Module 9 Lab
Reduction

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

Implement a kernel the performs reduction of a 1D list. The reduction should give the sum of the list. You should implement the improved kernel discussed in week 4. Your kernel should be able to handle input lists of arbitrary length. However, for simplicity, you can assume that the input list will be at most `2048 x 65535` elements so that it can be handled by only one kernel launch. The boundary condition can be handled by filling "identity value (0 for sum)" into the shared memory of the last block when the length is not a multiple of the thread block size. Further assume that the reduction sums of each section generated by individual blocks will be summed up by the CPU. Prerequisites

PREREQUISITES

Before starting this lab, make sure that:

- You have completed week 4 lecture videos

INSTRUCTION

Edit the code in the code tab to perform the following:

- allocate device memory
- copy host memory to device
- initialize thread block and kernel grid dimensions
- invoke CUDA kernel
- copy results from device to host
- deallocate device memory
- implement the improved reduction routine
- use shared memory to reduce the number of global accesses, handle the boundary conditions in when loading input list elements into the shared memory
- implement a CPU loop to perform final reduction based on the sums of sections generated by the thread blocks

Instructions about where to place each part of the code is demarcated by the `//@@` comment lines.

QUESTIONS

(1) Name 3 applications of reduction.

ANSWER: **Determining min/max/average of a data series.**

(2) How many floating operations are being performed in your reduction kernel? explain.

ANSWER: **Input length - 1.**

(3) How many global memory reads are being performed by your kernel? explain.

ANSWER: **numInputs. Each input value is copied to shared memory.**

(4) How many global memory writes are being performed by your kernel? explain.

ANSWER: **numInputs / (BLOCK_SIZE * 2). Only final outputs are written to global memory.**

(5) What is the minimum, maximum, and average number of real operations that a thread will perform? Real operations are those that directly contribute to the final reduction value.

ANSWER: **Minimum is 1. Max is log2(BLOCK_SIZE)+1. Average is (sum from i = 0 to log2(BLOCK_SIZE) of 2î)/BLOCK_SIZE.**

(6) How many times does a single thread block synchronize to reduce its portion of the array to a single value?

ANSWER: **log2(BLOCK_SIZE)+1**

(7) Describe what optimizations were performed to your kernel to achieve a performance speedup.

ANSWER: **Active threads are adjacent so as high a fraction of each warp is active as possible.**

(8) Describe what further optimizations can be implemented to your kernel to achieve a performance speedup.

ANSWER: **Use atomic operations once the reduction tree is small enough to accumulate a single final value.**

(9) Suppose the input is greater than 2048*6535, what modifications are needed to your kernel?

ANSWER: **The work could be further coarsened, so a thread does more than two inputs in the beginning.**

(10) Suppose a you want to scan using a a binary operator that's not commutative, can you use a parallel reduction for that?

ANSWER: **To parallelize a reduction the operator must be associative, not necessarily commutative.**

(11) Is it possible to get different results from running the serial version and parallel version of reduction? explain.

ANSWER: **The order of floating-point operations in the serial and parallel version is different, so it is possible.**

CODE TEMPLATE

The following code is suggested as a starting point for students. The code handles the import and export as well as the checking of the solution. Students are expected to insert their code is the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```

// Given a list (lst) of length n
// Output its sum = lst[0] + lst[1] + ... + lst[n-1];

#include <wb.h>

#define BLOCK_SIZE 512 //@@ You can change this

#define wbCheck(stmt)                                                  \
  do {                                                                 \
    cudaError_t err = stmt;                                            \
    if (err != cudaSuccess) {                                          \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                      \
      return -1;                                                       \
    }                                                                  \
  } while (0)

__global__ void total(float *input, float *output, int len) {
  //@@ Load a segment of the input vector into shared memory

  //@@ Traverse the reduction tree

  //@@ Write the computed sum of the block to the output vector at the
  //@@ correct index
}

int main(int argc, char **argv) {
  int ii;
  wbArg_t args;
  float *hostInput;  // The input 1D list
  float *hostOutput; // The output list
  float *deviceInput;
  float *deviceOutput;
  int numInputElements;  // number of elements in the input list
  int numOutputElements; // number of elements in the output list
```

```
  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput =
      (float *)wbImport(wbArg_getInputFile(args, 0), &numInputElements);

  numOutputElements = numInputElements / (BLOCK_SIZE << 1);
  if (numInputElements % (BLOCK_SIZE << 1)) {
    numOutputElements++;
  }
  hostOutput = (float *)malloc(numOutputElements * sizeof(float));

  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The number of input elements in the input is ",
        numInputElements);
  wbLog(TRACE, "The number of output elements in the input is ",
        numOutputElements);

  wbTime_start(GPU, "Allocating GPU memory.");
  //@@ Allocate GPU memory here
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  //@@ Copy memory to the GPU here
  wbTime_stop(GPU, "Copying input memory to the GPU.");
  //@@ Initialize the grid and block dimensions here
  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Launch the GPU Kernel here
  cudaDeviceSynchronize();
  wbTime_stop(Compute, "Performing CUDA computation");

  wbTime_start(Copy, "Copying output memory to the CPU");
  //@@ Copy the GPU memory back to the CPU here
  wbTime_stop(Copy, "Copying output memory to the CPU");

  /********************************************************************
   * Reduce output vector on the host
   * NOTE: One could also perform the reduction of the output vector
   * recursively and support any size input. For simplicity, we do not
   * require that for this lab.
   ********************************************************************/

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here
  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, 1);

  free(hostInput);
  free(hostOutput);

```

```
  return 0;
}
```

CODE SOLUTION

The following is a possible implementation of the lab. This solution is intended for use only by the teaching staff and should not be distributed to students.

```
// Given a list (lst) of length n
// Output its sum = lst[0] + lst[1] + ... + lst[n-1];

#include <wb.h>

#define BLOCK_SIZE 512 //@@ You can change this

#define wbCheck(stmt)                                                  \
  do {                                                                 \
    cudaError_t err = stmt;                                            \
    if (err != cudaSuccess) {                                          \
      wbLog(ERROR, "Failed to run stmt ", #stmt);                      \
      return -1;                                                       \
    }                                                                  \
  } while (0)

__global__ void total(float *input, float *output, int len) {
  //@@ Load a segment of the input vector into shared memory
  __shared__ float partialSum[2 * BLOCK_SIZE];
  unsigned int t = threadIdx.x, start = 2 * blockIdx.x * BLOCK_SIZE;
  if (start + t < len)
    partialSum[t] = input[start + t];
  else
    partialSum[t] = 0;
  if (start + BLOCK_SIZE + t < len)
    partialSum[BLOCK_SIZE + t] = input[start + BLOCK_SIZE + t];
  else
    partialSum[BLOCK_SIZE + t] = 0;
  //@@ Traverse the reduction tree
  for (unsigned int stride = BLOCK_SIZE; stride >= 1; stride >>= 1) {
    __syncthreads();
    if (t < stride)
      partialSum[t] += partialSum[t + stride];
  }
  //@@ Write the computed sum of the block to the output vector at the
  //@@ correct index
  if (t == 0)
    output[blockIdx.x] = partialSum[0];
}

int main(int argc, char **argv) {
  int ii;
  wbArg_t args;
```

```
float *hostInput;  // The input 1D list
float *hostOutput; // The output list
float *deviceInput;
float *deviceOutput;
int numInputElements;  // number of elements in the input list
int numOutputElements; // number of elements in the output list

args = wbArg_read(argc, argv);

wbTime_start(Generic, "Importing data and creating memory on host");
hostInput =
    (float *)wbImport(wbArg_getInputFile(args, 0), &numInputElements);

numOutputElements = numInputElements / (BLOCK_SIZE << 1);
if (numInputElements % (BLOCK_SIZE << 1)) {
  numOutputElements++;
}
hostOutput = (float *)malloc(numOutputElements * sizeof(float));

wbTime_stop(Generic, "Importing data and creating memory on host");

wbLog(TRACE, "The number of input elements in the input is ",
      numInputElements);
wbLog(TRACE, "The number of output elements in the input is ",
      numOutputElements);

wbTime_start(GPU, "Allocating GPU memory.");
//@@ Allocate GPU memory here
cudaMalloc(&deviceInput, sizeof(float) * numInputElements);
cudaMalloc(&deviceOutput, sizeof(float) * numOutputElements);

wbTime_stop(GPU, "Allocating GPU memory.");

wbTime_start(GPU, "Copying input memory to the GPU.");
//@@ Copy memory to the GPU here
cudaMemcpy(deviceInput, hostInput, sizeof(float) * numInputElements,
           cudaMemcpyHostToDevice);

wbTime_stop(GPU, "Copying input memory to the GPU.");
//@@ Initialize the grid and block dimensions here
dim3 dimGrid(numOutputElements, 1, 1);
dim3 dimBlock(BLOCK_SIZE, 1, 1);

wbTime_start(Compute, "Performing CUDA computation");
//@@ Launch the GPU Kernel here
total<<<dimGrid, dimBlock>>>(deviceInput, deviceOutput,
                             numInputElements);

cudaDeviceSynchronize();
wbTime_stop(Compute, "Performing CUDA computation");

wbTime_start(Copy, "Copying output memory to the CPU");
//@@ Copy the GPU memory back to the CPU here
```

```
  cudaMemcpy(hostOutput, deviceOutput, sizeof(float) * numOutputElements,
             cudaMemcpyDeviceToHost);

  wbTime_stop(Copy, "Copying output memory to the CPU");

  /********************************************************************
   * Reduce output vector on the host
   * NOTE: One could also perform the reduction of the output vector
   * recursively and support any size input. For simplicity, we do not
   * require that for this lab.
   ********************************************************************/
  for (ii = 1; ii < numOutputElements; ii++) {
    hostOutput[0] += hostOutput[ii];
  }

  wbTime_start(GPU, "Freeing GPU Memory");
  //@@ Free the GPU memory here
  cudaFree(deviceInput);
  cudaFree(deviceOutput);

  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, 1);

  free(hostInput);
  free(hostOutput);

  return 0;
}
```