Lab 6

George Onwubuya

November 15, 2018

1 Histogram

1.1 Output

```
Setting up the problem...0.002114 s
    Input size = 100000
    Number of bins = 4096
Allocating device variables...0.000280 s
Copying data from host to device...0.000272 s
Launching kernel...0.000452 s
Copying data from device to host...0.000039 s
Verifying results...TEST PASSED
```

```
Setting up the problem...0.003954 s
    Input size = 200000
    Number of bins = 4096
Allocating device variables...0.000304 s
Copying data from host to device...0.000484 s
Launching kernel...0.000466 s
Copying data from device to host...0.000040 s
Verifying results...TEST PASSED
```

```
Setting up the problem...0.008680 s
    Input size = 400000
    Number of bins = 4096
Allocating device variables...0.000328 s
Copying data from host to device...0.000805 s
Launching kernel...0.000458 s
Copying data from device to host...0.000039 s
Verifying results...TEST PASSED
```

```
Setting up the problem...0.017020 s
    Input size = 800000
    Number of bins = 4096
Allocating device variables...0.000258 s
Copying data from host to device...0.001406 s
Launching kernel...0.000510 s
Copying data from device to host...0.000043 s
Verifying results...TEST PASSED


Setting up the problem...0.022502 s
    Input size = 1000000
    Number of bins = 4096
Allocating device variables...0.000264 s
Copying data from host to device...0.001899 s
Launching kernel...0.000525 s
Copying data from device to host...0.000041 s
Verifying results...TEST PASSED


Setting up the problem...0.033799 s
    Input size = 1600000
    Number of bins = 4096
Allocating device variables...0.000311 s
Copying data from host to device...0.002794 s
Launching kernel...0.000543 s
Copying data from device to host...0.000040 s
Verifying results...TEST PASSED
```

1.2 Performance Analysis (4096 Bins)

Execution Time (seconds) for Each Process					
Elements(m)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
100000	0.002114	0.000280	0.000272	0.000452	0.000039
200000	0.003954	0.000304	0.000484	0.000466	0.000040
400000	0.008680	0.000328	0.000805	0.000458	0.000039
800000	0.017020	0.000258	0.001406	0.000510	0.000043
1000000	0.022502	0.000264	0.001899	0.000525	0.000041
1600000	0.033799	0.000311	0.002794	0.000543	0.000040

1.2.1 Comments

When setting up the shell script file, the number of elements were varied but I chose to use the same number of bins, 4096. The time taken to allocate device variables and copy from device to host are approximately the same. This is because the same number of variables are allocated regardless of the size of the array of input elements. The times taken to set up the problem, copy from host to device and launching the kernel are all directly proportional to the size of the input array.

2 Histogram (Optimized)

2.1 Output

```
Setting up the problem...0.002145 s
    Input size = 100000
    Number of bins = 4096
Allocating device variables...0.000262 s
Copying data from host to device...0.000236 s
Launching kernel...0.000498 s
Copying data from device to host...0.000036 s
Verifying results...TEST PASSED


Setting up the problem...0.004105 s
    Input size = 200000
    Number of bins = 4096
Allocating device variables...0.000303 s
Copying data from host to device...0.000387 s
Launching kernel...0.000504 s
Copying data from device to host...0.000042 s
Verifying results...TEST PASSED


Setting up the problem...0.008779 s
    Input size = 400000
    Number of bins = 4096
Allocating device variables...0.000259 s
Copying data from host to device...0.000713 s
Launching kernel...0.000528 s
Copying data from device to host...0.000041 s
Verifying results...TEST PASSED


Setting up the problem...0.016875 s
    Input size = 800000
    Number of bins = 4096
Allocating device variables...0.000261 s
Copying data from host to device...0.001438 s
```

```
Launching kernel...0.000560 s
Copying data from device to host...0.000041 s
Verifying results...TEST PASSED


Setting up the problem...0.021037 s
    Input size = 1000000
    Number of bins = 4096
Allocating device variables...0.000290 s
Copying data from host to device...0.001656 s
Launching kernel...0.000589 s
Copying data from device to host...0.000040 s
Verifying results...TEST PASSED


Setting up the problem...0.033860 s
    Input size = 1600000
    Number of bins = 4096
Allocating device variables...0.000284 s
Copying data from host to device...0.002730 s
Launching kernel...0.000629 s
Copying data from device to host...0.000040 s
Verifying results...TEST PASSED
```

2.2 Performance Analysis

2.2.1 Array Size

Execution Time (seconds) for Each Process					
Elements(m)	Setting Up	DeviceVar	HostToDevice	Kernel	DeviceToHost
100000	0.002145	0.000262	0.000236	0.000498	0.000036
200000	0.004105	0.000303	0.000387	0.000504	0.000042
400000	0.008779	0.000259	0.000713	0.000528	0.000041
800000	0.016875	0.000261	0.001438	0.000560	0.000041
1000000	0.021037	0.000290	0.001656	0.000589	0.000040
1600000	0.033860	0.000284	0.002730	0.000629	0.000040

2.2.2 Comments

The execution times relate to the same way to the initial histogram kernel that had no optimization.

2.3 Answers

2.3.1 Description of Optimization

To optimize the histogram kernel, the kernel code was written such that a location in shared memory would only be updated once. Three new variables were introduced to the kernel code. The first two variables current index and previous index were meant to keep track of the elements from the input array. The current index and previous were compared to check if the same element was still being evaluated because of possible repetitions in data. If they were found to be the same, a third variable accumulator would store the number of times this element occurred in succession in the array . However if the elements were not the same, the value in the accumulator would be written to a location in shared memory as indicated by the previous index, the previous index would then be set to the current index and the value in the accumulator would be set to one to indicate a shift to the next element. The accumulator and the previous index variables were initialized to zero outside the while loop and the current index variable was set to the current input element under evaluation by the histogram kernel code.

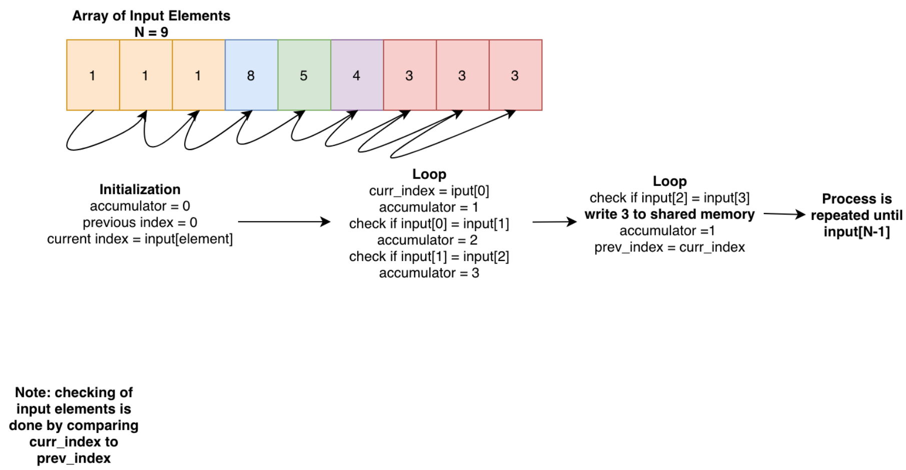


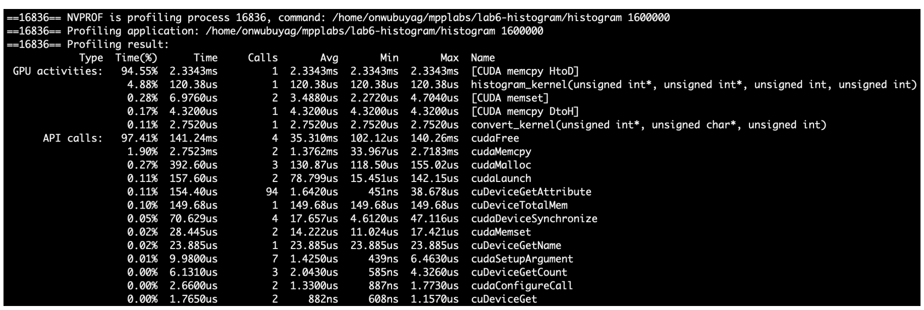
Figure 1: An illustration of what the kernel code does

2.3.2 Difficulties with Optimization

A couple of difficulties I experienced during this optimization were found in not analyzing the the second while loop structure of the code that is used to update the value in the accumulator to the shared memory location. I examined how the loop would behave at the beginning of the and during the middle part of the execution and neglected to check what happens when the loop ends. As a result my code was incomplete and was unable to simulate past the first bin. I also made the mistake of attempting to carry out the optimization without first checking if the code was properly written out.

2.3.3 Change in Execution Time

Kernel Times in microseconds		
Elements(m)	Non-Optimized	Optimized
100000	11.776	76.414
200000	18.367	82.398
400000	32.614	99.646
800000	63.039	129.12
1000000	77.356	144.09
1600000	120.38	190.17



```
==16836== NVPROF is profiling process 16836, command: /home/onwubuyag/mpplabs/lab6-histogram/histogram 1600000
==16836== Profiling application: /home/onwubuyag/mpplabs/lab6-histogram/histogram 1600000
==16836== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   94.55%  2.3343ms         1  2.3343ms  2.3343ms  2.3343ms  [CUDA memcpy HtoD]
                    4.88%  120.38us         1  120.38us  120.38us  120.38us  histogram_kernel(unsigned int*, unsigned int*, unsigned int, unsigned int)
                    0.28%  6.9760us         2  3.4880us  2.2720us  4.7040us  [CUDA memset]
                    0.17%  4.3200us         1  4.3200us  4.3200us  4.3200us  [CUDA memcpy DtoH]
                    0.11%  2.7520us         1  2.7520us  2.7520us  2.7520us  convert_kernel(unsigned int*, unsigned char*, unsigned int)
      API calls:   97.41%  141.24ms         4  35.310ms  102.12us  140.26ms  cudaFree
                    1.90%  2.7523ms         2  1.3762ms  33.967us  2.7183ms  cudaMemcpy
                    0.27%  392.60us         3  130.87us  118.50us  155.02us  cudaMalloc
                    0.11%  157.60us         2  78.799us  15.451us  142.15us  cudaLaunch
                    0.11%  154.40us        94  1.6420us     451ns  38.678us  cuDeviceGetAttribute
                    0.10%  149.68us         1  149.68us  149.68us  149.68us  cuDeviceTotalMem
                    0.05%  70.629us         4  17.657us  4.6120us  47.116us  cudaDeviceSynchronize
                    0.02%  28.445us         2  14.222us  11.024us  17.421us  cudaMemset
                    0.02%  23.885us         1  23.885us  23.885us  23.885us  cuDeviceGetName
                    0.01%  9.9800us         7  1.4250us     439ns  6.4630us  cudaSetupArgument
                    0.00%  6.1310us         3  2.0430us     585ns  4.3260us  cuDeviceGetCount
                    0.00%  2.6600us         2  1.3300us     887ns  1.7730us  cudaConfigureCall
                    0.00%  1.7650us         2     882ns     608ns  1.1570us  cuDeviceGet
```

Figure 2: Nvidia profiler result for non-optimized histogram kernel for m = 1600000

```
==17003== NVPROF is profiling process 17003, command: /home/onwubuyag/mpplabs/lab6-histogram/histogram 1600000
==17003== Profiling application: /home/onwubuyag/mpplabs/lab6-histogram/histogram 1600000
==17003== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   91.79%  2.2817ms         1  2.2817ms  2.2817ms  2.2817ms  [CUDA memcpy HtoD]
                    7.65%  190.17us         1  190.17us  190.17us  190.17us  histogram_kernel_optimized(unsigned int*, unsigned int*, unsigned int, unsigned i
nt)
                    0.28%  6.8800us         2  3.4400us  2.2400us  4.6400us  [CUDA memset]
                    0.18%  4.4800us         1  4.4800us  4.4800us  4.4800us  [CUDA memcpy DtoH]
                    0.11%  2.6560us         1  2.6560us  2.6560us  2.6560us  convert_kernel(unsigned int*, unsigned char*, unsigned int)
      API calls:   97.40%  142.05ms         4  35.513ms  104.83us  140.98ms  cudaFree
                    1.84%  2.6820ms         2  1.3410ms  33.133us  2.6489ms  cudaMemcpy
                    0.27%  387.24us         3  129.08us  103.39us  151.38us  cudaMalloc
                    0.14%  205.15us         1  205.15us  205.15us  205.15us  cuDeviceTotalMem
                    0.14%  202.01us        94  2.1490us     442ns  56.735us  cuDeviceGetAttribute
                    0.11%  159.70us         2  79.850us  16.012us  143.69us  cudaLaunch
                    0.05%  76.425us         4  19.106us  4.9700us  51.634us  cudaDeviceSynchronize
                    0.02%  29.115us         2  14.557us  10.949us  18.166us  cudaMemset
                    0.02%  28.463us         1  28.463us  28.463us  28.463us  cuDeviceGetName
                    0.01%  10.258us         7  1.4650us     452ns  6.7510us  cudaSetupArgument
                    0.00%  5.9090us         3  1.9690us     527ns  4.3270us  cuDeviceGetCount
                    0.00%  2.7080us         2  1.3540us     907ns  1.8010us  cudaConfigureCall
                    0.00%  1.6430us         2     821ns     545ns  1.0980us  cuDeviceGet
```

Figure 3: Nvidia profiler result for optimized histogram kernel for m = 1600000

2.3.4 Explanation for Optimization

The optimization increased the amount of time spent executing the kernel. I suspect that might be the result of the input array being generated with less repetition than originally anticipated. The contention rate was low and hence the addition of more variables would only serve to increase the execution time.

2.4 Kernel

```
/******************************************************************************
 *cr
 *cr            (C) Copyright 2010 The Board of Trustees of the
 *cr                        University of Illinois
 *cr                         All Rights Reserved
 *cr
 
 ↪   ******************************************************************************/

// Define your kernels in this file you may use more than one
↪   kernel if you
// need to
__global__ void histogram_kernel(unsigned int* input, unsigned
↪   int* bins,
    unsigned int num_elements, unsigned int num_bins){

        extern __shared__ unsigned int bins_s[];

        //Shared Memory
        int thid = threadIdx.x;
        while(thid < num_bins){

                bins_s[thid] = 0u;
                thid += blockDim.x;
```

```
        }
        __syncthreads();


        //Histogram calculation
        unsigned int element = blockIdx.x * blockDim.x + threadIdx.x;

        while(element < num_elements){

                atomicAdd(&(bins_s[input[element]]), 1);
                element += blockDim.x * gridDim.x;
        }
        __syncthreads();

        //Global Memory
        thid = threadIdx.x;
        while(thid < num_bins){

                atomicAdd(&(bins[thid]), bins_s[thid]);
                thid += blockDim.x;
        }
}


__global__ void histogram_kernel_optimized(unsigned int* input, unsigned int* bins,
    unsigned int num_elements, unsigned int num_bins) {

              // INSERT CODE HERE
        extern __shared__ unsigned int bins_s[];

        //Shared memory
        int thid = threadIdx.x;
        while ( thid < num_bins){

                bins_s[thid] = 0u;
                thid += blockDim.x;
        }
        __syncthreads();

        //Histogram calculation
        unsigned int element = blockIdx.x * blockDim.x + threadIdx.x;
        unsigned int accumulator = 0;
        unsigned int prev_index = 0;
```

```

        while(element < num_elements){

                unsigned int curr_index = input[element];

                if(curr_index != prev_index){

                        atomicAdd(&(bins_s[prev_index]), accumulator);
                        accumulator = 1;
                        prev_index = curr_index;

                }

                else{
                        accumulator++;
                }
                element += blockDim.x * gridDim.x;
        }
        if(accumulator > 0){
                atomicAdd(&(bins_s[prev_index]), accumulator);
        }
        __syncthreads();

        //Global memory
        thid = threadIdx.x;
        while(thid < num_bins){

                atomicAdd(&(bins[thid]), bins_s[thid]);
                thid += blockDim.x;
        }

}

__global__ void convert_kernel(unsigned int *bins32, uint8_t *bins8,
    unsigned int num_bins) {

      // INSERT CODE HERE
        int thid = blockIdx.x * blockDim.x + threadIdx.x;

        while (thid < num_bins){

                //Use local  register value (avoids copying from global twice)
                unsigned int reg_bin = bins32[thid];
```

```
                if(reg_bin > 255){
                        bins8[thid] = 255u;
                }

                else{
                        bins8[thid] = (uint8_t) reg_bin;
                }
                thid += blockDim.x * gridDim.x;
        }

}

/******************************************************************************
Setup and invoke your kernel(s) in this function. You may also allocate more
GPU memory if you need to
*******************************************************************************/
void histogram(unsigned int* input, uint8_t* bins, unsigned int num_elements,
        unsigned int num_bins) {

    // Create 32 bit bins
    unsigned int *bins32;
    cudaMalloc((void**)&bins32, num_bins * sizeof(unsigned int));
    cudaMemset(bins32, 0, num_bins * sizeof(unsigned int));

    // Launch histogram kernel using 32-bit bins
    dim3 dim_grid, dim_block;
    dim_block.x = 512; dim_block.y = dim_block.z = 1;
    dim_grid.x = 30; dim_grid.y = dim_grid.z = 1;

    //Comment out the kernel not used
    //histogram_kernel<<<dim_grid, dim_block, num_bins*sizeof(unsigned int)>>>
       // (input, bins32, num_elements, num_bins);
   histogram_kernel_optimized<<<dim_grid, dim_block, num_bins*sizeof(unsigned int)>>>
        (input, bins32, num_elements, num_bins);

    // Convert 32-bit bins into 8-bit bins
    dim_block.x = 512;
    dim_grid.x = (num_bins - 1)/dim_block.x + 1;
    convert_kernel<<<dim_grid, dim_block>>>(bins32, bins, num_bins);

```

```
    // Free allocated device memory
    cudaFree(bins32);

}
```