  $n$  matrix is sparse usually means that at most  $O(m + n)$  of its  $mn$  elements are nonzero.



Since `col_index` encodes the column index of each nonzero value of  $A$ , we could complete the encoding by defining a similar array `row_index` which encodes the row index of each nonzero value:

```
int row_index[7] = {0, 0, 1, 1, 1, 3, 3};
```

Note that the entries in `row_index` will correspond to the two values in row 0, then the three values in row 1, then the two values in row 3. This is due to the order in which the nonzero values of  $A$  were read into `data`. To take advantage of this, we can instead define the array `row_ptr`, which only records the first index of `data` corresponding to each row:

```
int row_ptr[4] = {0, 2, 5, 5};
```

Here we have that `row_ptr[1]` is 2 since `data[2]` is the first nonzero value on row 1. Likewise, `row_ptr[0]` is 0 and `row_ptr[3]` is 5. In the case where a row is all zeros (e.g. row 2), its value will be equal to the next value in `row_ptr`. Thus, `row_ptr[2] == row_ptr[3] == 5`.

This definition yields the following invariant: the nonzero values of  $A$  in row  $i$  lie between the indices `row_ptr[i]` and `row_ptr[i+1]-1` of `data`. If row  $i$  is all zeros, then we can deduce this from the fact that `row_ptr[i]` will equal `row_ptr[i+1]`.

For convenience, an extra element is often added to the end of `row_ptr` so that the above invariant still holds for the last row of  $A$ . The extra element will always be equal to the length of `data`:

```
int row_ptr[5] = {0, 2, 5, 5, 7};
```

Applying the invariant to row 3 tells us that the nonzero elements of row 3 lie between indices `row_ptr[3] == 5` and `row_ptr[4]-1 == 6` of `data`. This correctly points to the elements 3 and 2 on the last row of  $A$ .

**Exercise 23.** Suppose a sparse matrix of `float` values  $A$  has 1000 rows, 1000 columns, and that  $A$  has exactly 2 nonzero entries per row. Compare the amount of memory needed to store  $A$  as a traditional two-dimensional array to the memory needed to store it as a CSR-encoded matrix. Assume that `float` and `int` values each use 4 bytes of memory.

**Program 11.** The program `spmv-csr.cu` implements matrix-vector multiplication for CSR-encoded matrices. The kernel function `spmv_csr` is similar to the kernel function found in `dot-product-parallel.cu` (which computed the dot product of two vectors), since matrix-vector multiplication is defined as a series of dot products.

The following code block uses what has been previously established about the CSR encoding to compute the dot product of the row of index `row` of  $A$  with the input vector `x`:

```

float partial = 0;
int min_idx = row_ptr[row];
int max_idx = row_ptr[row + 1];
for (int i = min_idx; i < max_idx; i++) {
    partial += data[i] * x[col_index[i]];
}
y[row] = partial;

```

First, subarrays of `data` and `col_index` corresponding to row index `row` are computed using `row_ptr`. Each element in this subarray of `data` is then multiplied with the corresponding value of `x` and added to the intermediate result `partial`. At the end of the loop, `partial` contains the dot product, and it is stored in the output vector `y`.

**Exercise 24.** Compile and execute `spmv-csr.cu`. Improve the performance of the program by trying several different values for `THREADS_PER_BLOCK` and `NUM_BLOCKS`. Note the runtime for each pair of values tried.

**Exercise 25.** The value `NONZEROELTS_PER_ROW` controls the number of nonzero elements in each row of the generated input matrix. To see how the CSR encoding performs when the input matrix is less sparse, record the program runtime when the elements per row is increased to 100, 250, and 500.

#### Additional resources

- "Efficient Sparse Matrix-Vector Multiplication on CUDA" by Nathan Bell, Michael Garland.  
<https://www.nvidia.com/docs/I0/66889/nvr-2008-004.pdf>  
(Local copy saved to `resources/nvr-2008-004.pdf`)
- Chapter 10 of "Programming Massively Parallel Processors: A Hands-on Approach", 3rd Edition by David B. Kirk, Wen-mei W. Hwu.