EE-147 Lab 3
Histogram

Cody Simons
861177050

May 10, 2018

1 Introduction

In this lab we learn how to use the atomicAdd operation. Our use case is calculating a histogram for a random vector. It is important that we use the atomicAdd operation rather than normal add to increment, because it ensures that no other may access the values in the bins while another process is working with them. This prevents a lot of undefined behaviors that would be prevalent if memory read-modify-writes were randomly interleaved.

2 Questions

1. Use visual profiler to report relevant statistics (e.g. utilization and memory hierarchy related) about the execution of your kernels. Did you find any surprising results?
   I was unable to work with the visual profiler, but using the command line version I was able to find some interesting results. There are no global store request which is odd, because I would expect that when all the private histograms are added together that would generate gobal store requests.

```
==13880== Profiling application: ./histogram
==13880== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min
       Max  Name
 GPU activities:   69.12%  1.6636ms         1  1.6636ms  1.6636ms  1.6636
    ms  [CUDA memcpy HtoD]
                   30.51%  734.34us         1  734.34us  734.34us  734.34
                 us  histogram_kernel(unsigned int*, unsigned int*,
                 unsigned int, unsigned int)
                    0.19%  4.5440us         1  4.5440us  4.5440us  4.5440
                   us  [CUDA memset]
                    0.19%  4.5120us         1  4.5120us  4.5120us  4.5120
                   us  [CUDA memcpy DtoH]
      API calls:   98.23%  211.18ms         2  105.59ms  130.71us  211.05
         ms  cudaMalloc
                    0.86%  1.8527ms         2  926.33us  40.074us  1.8126
                   ms  cudaMemcpy
                    0.41%  879.03us         4  219.76us  3.7380us  743.57
                   us  cudaDeviceSynchronize
                    0.33%  703.52us         2  351.76us  119.70us  583.82
                   us  cudaFree
                    0.07%  148.34us        94  1.5780us     187ns  57.658
                   us  cuDeviceGetAttribute
                    0.06%  136.36us         1  136.36us  136.36us  136.36
                   us  cuDeviceTotalMem
```

```
                  0.02%   45.102us          1   45.102us   45.102us   45.102
                     us   cudaLaunch
                  0.01%   21.961us          1   21.961us   21.961us   21.961
                     us   cuDeviceGetName
                  0.01%   15.589us          1   15.589us   15.589us   15.589
                     us   cudaMemset
                  0.00%   6.3150us          4   1.5780us      179ns   5.3240
                     us   cudaSetupArgument
                  0.00%   2.8810us          2   1.4400us      318ns   2.5630
                     us   cuDeviceGet
                  0.00%   2.8700us          3      956ns      200ns   1.9420
                     us   cuDeviceGetCount
                  0.00%      887ns          1      887ns      887ns      887
                     ns   cudaConfigureCall
```

2. What, if any, limitations are there on m and n for your implementation? Explain these limitations and how you may overcome them with a different implementation.
   The lower bound would be that you need at least one bin and one element in your input. This can't really be changed due the nature of the problem. The upper bound would be limited by two different factors. The number of bins would have to be less than the maximum amount of shared memory divided by the size of an unsigned int. The number of elements in the input vector has a similar constraint, but based on the size of global memory. Unfortunately you can't change the bin size limit because atomicAdd only supports 32 bit values. You could increase the number of elements in the input by using a smaller data size.

3 Code

Below is my kernel for generating the histogram. Each SM has its own private histogram that is shared between all the thread in the warp. These are all then summed together to get the final result.

3.1 Kernel.cu

```
/******************************************************************************
 *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

// INSERT KERNEL(S) HERE
#include <stdio.h>

__global__ void histogram_kernel(unsigned int *inputs, unsigned int *bins, unsigned int num_elements, unsig
    extern __shared__ unsigned int private_histogram[];

    int bx = blockIdx.x;  int bd = blockDim.x;
    int tx = threadIdx.x; int gd = gridDim.x;

    int i=0;
    while(i*bd+tx < num_bins) {
        private_histogram[i*bd+tx] = 0;
        i++;
    }
    __syncthreads();

    int index = bx*bd+tx;
    int stride = bd*gd;
    i=0;
    while(i*stride+index < num_elements) {
        atomicAdd(private_histogram+inputs[i*stride+index],1);
        i++;
    }
    __syncthreads();

    i=0;
    while(i*bd+tx < num_bins) {
        atomicAdd(bins+i*bd+tx,private_histogram[i*bd+tx]);
        i++;
    }

    return;
}

/****************************************************************************
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, unsigned int* bins, unsigned int num_elements,
        unsigned int num_bins) {

    // INSERT CODE HERE
    dim3 gridDim(30,1,1);
    dim3 blockDim(32,1,1);

    histogram_kernel<<<gridDim, blockDim, num_bins*sizeof(unsigned int)>>>(input,bins,num_elements,num_bins
}
```

4 Program Output

```
bender /home/eemaj/csimons/EE-147/Lab 3 $ ./histogram

Setting up the problem...0.014242 s
    Input size = 1000000
    Number of bins = 4096
Allocating device variables...0.319838 s
Copying data from host to device...0.001842 s
Launching kernel...0.000785 s
Copying data from device to host...0.000043 s
Verifying results...TEST PASSED
```