Report Lab 3

Andres Calderon - SID:861243796

November 12, 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.

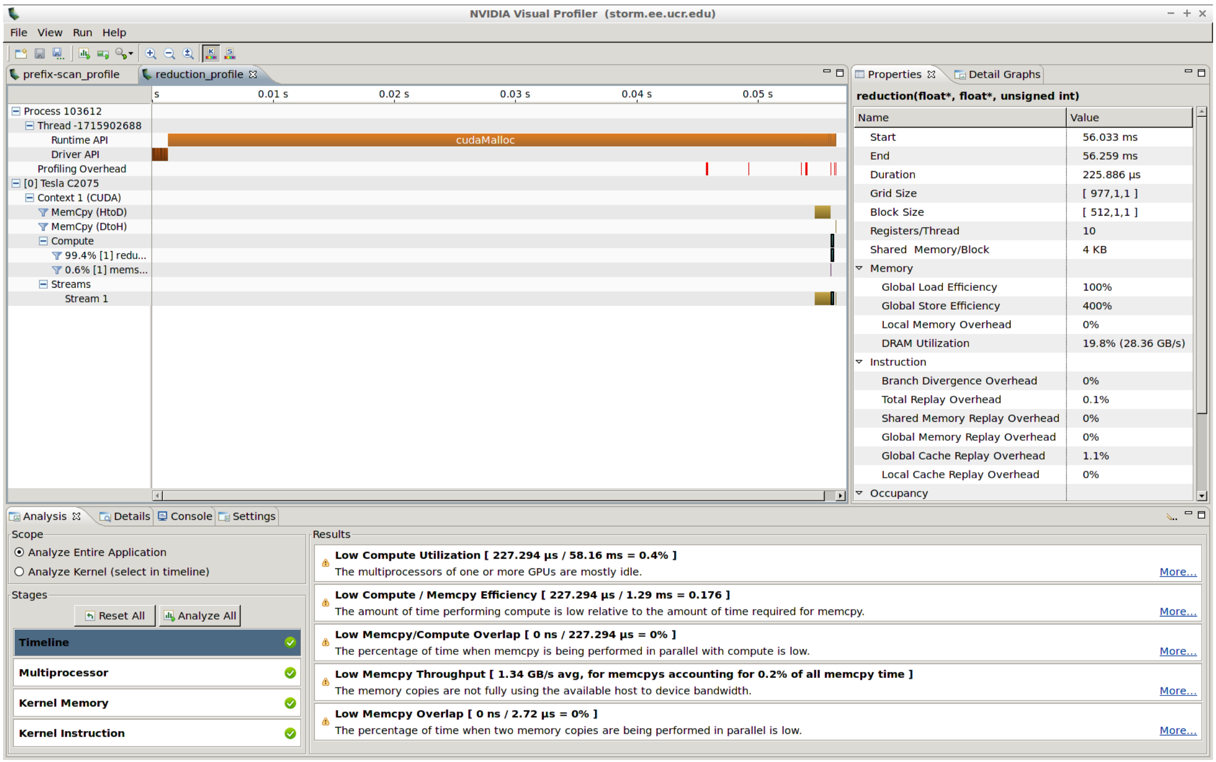
1.2 Prefix-scan

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){
  __shared__ float section[2 * BLOCK_SIZE];
  int t = blockDim.x * blockIdx.x + threadIdx.x;

  if(t < size)
    if(t == 0)
      section[0] = 0.0f;
    else
      section[threadIdx.x] = in[t - 1];
  __syncthreads();

  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();
  }

  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();
  }
  //__syncthreads();
  if(t < size)
    out[t] = section[threadIdx.x];
}

__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
*******************************************************************************/
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));

  // My code...
  float *partial_h, *partial_d;
  partial_h = (float *) malloc((num_elements/BLOCK_SIZE + 1) * sizeof(float));
  partial_h[0] = 0;
  int n = 1;
  for(int i = BLOCK_SIZE - 1; i < num_elements; i += BLOCK_SIZE){
    partial_h[n] = partial_h[n - 1] + out_h[i];
    n++;
  }
  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");
  }
  //
  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");
  //
  postScan(out_d, partial_d, num_elements);
  cuda_ret = cudaDeviceSynchronize();
  if(cuda_ret != cudaSuccess) FATAL("Unable to launch/execute kernel");

  //
  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);
```



reduction(float*, float*, unsigned int)

Name	Value
Start	56.033 ms
End	56.259 ms
Duration	225.886 µs
Grid Size	[977,1,1]
Block Size	[512,1,1]
Registers/Thread	10
Shared Memory/Block	4 KB
Memory	
Global Load Efficiency	100%
Global Store Efficiency	400%
Local Memory Overhead	0%
DRAM Utilization	19.8% (28.36 GB/s)
Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	0.1%
Shared Memory Replay Overhead	0%
Global Memory Replay Overhead	0%
Global Cache Replay Overhead	1.1%
Local Cache Replay Overhead	0%
Occupancy	

Scope

● Analyze Entire Application

○ Analyze Kernel (select in timeline)

Stages

Timeline ✓

Multiprocessor ✓

Kernel Memory ✓

Kernel Instruction ✓

Results

Low Compute Utilization [227.294 µs / 58.16 ms = 0.4%]
The multiprocessors of one or more GPUs are mostly idle. More...

Low Compute / Memcpy Efficiency [227.294 µs / 1.29 ms = 0.176]
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

Low Memcpy/Compute Overlap [0 ns / 227.294 µs = 0%]
The percentage of time when memcpy is being performed in parallel with compute is low. More...

Low Memcpy Throughput [1.34 GB/s avg, for memcpys accounting for 0.2% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Low Memcpy Overlap [0 ns / 2.72 µs = 0%]
The percentage of time when two memory copies are being performed in parallel is low. More...

Figure 1: NVVP performance analysis for *reduction*.

```
    free(in_h); free(out_h); free(partial_h);

    return 0;
}
```

2 Answers to Questions

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

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

   For reduction: padding...

   For prefix-scan: Harris' algorithm...

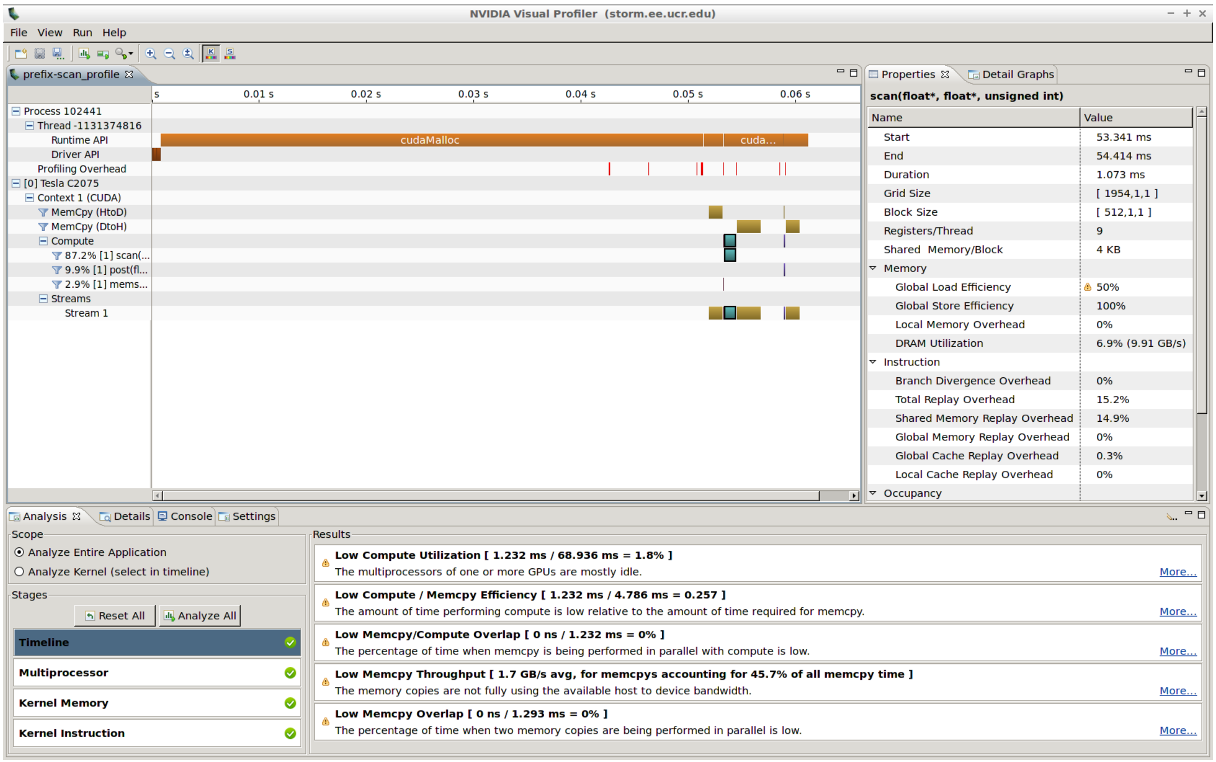
References

[1] Andres Calderon. *GitHub Personal Repository*, 2015. `https://github.com/aocalderon/PhD/tree/master/Y1Q1/GPU/lab2`.

[2] David Kirk and Wen-Mei Hwu. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann, 2012.

[3] Wen-Mei Hwu. *A Tiled Kernel for Arbitrary Matrix Dimensions - Heterogeneous Parallel Programming*. Coursera Course, 2015. `https://www.dropbox.com/s/4y06b1m6dozp2kt/2%20-%208%20-%202.8-%20A%20Tiled%20Kernel%20for%20Arbitrary%20Matrix%20Dimensions.mp4?dl=0`.

[4] David Luebke, John Owens, Mike Roberts and Cheng-Han Lee. *Using NVVP Part1 and Part 2 - Intro to Parallel Programming*. Udacity Course, 2015. `https://www.youtube.com/watch?v=hyKA5fb5ZJI`.



scan(float*, float*, unsigned int)

Name	Value
Start	53.341 ms
End	54.414 ms
Duration	1.073 ms
Grid Size	[1954,1,1]
Block Size	[512,1,1]
Registers/Thread	9
Shared Memory/Block	4 KB
▿ Memory	
Global Load Efficiency	⚠ 50%
Global Store Efficiency	100%
Local Memory Overhead	0%
DRAM Utilization	6.9% (9.91 GB/s)
▿ Instruction	
Branch Divergence Overhead	0%
Total Replay Overhead	15.2%
Shared Memory Replay Overhead	14.9%
Global Memory Replay Overhead	0%
Global Cache Replay Overhead	0.3%
Local Cache Replay Overhead	0%
▿ Occupancy	

Scope

● Analyze Entire Application

○ Analyze Kernel (select in timeline)

Stages

Reset All | Analyze All

- **Timeline** ✓
- **Multiprocessor** ✓
- **Kernel Memory** ✓
- **Kernel Instruction** ✓

Results

⚠ **Low Compute Utilization [1.232 ms / 68.936 ms = 1.8%]**
The multiprocessors of one or more GPUs are mostly idle. More...

⚠ **Low Compute / Memcpy Efficiency [1.232 ms / 4.786 ms = 0.257]**
The amount of time performing compute is low relative to the amount of time required for memcpy. More...

⚠ **Low Memcpy/Compute Overlap [0 ns / 1.232 ms = 0%]**
The percentage of time when memcpy is being performed in parallel with compute is low. More...

⚠ **Low Memcpy Throughput [1.7 GB/s avg, for memcpys accounting for 45.7% of all memcpy time]**
The memory copies are not fully using the available host to device bandwidth. More...

⚠ **Low Memcpy Overlap [0 ns / 1.293 ms = 0%]**
The percentage of time when two memory copies are being performed in parallel is low. More...

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

[5] Nvidia Corporation. *GeForce® GTX 200 GPU Architectural Overview*. Technical Brief #TB-04044-001_v01, 2008. http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf.