Report Lab 3

Andres Calderon - SID:861243796

November 13, 2015

1 Code

The following code was used to complete the report:

1.1 Reduction

1.1.1 kernel.cu

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

#define BLOCK_SIZE 512

__global__ void reduction(float *out, float *in, unsigned size)
{
    /********************************************************************
    Load a segment of the input vector into shared memory
    Traverse the reduction tree
    Write the computed sum to the output vector at the correct index
    ********************************************************************/

  // Declare an array for share memory...
  __shared__ float partialSum[2 * BLOCK_SIZE];

  // Initialize some variables to access data...
  unsigned int t = threadIdx.x;
  unsigned int start = 2 * blockIdx.x * blockDim.x;

  // Validation to avoid load data outside of the input array...
  if(start + t < size)
    partialSum[t] = in[start + t];
  else
    partialSum[t] = 0.0f;

  // Same validation for the other position...
  if(start + blockDim.x + t < size)
    partialSum[blockDim.x + t] = in[start + blockDim.x + t];
  else
    partialSum[blockDim.x + t] = 0.0f;

  // Iterate through share memory to compute the sum...
  for (int stride = blockDim.x; stride > 0; stride /= 2){
    __syncthreads();  // Synchronize the share memory load and each iteration...
    if (t < stride)
      partialSum[t] += partialSum[t + stride];
  }
  // Do not forget to synchronize last iteration...
  __syncthreads();

```

```
  // Copy back the result...
  out[blockIdx.x] = partialSum[0];
}
```

There are not significant changes in the other files.

1.2 Prefix-scan

1.2.1 kernel.cu

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

#define BLOCK_SIZE 512

// Define your kernels in this file you may use more than one kernel if you need to
__global__ void scan(float *out, float *in, unsigned size){
  // Declare share memory space...
  __shared__ float section[2 * BLOCK_SIZE];
  int t = blockDim.x * blockIdx.x + threadIdx.x;

  // Load data in memory share taking into account that exclusive scan is required...
  if(t < size)
    if(t == 0)
      section[0] = 0.0f;
    else
      section[threadIdx.x] = in[t - 1];
  __syncthreads();

  // First phase: reduction step...
  for(int stride = 1; stride <= BLOCK_SIZE; stride = stride * 2){
    int index = (threadIdx.x + 1) * stride * 2 - 1;
    if(index < 2 * BLOCK_SIZE)
      section[index] += section[index - stride];
    __syncthreads();
  }

  // Second phase: post-scan step...
  for(int stride = BLOCK_SIZE / 2; stride > 0; stride /= 2){
    int index = (threadIdx.x + 1) * stride * 2 - 1;
    if(index + stride < 2 * BLOCK_SIZE)
      section[index + stride] += section[index];
    __syncthreads();
  }

  // Copy back to global memory...
  if(t < size)
    out[t] = section[threadIdx.x];
}

// This function takes the output of the scan function and add the accumulated sum (stored in n) to positions in each block...
__global__ void post(float *out, float *n, unsigned size){
  int t = blockDim.x * blockIdx.x + threadIdx.x;

  out[t] += n[t / BLOCK_SIZE];
}

/******************************************************************************
Setup and invoke your kernel(s) in this function. You may also allocate more
GPU memory if you need to
*******************************************************************************/
// Functions to define block a grid dimensions and lunch the kernels...
void preScan(float *out, float *in, unsigned size){
  dim3 dim_block(BLOCK_SIZE, 1, 1);
```

```
  dim3 dim_grid(size/BLOCK_SIZE + 1, 1, 1);
  scan<<<dim_grid, dim_block>>>(out, in, size);
}

void postScan(float *out, float *n, unsigned size){
  dim3 dim_block(BLOCK_SIZE, 1, 1);
  dim3 dim_grid(size/BLOCK_SIZE + 1, 1, 1);
  post<<<dim_grid, dim_block>>>(out, n, size);
}
```

1.2.2 main.cu

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

#include <stdio.h>
#include "support.h"
#include "kernel.cu"

int main(int argc, char* argv[])
{
  Timer timer;
  // Initialize host variables
  printf("\nSetting up the problem..."); fflush(stdout);
  startTime(&timer);

  float *in_h, *out_h;
  float *in_d, *out_d;
  unsigned num_elements;
  cudaError_t cuda_ret;

  // Allocate and initialize input vector
  if(argc == 1) {
    num_elements = 1000000;
  } else if(argc == 2) {
    num_elements = atoi(argv[1]);
  } else {
    printf("\n  Invalid input parameters!"
      "\n  Usage: ./prefix-scan     # Input of size 1,000,000 is used"
      "\n  Usage: ./prefix-scan <m> # Input of size m is used"
      "\n");
    exit(0);
  }
  initVector(&in_h, num_elements);

  // Allocate and initialize output vector
  out_h = (float*)calloc(num_elements, sizeof(float));
  if(out_h == NULL) FATAL("Unable to allocate host");

  stopTime(&timer); printf("%f s\n", elapsedTime(timer));
  printf("Input size = %u\n", num_elements);

  // Allocate device variables
  printf("Allocating device variables..."); fflush(stdout);
  startTime(&timer);
  cuda_ret = cudaMalloc((void**)&in_d, num_elements*sizeof(float));
  if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");
  cuda_ret = cudaMalloc((void**)&out_d, num_elements*sizeof(float));
  if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");
  cudaDeviceSynchronize();
  stopTime(&timer); printf("%f s\n", elapsedTime(timer));

  // Copy host variables to device
  printf("Copying data from host to device..."); fflush(stdout);
  startTime(&timer);
```

```
  cuda_ret = cudaMemcpy(in_d, in_h, num_elements*sizeof(float), cudaMemcpyHostToDevice);
  if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to the device");
  cuda_ret = cudaMemset(out_d, 0, num_elements*sizeof(float));
  if(cuda_ret != cudaSuccess) FATAL("Unable to set device memory");
  cudaDeviceSynchronize();
  stopTime(&timer); printf("%f s\n", elapsedTime(timer));

  // Launch kernel
  printf("Launching kernel..."); fflush(stdout);
  startTime(&timer);
  // Set up and invoke your kernel inside the preScan function, which is in kernel.cu
  preScan(out_d, in_d, num_elements);
  cuda_ret = cudaDeviceSynchronize();
  if(cuda_ret != cudaSuccess) FATAL("Unable to launch/execute kernel");
  stopTime(&timer); printf("%f s\n", elapsedTime(timer));

  // Copy device variables from host
  printf("Copying data from device to host..."); fflush(stdout);
  startTime(&timer);
  cuda_ret = cudaMemcpy(out_h, out_d, num_elements*sizeof(float), cudaMemcpyDeviceToHost);
  if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to host");
  cudaDeviceSynchronize();
  stopTime(&timer); printf("%f s\n", elapsedTime(timer));

  /* Now we need to traverse the out_h array to extract the last accumulated sum for each block and then add them to its corresponding positions... */

  // Let's declare a auxiliar array to store the accumulated sums of each block...
  float *partial_h, *partial_d;
  // Allocate memory for the new array...
  partial_h = (float *) malloc((num_elements/BLOCK_SIZE + 1) * sizeof(float));
  // We do not add nothig to the first block
  partial_h[0] = 0;
  int n = 1;
  // Iterate through the array out_h extracting partial sums...
  for(int i = BLOCK_SIZE - 1; i < num_elements; i += BLOCK_SIZE){
    partial_h[n] = partial_h[n - 1] + out_h[i];
    n++;
  }
  // Print the partial_h array just for debugging purposes...
  if((num_elements/BLOCK_SIZE + 1) <= 10){
    for(int i = 0; i < n; i ++){
      printf("\nPARTIAL[%d] = %0.3f", i, partial_h[i]);
    }
    printf("\n");
  }
  // Allocate memory and copy the array in the device...
  cuda_ret = cudaMalloc((void**)&partial_d, (num_elements/BLOCK_SIZE + 1) * sizeof(float));
  if(cuda_ret != cudaSuccess) FATAL("Unable to allocate device memory");
  cuda_ret = cudaMemcpy(partial_d, partial_h, (num_elements/BLOCK_SIZE + 1) * sizeof(float), cudaMemcpyHostToDevice);
  if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to the device");

  // Invoke a new kernel (it is in kernel.cu) where each thread add the accumulated sum in partial_d to its correponding position in out_d...
  postScan(out_d, partial_d, num_elements);
  cuda_ret = cudaDeviceSynchronize();
  if(cuda_ret != cudaSuccess) FATAL("Unable to launch/execute kernel");

  // Copy back the new results...
  cuda_ret = cudaMemcpy(out_h, out_d, num_elements*sizeof(float), cudaMemcpyDeviceToHost);
  if(cuda_ret != cudaSuccess) FATAL("Unable to copy memory to host");

  // Verify correctness
  printf("Verifying results..."); fflush(stdout);
  verify(in_h, out_h, num_elements);

  // Printing results (just for debugging purposes)...
  if(num_elements <= 100){
    printf("\nPrinting IN (%d elements)...\n", num_elements);
```

```
    for(int i = 0; i < num_elements; i++){
      printf("%0.3f ", in_h[i]);
    }
    printf("\n");

    printf("\nPrinting OUT (%d elements)...\n", num_elements);
    for(int i = 0; i < num_elements; i++){
      printf("%0.3f ", out_h[i]);
    }
    printf("\n");
  }

  // Free memory
  cudaFree(in_d); cudaFree(out_d); cudaFree(partial_d);
  free(in_h); free(out_h); free(partial_h);

  return 0;
}
```

Full code and other materials are available at [1].

2 Answers to Questions

1. Use visual profiler to report relevant statistics about the execution of your kernels. Did you find any surprising results?

   Figure 1 shows the summary of the profile analysis over the `reduction` kernel. One important metric that should be noted is the high performance for *Global Load and Global Store Efficiency* (highlighted by 1 in the figures). Those values are significantly larger in comparison with the `prefix-scan` kernel (figure 2). The strategy for avoiding uncoalesced memory access implemented in the `reduction` kernel proves to be more efficient.

   Another important metric is the *Low Compute / Memcpy Efficiency* in both kernels (pointed by 2 in both figures). As it can be seen, the time spent to copy data between the host and the device is considerably larger that the time spent in computation. The NVVP Help proposes asynchronous and overlapping transfers using `cudaMemcpyAsync()` function as a possible alternative to address the problem [5] [6].

2. For each of reduction and prefix scan, suggest one approach to speed up your implementation.

   For reduction, it is possible to chunk input data to perform more `memcpy` operations and take advantage of the time of each of them to perform computation using streams. So, when the program is copying some data, it already can compute some of them asynchronously [5] [6]. In this way, it is possible to overlap the copy and compute stages.

   For prefix-scan, [4] explore an alternative in order to avoid share memory bank conflicts. The main idea is to avoid most of the bank conflicts by adding a variable amount of padding to each shared memory array index they compute. Specifically, the value of the index divided by the number of shared memory banks is added to each index. A gentle explanation of the algorithm is available in [3]. However, as [2] argues, this approach (although offers an increase in time response) could be more computationally expensive.

References

[1] Andres Calderon. *GitHub Personal Repository*, 2015. `https://github.com/aocalderon/PhD/tree/master/Y1Q1/GPU/lab3`.

[2] Wen-Mei Hwu. *Parallel Computation Patterns, More on Parallel Scan - Heterogeneous Parallel Programming*. Coursera Course, 2015. `https://www.dropbox.com/s/ad1ifkeoucwargz/4%20-%207%20-%204.7-%20Parallel%20Computation%20Patterns%20-%20More%20on%20Parallel%20Scan.mp4?dl=0`.

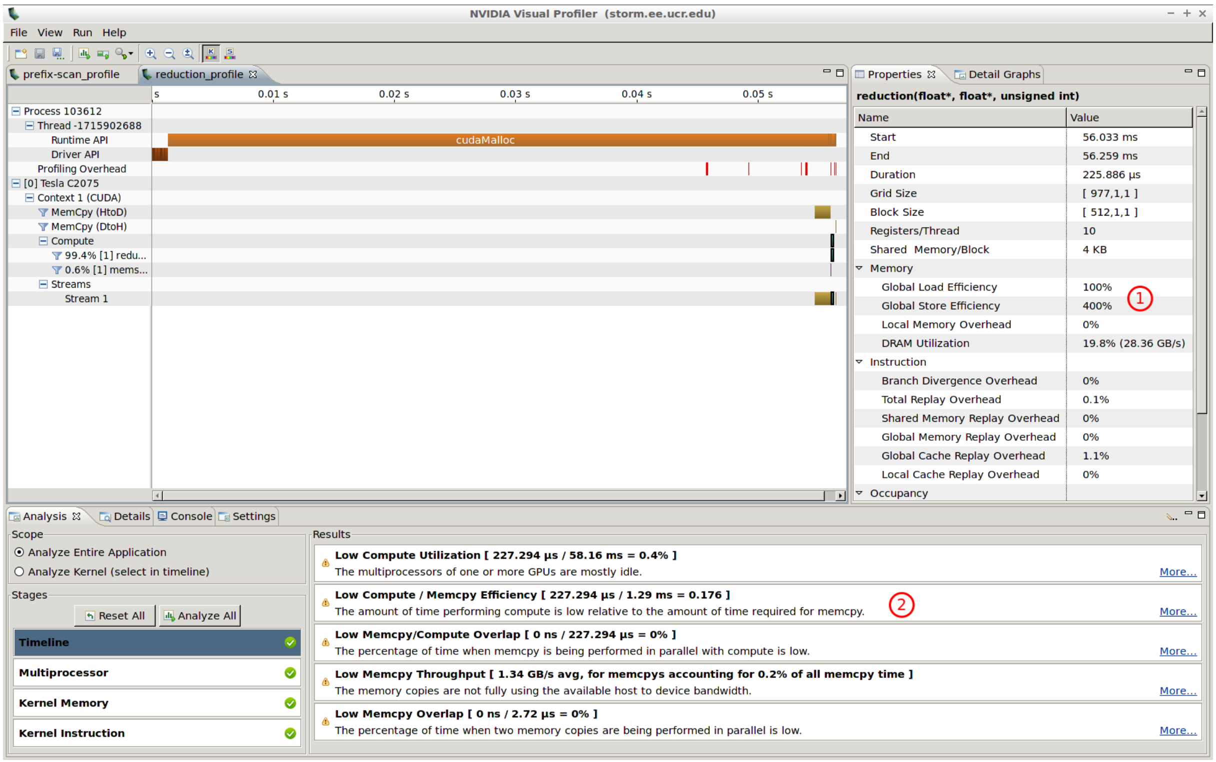

Figure 1: NVVP performance analysis for *reduction*.

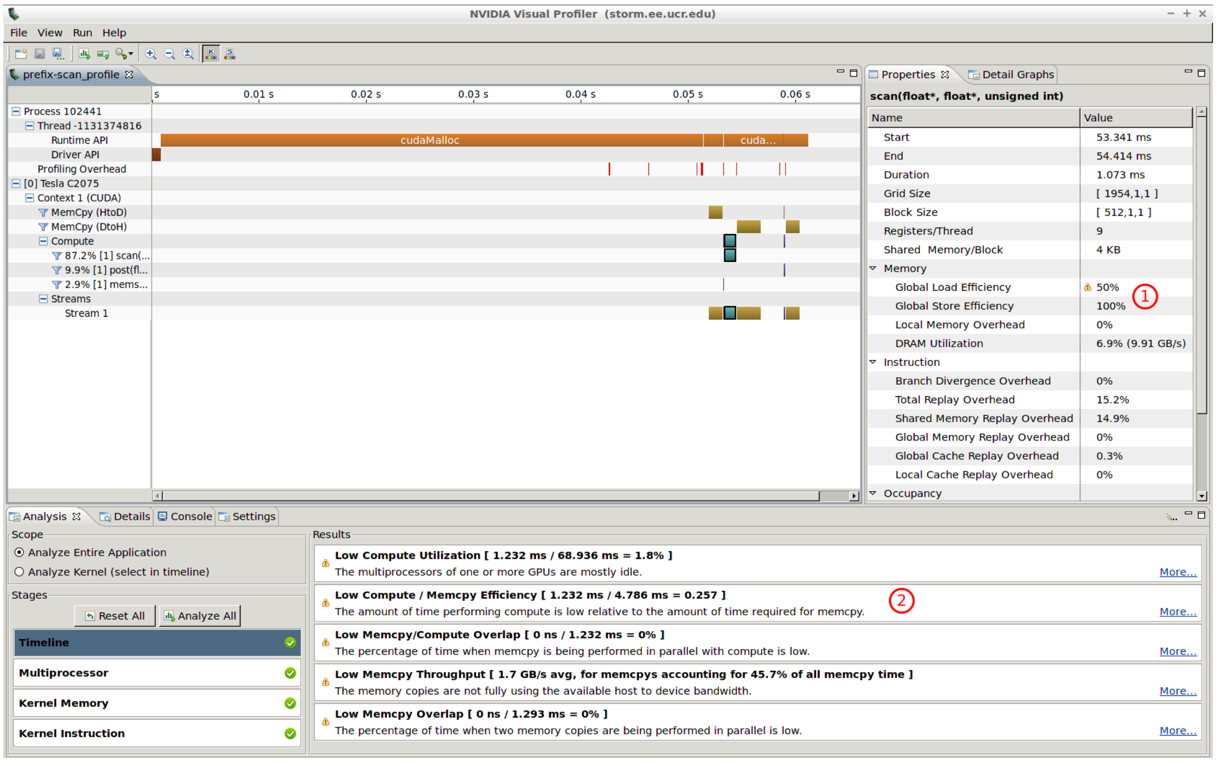

Figure 2: NVVP performance analysis for *prefix-scan*.

[3] David Luebke, John Owens, Mike Roberts and Cheng-Han Lee. *Blelloch Scan - Intro to Parallel Programming*. Udacity Course, 2015. `https://www.youtube.com/watch?v=hyKA5fb5ZJI`.

[4] Mark Harris, Shubhabrata Sengupta and John D. Owens. *Parallel Prefix Sum (Scan) with CUDA* in *GPU Gems 3* edited by Hubert Nguyen. Addison-Wesley, 2007. `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html`.

[5] Mark Harris. *How to Overlap Data Transfers in CUDA C/C++* in *Parallel Forall* Website. Nvidia, 2012. `http://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/`.

[6] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*. pag 52-60. Morgan Kaufmann. 2013.