List Scan Grade

Grade Summary (History) (/grade/history/5985)

Created:	less than a minute ago (2022-04-18 04:22:38 +0000 UTC)
Total Score:	100 out of 100 points
Coding Score:	100 out of 100 points
Questions Score:	0

Program Code

```
/*
Dataset Id:  5
Created:  less than a minute ago
Status: Correct solution for this dataset.
Timer Output
Kind  Location  Time (ms) Message
Generic main.cu::127  54.583396 Importing data and creating memory on ho
GPU main.cu::134  1.679192  Allocating GPU memory.
GPU main.cu::141  0.043779  Clearing output memory.
GPU main.cu::145  0.064188  Copying input memory to the GPU.
Compute main.cu::153  0.092417  Performing CUDA computation
Copy  main.cu::167  0.05866 Copying output memory to the CPU
GPU main.cu::171  0.2507  Freeing GPU Memory
Logger Output
Level Location  Message
Trace main::132 The number of input elements in the input is 9010

*/


// MP Scan
// Given a list (lst) of length n
// Output its prefix sum = {lst[0], lst[0] + lst[1], 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);
      wbLog(ERROR, "Got CUDA error ...  ", cudaGetErrorString(err));
      return -1;
    }
  } while (0)

__global__ void scan(float *input, float *output, int len, int flag) {
  //@@ Modify the body of this function to complete the functionality of
  //@@ the scan on the device
  //@@ You may need multiple kernel calls; write your kernels before this
  //@@ function and call them from here
  __shared__ float block_arrray_scan[2 * BLOCK_SIZE];

  int loop_index = 0;
  int stride = 1;
  if (!flag){
    loop_index = (2 * blockIdx.x * blockDim.x) + threadIdx.x;
    stride = blockDim.x;
  }
  else{
    loop_index = (threadIdx.x + 1) * (2 * blockDim.x) - 1;
    stride = 2 * blockDim.x;
  }

  int storeIndex = (2 * blockIdx.x * blockDim.x) + threadIdx.x;

  //data input
  if (loop_index < len){
    block_arrray_scan[threadIdx.x] = input[loop_index];
  }
  else{
    block_arrray_scan[threadIdx.x] = 0;
  }
  if (loop_index + stride < len){
    block_arrray_scan[threadIdx.x + blockDim.x] = input[loop_index + stri
  }
  else{
    block_arrray_scan[threadIdx.x + blockDim.x] = 0;
  }

  //First Step: Reduction
  for (int stride = 1; stride <= (2 * BLOCK_SIZE); stride *= 2) {
    __syncthreads();

    int loop_index = (threadIdx.x + 1) * 2 * stride - 1;

    if ((loop_index < 2 * BLOCK_SIZE) && ((loop_index - stride) >= 0)){
        block_arrray_scan[loop_index] += block_arrray_scan[loop_index - s
    }
  }

  //Use Distribution Tree method after Scanning
  for (int stride = 2 * BLOCK_SIZE / 4; stride > 0; stride /= 2) {
    __syncthreads();

    int loop_index = (threadIdx.x + 1) * 2 * stride - 1;
    if ((loop_index + stride) < 2 * BLOCK_SIZE){
        block_arrray_scan[loop_index + stride] += block_arrray_scan[loop_
    }
  }

  __syncthreads();
  if (storeIndex < len){
    output[storeIndex] = block_arrray_scan[threadIdx.x];
  }

  if (storeIndex + blockDim.x < len){
    output[storeIndex + blockDim.x] = block_arrray_scan[threadIdx.x + blo
  }
}

__global__ void add(float *input, float *output, float *array_sum, int le
  __shared__ float move_loop;

  int loop_index = threadIdx.x + (2 * blockIdx.x * blockDim.x);

  if (threadIdx.x == 0){
    if (blockIdx.x == 0){
        move_loop = 0;
    }
    else{
        move_loop = array_sum[blockIdx.x - 1];
    }
  }

  __syncthreads();

  if (loop_index < len){
    output[loop_index] = input[loop_index] + move_loop;
  }
  if (loop_index + blockDim.x < len){
    output[loop_index + blockDim.x] = input[loop_index + blockDim.x] + mo
  }
}


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

  //Additional Variables
  //store temporary results from scanning
  //store block summations from scanning
  float *device_temporary_value;
  float *scanned_dev_temp_val;

  args = wbArg_read(argc, argv);

  wbTime_start(Generic, "Importing data and creating memory on host");
  hostInput = (float *)wbImport(wbArg_getInputFile(args, 0), &numElements
  hostOutput = (float *)malloc(numElements * sizeof(float));
  wbTime_stop(Generic, "Importing data and creating memory on host");

  wbLog(TRACE, "The number of input elements in the input is ", numElemen

  wbTime_start(GPU, "Allocating GPU memory.");
  wbCheck(cudaMalloc((void **)&deviceInput, numElements * sizeof(float)))
  wbCheck(cudaMalloc((void **)&deviceOutput, numElements * sizeof(float))
  wbCheck(cudaMalloc((void **)&device_temporary_value, numElements * size
  wbCheck(cudaMalloc((void **)&scanned_dev_temp_val, 2 * BLOCK_SIZE * siz
  wbTime_stop(GPU, "Allocating GPU memory.");

  wbTime_start(GPU, "Clearing output memory.");
  wbCheck(cudaMemset(deviceOutput, 0, numElements * sizeof(float)));
  wbTime_stop(GPU, "Clearing output memory.");

  wbTime_start(GPU, "Copying input memory to the GPU.");
  wbCheck(cudaMemcpy(deviceInput, hostInput, numElements * sizeof(float),
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here
  dim3 dimGrid(ceil(numElements/(BLOCK_SIZE * 2.0)),   1, 1);
  dim3 dimBlock(BLOCK_SIZE, 1, 1);

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Modify this to complete the functionality of the scan
  //@@ on the deivce

  //Here I store the temporary value in deviceOutput
  scan<<<dimGrid, dimBlock>>>(deviceInput, device_temporary_value, numEle

  dim3 postScanGrid(1, 1, 1);
  scan<<<postScanGrid, dimBlock>>>(device_temporary_value, scanned_dev_te
  add<<<dimGrid, dimBlock>>>(device_temporary_value, deviceOutput, scanne

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

  wbTime_start(Copy, "Copying output memory to the CPU");
  wbCheck(cudaMemcpy(hostOutput, deviceOutput, numElements * sizeof(float
  wbTime_stop(Copy, "Copying output memory to the CPU");

  wbTime_start(GPU, "Freeing GPU Memory");
  cudaFree(deviceInput);
  cudaFree(deviceOutput);
  cudaFree(device_temporary_value);
  cudaFree(scanned_dev_temp_val);
  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, numElements);

  free(hostInput);
  free(hostOutput);

  return 0;
}
```

Designed and architected by Abdul Dakkak (https://www.dakkak.dev/).