

# AMS 148 Chapter 6: Histogram, Sort, and Sparse Matrices

Steven Reeves

Now that we have completed the more fundamental parallel primitives on GPU, we will dive into more advanced topics. Histogram is a very important algorithm for applications in statistics, and image processing. Following histogram, we will discuss sparse matrices and their applications.

## 1 Histogram

Histogram is an accurate representation of the distribution of numerical data. It's an estimate of the probability distribution of a quantitative variable. Histograms are based on a concept called **binning**. Histogram requires a search on an array, and according to a certain criteria, the value is binned. That is – when the algorithm detects a value of the array that is classified in the bin, the bin count is increased by one.

A serial implementation is straightforward:

```
void histogram( unsigned int *histo, type *measurements, int bin_count, int array_length)
{
    for (int i = 0; i < bin_count; i++)
        histo[i] = 0; //Zero out histogram

    for(int i = 0; i < array_length; i++)
        histo[computeBin(measurements[i])]++;
}
```

Depending on your measurement data, you could have strings, integers, floats, or some other data type. Histograms aren't necessarily dependent on numerical data, although they classically are. An example: we could create a histogram of the class based on eye color. We could use the strings: "Green", "Blue", "Brown" etc or assign a numerical value to each string.

Note that in this implementation, we need to calculate some bin for classification. This portion will be application dependent. For now, we will use some arbitrary function `computeBin`.

### 1.1 First Try Parallel Histogram

In this chapter we will examine 4 different implementations. Our first try will be to unroll the loop in the serial implementation and launch  $n$ , each of which increments one bin count.

```
__global__ void first_hist(unsigned int *histo, type *data, int n)
{
    int tid = threadIdx.x +blockDim.x*blockIdx.x;
    if(tid > n)
    {
        return;
    }
    histo[computeBin(data[tid])]++;
}
```

In this case, `computeBin` is a `__device__` function.

However, this does not work, and it is important to understand why.

#### 1.1.1 Example

To show why it does not work, it is best to reveal an example. In this case we use the modulo operator to compute the bin number.

```

__global__ void faux_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x * blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    d_bins[myBin]++;
}

```

In the main routine we will have the following parameters:

```

const int ARRAY_SIZE = 65536;
const int ARRAY_BYTES = ARRAY_SIZE*sizeof(int);
const int BIN_COUNT = 16;
const int BIN_BYTES = BIN_COUNT*sizeof(int);

```

Here we have 65536 elements, and 16 bins. So we should expect 4096 elements per bin. However, this is the output:

```

steven@steven-ubuntu: ~/AMS/AMS148/lectures/lecture_notes/lecture6
steven@steven-ub... x steven@steven-ub... x steven@steven-ub... x steven@steven-ub... x +
Histogram =
Bin 0 = 9
Bin 1 = 9
Bin 2 = 9
Bin 3 = 9
Bin 4 = 9
Bin 5 = 9
Bin 6 = 9
Bin 7 = 9
Bin 8 = 9
Bin 9 = 9
Bin 10 = 9
Bin 11 = 9
Bin 12 = 9
Bin 13 = 9
Bin 14 = 9
Bin 15 = 9
steven@steven-ubuntu: ~/AMS/AMS148/lectures/lecture_notes/lecture6 (master)$ █

```

Figure 1: False Histogram

This is clearly not this histogram we’re looking for. So what is going on here? Let’s look at the `faux_histo` kernel’s line

```
d_bins[myBin]++;
```

In this line each thread is doing 3 things:

1. Read Bin Value from global memory to a register
2. Increment Bin Value
3. Write Bin Value from register to global memory

Lets illustrate how this is goes wrong. Suppose we have two threads, thread 0 and thread 1. Both want to increment the same bin, and are running at the same time. Both threads load the item from global memory into its register, without issue. However, now both threads have the same value in their register, and increment it up one. In the last step they both write their data to the same place. Say the original value was 5, then both threads will update the value to 6. In this case, they both write 6 back to global memory, when the answer should be 7. This error is another example of a **race condition**. The problem is that incrementing global memory takes multiple steps, and it is possible, as we’ve shown with this example, for 2 processors to interleave these steps.

So its clear that this implementation will not work. Now lets examine 3 different ways we might implement a parallel histogram that will work.

## 1.2 A Second Approach Using Atomics

Recall that in the first attempt parallelization didn't work because the read, modify, and write operations were inducing a race condition in global memory. What we seek is to have the read, modify and write operations actually be 1 operation instead of 3. We call such an operation *atomic*. What the GPU does, in essence, is locks a particular memory location during the read/modify/write process so that no other thread can access it. This is a simple change in our code.

```
__global__ void simple_histo(int *d_bins, const int *d_in, const int BIN_COUNT)
{
    int myId = threadIdx.x + blockDim.x*blockIdx.x;
    int myItem = d_in[myId];
    int myBin = myItem % BIN_COUNT;
    atomicAdd(&(d_bins[myBin]),1); //Add bin by 1 atomically
}
```

All we did in our first try histogram was change the increment line to be atomic. A simple change but let's see the results.

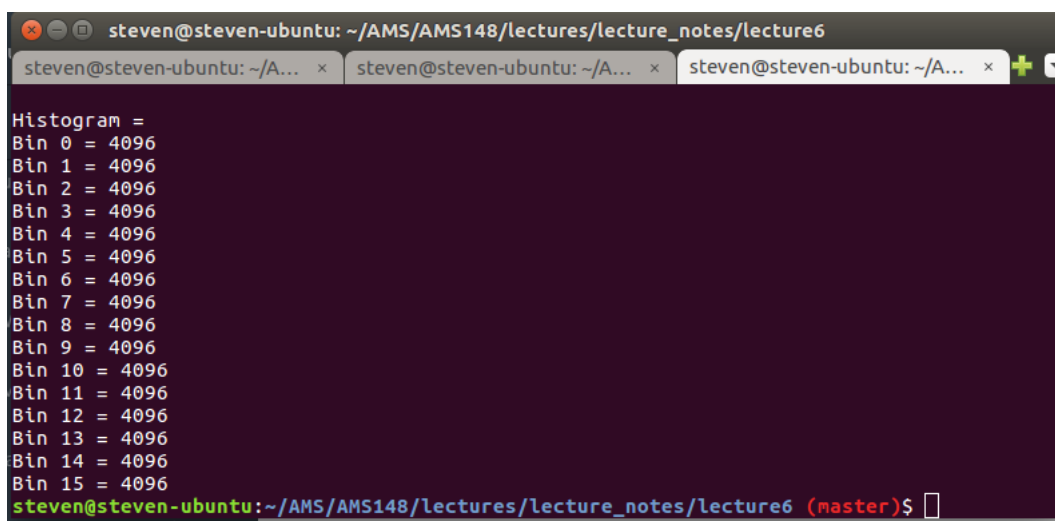
A terminal window screenshot showing the output of a histogram program. The terminal title is 'steven@steven-ubuntu: ~/AMS/AMS148/lectures/lecture\_notes/lecture6'. The output displays 'Histogram =' followed by 16 lines, each showing 'Bin X = 4096' for X from 0 to 15. The prompt at the bottom is 'steven@steven-ubuntu:~/AMS/AMS148/lectures/lecture\_notes/lecture6 (master)\$'.

Figure 2: Histogram that works

As we see, we get the values that we expect. A disadvantage of this method is that it serializes access to any particular bin during the atomic operation. Many threads might want to update the bin, but only one will be able to do so at a time. This contention in the atomic will likely be a performance bottleneck in a histogram code.

Lets say that we have a million measurements, which we have to create a histogram for. Supposing that we use the atomic add method, which of the following will be the fastest?

1. A histogram with 10 bins
2. A histogram with 100 bins
3. A histogram with 1000 bins

## 1.3 Implementing Histogram using Shared Memory

In general algorithms that rely on atomics will have limited scalability, because the atomics limit the amount of parallelism. If we have 100 bins and use atomics to access them, we essentially only have 100 points of concurrency. Thus, it will be better to reduce the number of atomic interactions. Or, use hardware to optimize the necessary atomic operations. To do this, we'll use shared memory, and privatization of local histograms. Then we will combine the histograms for a global histogram.

```

__global__ void smem_histogram(int *d_bins, const int *d_in, const int BIN_COUNT, const int size)
{
    //Create Private copies of histo[] array;
    __shared__ unsigned int histo_private[BIN_COUNT];

    int tid = threadIdx.x;
    if(threadIdx.x < BIN_COUNT)
        histo_private[tid] = 0;
    __syncthreads();

    int i = threadIdx.x + blockDim.x*blockIdx.x;
    //stride total number of threads
    while( i < size)
    {
        int buffer = i % BIN_COUNT;
        atomicAdd(&(histo_private[buffer]), 1);
        i += stride;
    }
    __syncthreads();

    //Build Final Histogram using private histograms.
    if(tid < BIN_COUNT)
    {
        atomicAdd(&(d_bins[tid]), histo_private[tid]);
    }
}

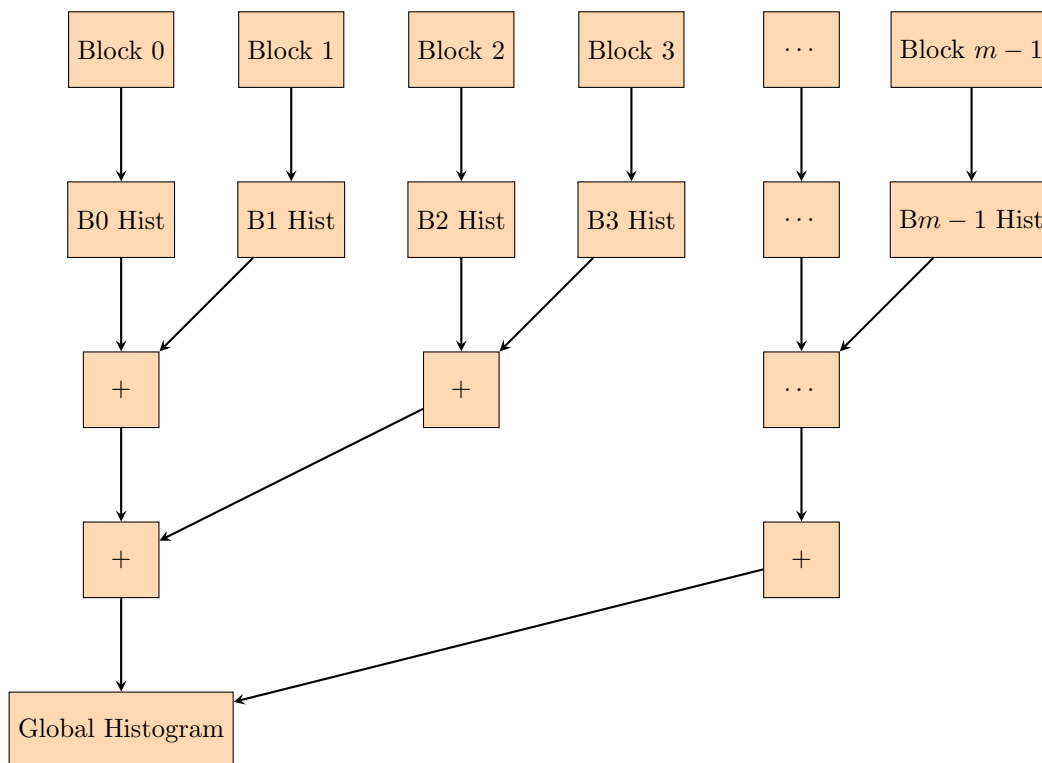
```

In our example problem we find that using this kernel reduces the time from 0.02ms to 0.009ms, when using 128 grids, vs 4096 grids in the simple atomic case.

## 1.4 Further Expanding Histogram

Note that we can further expand this private histogram method, and instead of computing atomic adds one can use a reduction routine on local histograms. A computational tree would look like this.

Figure 3: Generalization Histogram



This type of histogram routine would remove the atomic adds from the final histogram allowing for  $\log_2(n)$  steps per bin vs  $n$  steps per bin, where  $n$  is the number of local histograms. Note that it may be more efficient to define an add operator for histograms, instead of reducing bin by bin. The implementation is left to an interested reader.

Note that this is not the end-all to histogram optimization. We will discuss more ways of saving time in the chapter on optimization. There is a more complicated algorithm called *sort then reduce by key*, which doesn't require the use of atomics.

## 1.5 Application of Histogram to Generate Color Distributions

One useful application of histogram is generate color distributions in an image for editing. All major photo editing software generates color "curves" which represent the histogram of each color. Sometimes a photographer may want to alter the distribution of colors, generally to give some artistic effect or enhance/mute colors.

In this application we'll assume our image is *true color*, that is 24-bit, 8-bits of red, green, and blue respectively. This will impact the number of bins to use in our histogram. Assuming each color channel has 8-bits, will imply 256 different shades in each channel. So, naturally, our histogram will have 256 bins. Figure 4 shows the image that we will compute the color histograms on.



Figure 4: A stained glass window with interesting colors

This image is of a stained glass window with many different colors, but has strong reds, greens, and blues. In the previous examples, our histogram kernels relied on one dimensional indexing. In the case of images we'll need to map a two dimensional index into something a one dimensional histogram will understand. We will use the shared memory model of histogram to compute our color distributions.

```

__global__ void histogram_smem_atomics(const unsigned char *in,
                                     int width, int height,
                                     unsigned int *out)
{
    //pixel coordinates
    int x = threadIdx.x + blockDim.x*blockIdx.x;
    int y = threadIdx.y + blockDim.y*blockIdx.y;

    // linear thread index within 2D block
    int t = threadIdx.x + threadIdx.y*blockDim.x;

    //initialize shared memory
    __shared__ uint3 smem[NUM_BINS];
    smem[t].x = 0; // R

```

```

smem[t].y = 0; // G
smem[t].z = 0; // B
__syncthreads();

//create block histograms
uint3 rgb;
rgb.x = (unsigned int)(in[y * width + x]); //Numbers between 0 and 255
rgb.y = (unsigned int)(in[(y + height) * width + x]);
rgb.z = (unsigned int)(in[(y + height*2) * width + x]);
atomicAdd(&smem[rgb.x].x, 1);
atomicAdd(&smem[rgb.y].y, 1);
atomicAdd(&smem[rgb.z].z, 1);
__syncthreads();

//Global Histograms
atomicAdd(&out[t], smem[t].x); //R
atomicAdd(&out[t + NUM_BINS * 1], smem[t].y); //G
atomicAdd(&out[t + NUM_BINS * 2], smem[t].z); //B
}

```

After running the code we obtain these color distributions.

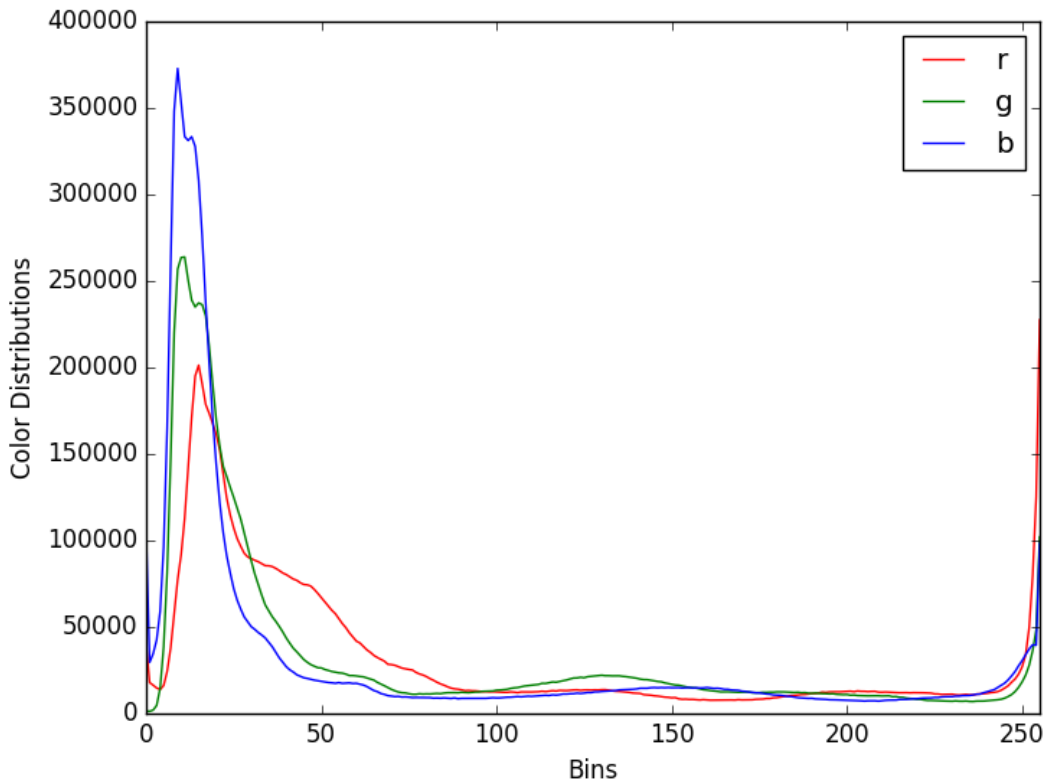


Figure 5: Color Histograms from the stained glass window

## 2 Sort

A very important algorithm in parallel computing is sorting an array. To do this we need a couple of subalgorithms. First we will explore segmented scan. Next we will discover an algorithm called compact. Finally, we will put these concepts together to illustrate a parallel sorting algorithm.

## 2.1 Segmented Scan

A segmented scan is exactly how it sounds – portions of an array scanned. One might be tempted to launch multiple scan kernels, however, this will be inefficient. A better method would be to combine the segments into an array (or keep them whole) and use a flagging array to mark the location of segments. Let's consider an example: An exclusive sum scan on the array

$$(1, 2, 3, 4, 5, 6, 7, 8)$$

yields

$$(0, 1, 3, 6, 10, 15, 21, 28)$$

Suppose we want to segment the array into 3 using the flagging array

$$(1, 0, 1, 0, 0, 1, 0, 0)$$

so the result is

$$(1, 2|3, 4, 5|6, 7, 8)$$

Then we scan the segments

$$(0, 1|0, 3, 7|0, 6, 13)$$

and thus we have the segmented scan algorithm.

## 2.2 Compact

Before we get to sort, let us discuss an algorithm called *compact*. Compact is an algorithm that partitions data. Compact takes an input array and compacts it into a smaller partition of that array. Generally, applying a compact is a good idea if we only wish to do computation on a subset of the array. As an example of compact suppose we have this array

$$[S_0, S_1, S_2, S_3, S_4, \dots]$$

this will be our *input*. Next we need a method for prescribing a partition, this is called a **predicate**. For this illustrative example we'll use the even index predicate. This generates a boolean array

$$[T, F, T, F, T, \dots]$$

Applying this predicate to the input array yields

$$[S_0, S_2, S_4, \dots]$$

a compacted array containing the even indexed elements of the input. To generate the output array as a **dense** array we need to compute what is called the *scatter address*. The scatter address is key to making this algorithm parallel. Given the set of predicates

$$[T, F, F, T, T, F, T, F]$$

we wish to compute the addresses

$$[0, -, -, 1, 2, -, 3, -]$$

We can do this by changing the boolean array into an array of 1's and 0's. Predicates:

$$[1, 0, 0, 1, 1, 0, 1, 0]$$

and generate

$$[0, 1, 1, 1, 2, 3, 3, 4]$$

We know this operation. This is a scan operation! Using these we can move to our sorting algorithm.

## 2.3 Radix Sort

The difficulty with a fast sorting algorithm is that most of them are serial. So we need to construct an efficient parallel algorithm for sorting an array. That is we need to

- Keep the hardware busy
- limit thread divergence
- use coalesced memory transactions

One of the best sorting algorithms to use on graphics processors is called the *Radix Sort*. With the Radix Sort, we operate on the bits of each element in our array of data. Here are the steps for a Radix Sort

1. Start with the least significant bit
2. Split input array into 2 sets based on this bit, otherwise preserve the order
3. Move to next most significant bit
4. Repeat until the last bit

### 2.3.1 Example

Suppose that we have the following array of unsigned integers:

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

We operate on these bit by bit

[000, 101, 010, 111, 001, 011, 110, 100]

The first step we will group together elements that have the same lowest order bit, revealing

[000, 010, 110, 100, 101, 111, 001, 011]

Notice that this is still unsorted, so we move to the next significant bit. In this step we group elements that have middle bit 0 and 1 together, otherwise keeping the order from the first stage.

[000, 100, 101, 001, 010, 110, 111, 011]

This array is still not completely sorted, so we move to the last bit and repeat the process

[000, 001, 010, 011, 100, 101, 110, 111]

which translates to

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

in unsigned integer.

### 2.3.2 Properties of Radix Sort

Notice that the work complexity of a Radix sort is  $\mathcal{O}(kn)$  where  $n$  is the size of the array and  $k$  is the number of bits in each element. Radix sort relies on grouping like bits together in each stage. This is where the compact algorithm comes in. For this sort, we have a specific predicate function  $(i \& 1) == 0$ . We can unwrap this predicate, the index  $i$  is operated on in binary by  $\&$  and uses 1 to look at the first bit. For example  $i = 4$  then  $i \& 1 = 4 \& 1 = 0100 \& 0001 = 0$ . Then an exclusive scan the values with 0 in their least significant bit to get the scatter addresses for the 0 bit compact array. Then inclusive scan the 1 bit predicates with the addition of the last 0 bit address to receive the 1 bit addresses.



### 2.3.3 CUDA Implementation

We will implement a Radix 2 Sort using a Hillis and Steele Scan algorithm. Recall that the Hillis and Steele scan is not work efficient, so for larger arrays, you should really use the Blelloch scan. The Hillis and Steele scan is like a moving reduction, it has  $n \log(n)$  work but does it in  $\log(n)$  stages. We will write this for a general *class*.

```
template <class T>
__device__ T plus_scan(T *x)
{
    __shared__ T temp[2*size]; // allocated on invocation
    int tid = threadIdx.x;
    int pout = 0, pin = 1;
    int n = size;
    // load input into shared memory.
    temp[tid] = x[tid];
    __syncthreads();
    for( int offset = 1; offset < n; offset <= 1 )
    {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;
        if (tid >= offset)
            temp[pout*n + tid] = temp[pin*n + tid] + temp[pin*n + tid - offset];
        else
            temp[pout*n + tid] = temp[pin*n + tid];
        __syncthreads();
    }
    x[tid] = temp[pout*n+tid]; // write output
    return x[tid];
}
```

This will allow us to scan for the scatter indices after compaction. Remember that the Hillis and Steele scan is inclusive. So instead of performing an exclusive scan on the 0s, we will perform an inclusive scan on the 1s, and induce the 0 scatter indices from there.

We will induce our kernel to partition the array *by bit*. Then, in a host function, we will loop through all the bits. First, let us take a look at the *partition by bit* kernel.

```
__global__ void partition_by_bit(unsigned int *values, unsigned int bit)
{
    unsigned int tid = threadIdx.x;
    unsigned int bsize = blockDim.x;
    unsigned int x_i = values[tid];
    __syncthreads();
    unsigned int p_i = (x_i >> bit) & 0b001; //value of x_i in binary at bits place predicate step!
    values[tid] = p_i;
    __syncthreads();

    unsigned int T_before = plus_scan(values); //scatter index before trues
    __syncthreads();
    unsigned int T_t = values[bsize - 1]; //total "trues"
    unsigned int F_t = bsize - T_t;
    __syncthreads();
    if(p_i)
    {
        values[T_before - 1 + F_t] = x_i;
        __syncthreads();
    }
    else
    {
        values[tid - T_before] = x_i;
        __syncthreads();
    }
}
```

Notice that the variable `p_i` contains the predicate for compaction. We then store the predicates in the original array, while the values are stored in the temporary variable `x_i`. Because we are using this style of scatter, we must synchronize the threads as we go along. Notice that this Radix Sort is only written for one block. We can invoke the strategy for scan to extend this to arbitrary sizes. Now that we have a main kernel for the operation let's look at our `radix_sort` function.

```
void radix_sort(unsigned int *values)
{
    unsigned int *d_vals;
    unsigned int bit;
    cudaMalloc(&d_vals, size*sizeof(unsigned int));
}
```

```

cudaMemcpy(d_vals, values, size*sizeof(unsigned int), cudaMemcpyHostToDevice);
for(bit = 0; bit < NUM_BITS; bit++)
{
    partition_by_bit<<<1,size>>>(d_vals, bit);
    cudaDeviceSynchronize();
}
cudaMemcpy(values, d_vals, size*sizeof(unsigned int), cudaMemcpyDeviceToHost);
cudaFree(d_vals);
}

```

Combining these functions and kernels together, we create a program for the Radix Sort. The Radix Sort is not the only sorting algorithm to work in parallel, the even-odd (brick sort) also works in parallel, however, it is not as efficient as the Radix Sort.

### 3 Sparse Matrices

We've looked into operations regarding *dense* matrices, however, a significant number of operations can be saved when a matrix is **sparse**. A sparse matrix is a matrix which contains many elements that are identically zero. Let's look into an example to illustrate a concept called **Compressed Sparse Row**(CSR). Suppose we have the matrix

$$\mathbf{A} = \begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix}$$

The CSR form will decompose a sparse matrix into three vectors, a value vector, which contains the non-zero values of the matrix, a column vector which gives the column number the value resides in, and a row pointer vector, this vector contains the index of an element of the value vector that starts the next row in the matrix **A**.

$$\begin{aligned}
\mathbf{v} &= [a, b, c, d, e, f] \\
\mathbf{c} &= [0, 2, 0, 1, 2, 2] \\
\mathbf{rp} &= [0, 2, 5]
\end{aligned}$$

For an example this small it may seem silly to represent the matrix in this fashion. However, in the case of large sparse matrices we can reduce a the operation count from  $N$  which is very large, to  $N - Z$  where  $Z$  is the number of zeros in the sparse matrix.

#### 3.1 Sparse Matrix Vector Multiplication

Suppose we wish to do the following multiplication

$$\begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

using sparse matrix operations

Sparse matrix-vector multiplication is pretty simple when we break it into steps. In general we follow this plan

1. Create Segmented representation from value and row pointer vectors

$$[a, b \mid c, d, e \mid f]$$

2. Gather vector values using column

$$[x, z, x, y, z, z]$$

3. Pairwise multiply 1 and 2

$$[ax, bz \mid cx, dy, ez \mid fz]$$

#### 4. Inclusive segmented sum scan

$$[ax + bz \mid cx + dy + ez \mid fz]$$

Together with all these steps we construct:

$$\begin{pmatrix} a & 0 & b \\ c & d & e \\ 0 & 0 & f \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + \cancel{0y} + bz \\ cx + dy + ez \\ \cancel{0x} + \cancel{0y} + fz \end{pmatrix}$$

where we were able to disregard the 0 elements multiply/adds. Here each row in the out vector correspond to the segments in the resulting vector from our plan.

### 3.2 CUDA Implementation

Let's implement this in CUDA C. For this application let's use structs.

```
typedef struct
{
    float *val;
    int *col;
    int *rowptr;
    int ncol;
    int nrow;
} csrMatrix;
```

Lets follow our plan:

```
__global__ void csr_mult(const csrMatrix A, const float *x, float *b) // Ax = b
{
    // Have kernel go over each row this will give step 1
    int row = blockDim.x*blockIdx.x + threadIdx.x;

    if(row < A.nrow){
        float dot = 0.0f;

        int row_start = A.rowptr[row];
        int row_end = A.rowptr[row+1];

        for(int jj = row_start; jj < row_end; jj++){
            dot += A.val[jj] * x[A.col[jj]]; // Step 2 3 and 4
        }
        b[row] = dot;
    }
}
```

In this implementation we have the global thread index spread cover the rows. This implicitly creates the segmented representation for the value array. We index on `x[A.col[jj]]` to gather the vector using the column values. Then steps 3 and 4 are combined in the loop from `row_start` to `row_end`.

### 3.3 Comparison with Dense Matrix Vector Product

We will test the CSR SpMV against the a dense matrix vector product. For this test we'll examine a sparse matrix of size 1024×1024. We can visualize this matrix, in the next figure:

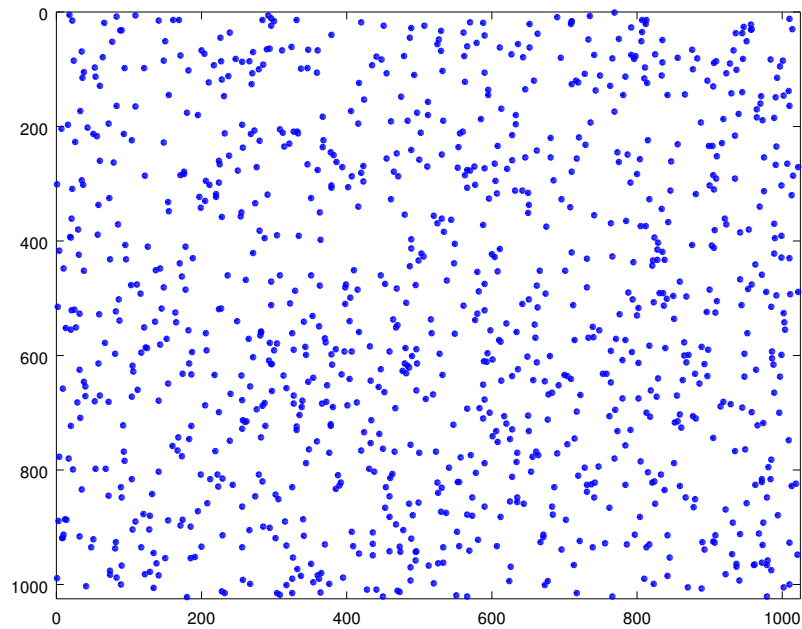


Figure 6: Sparse Matrix of size 1024 by 1024

Here the blank spaces denote the 0 values, and the colored points are values that are randomly generated between 0 and 1. This matrix has a density of 0.1, so only 10% of the values are non-zero.

Timing the programs we find that the dense matrix vector multiplication took 0.08985ms vs 0.016448ms using the CSR sparse matrix vector multiplication. By using the CSR representation we can leverage the sparsity and reduce computation time.