Module 10 Lab
List Scan

GPU Teaching Kit – Accelerated Computing

OBJECTIVE

Implement a kernel to perform an inclusive parallel scan on a 1D list. The scan operator will be the addition (plus) operator. You should implement the work efficient kernel in Lecture 4.6. Your kernel should be able to handle input lists of arbitrary length. To simplify the lab, the student can assume that the input list will be at most of length $2048 \times 65,535$ elements. This means that the computation can be performed using 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.

PREREQUISITES

Before starting this lab, make sure that:

- You have completed the required course modules

INSTRUCTIONS

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 work efficient scan routine
- use shared memory to reduce the number of global memory accesses, handle the boundary conditions when loading input list elements into the shared memory

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

LOCAL SETUP INSTRUCTIONS

The most recent version of source code for this lab along with the build-scripts can be found on the Bitbucket repository. A description on how to use the CMake tool in along with how to build the labs for local development found in the README document in the root of the repository.

The executable generated as a result of compiling the lab can be run using the following command:

```
./ListScan_Template -e <expected.raw> \
  -i <input.raw> -o <output.raw> -t vector
```

where `<expected.raw>` is the expected output, `<input.raw>` is the input dataset, and `<output.raw>` is an optional path to store the results. The datasets can be generated using the dataset generator built as part of the compilation process.

QUESTIONS

(1) Name 3 applications of parallel scan.

ANSWER: **Sorting, resource allocation, and numerical integration.**

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

ANSWER: **Let N be the input length. Reduction phase has $N - 1$ adds. $N - 1 - \log 2(N)$ adds.**

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

ANSWER: **One per input element.**

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

ANSWER: **Scratch work is done in shared memory, so one per input element.**

(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 op, max is $\log 2(N)$. Average in reduction phase is $(N-1)/(BLOCKSIZE)$, average in post-reduction has $(N-1-\log 2(N))/(BLOCKSIZE)$**

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

ANSWER: **$\log 2(N)$ syncs in reduction, $\log 2(N) - 1$ syncs in post-reduction.**

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

ANSWER: **A work-efficient algorithm was implemented. To reduce divergence, threads work on data that is far away instead of local.**

(8) Describe what further optimizations can be implemented to your kernel and what would be the expected performance behavior?

ANSWER: **Unroll loops when the bounds are known to reduce thread divergence.**

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

ANSWER: **An intial phase where each thread handles more than two inputs could be done.**

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

ANSWER: **Yes. The operation just must be associative.**

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

ANSWER: **Since floating point operations are not associative, a different answer 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 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) {
  //@@ 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
}

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

  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 ",
        numElements);

  wbTime_start(GPU, "Allocating GPU memory.");
  wbCheck(cudaMalloc((void **)&deviceInput, numElements * sizeof(float)));
  wbCheck(cudaMalloc((void **)&deviceOutput, numElements * sizeof(float)));
  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),
                     cudaMemcpyHostToDevice));
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here

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

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

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

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

  wbSolution(args, hostOutput, numElements);

  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 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);                         \
      return -1;                                                          \
    }                                                                     \
  } while (0)

__global__ void fixup(float *input, float *aux, int len) {
  unsigned int t = threadIdx.x, start = 2 * blockIdx.x * BLOCK_SIZE;
  if (blockIdx.x) {
    if (start + t < len)
      input[start + t] += aux[blockIdx.x - 1];
    if (start + BLOCK_SIZE + t < len)
      input[start + BLOCK_SIZE + t] += aux[blockIdx.x - 1];
  }
}

__global__ void scan(float *input, float *output, float *aux, int len) {
  // Load a segment of the input vector into shared memory
  __shared__ float scan_array[BLOCK_SIZE << 1];
  unsigned int t = threadIdx.x, start = 2 * blockIdx.x * BLOCK_SIZE;
  if (start + t < len)
    scan_array[t] = input[start + t];
  else
    scan_array[t] = 0;
  if (start + BLOCK_SIZE + t < len)
    scan_array[BLOCK_SIZE + t] = input[start + BLOCK_SIZE + t];
  else
    scan_array[BLOCK_SIZE + t] = 0;
  __syncthreads();

  // Reduction
  int stride;
  for (stride = 1; stride <= BLOCK_SIZE; stride <<= 1) {
    int index = (t + 1) * stride * 2 - 1;
    if (index < 2 * BLOCK_SIZE)
      scan_array[index] += scan_array[index - stride];
    __syncthreads();
  }

  // Post reduction
  for (stride = BLOCK_SIZE >> 1; stride; stride >>= 1) {
    int index = (t + 1) * stride * 2 - 1;
    if (index + stride < 2 * BLOCK_SIZE)
      scan_array[index + stride] += scan_array[index];
    __syncthreads();
  }

  if (start + t < len)
    output[start + t] = scan_array[t];
  if (start + BLOCK_SIZE + t < len)
    output[start + BLOCK_SIZE + t] = scan_array[BLOCK_SIZE + t];

  if (aux && t == 0)
    aux[blockIdx.x] = scan_array[2 * BLOCK_SIZE - 1];
}

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

  args = wbArg_read(argc, argv);

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

  wbLog(TRACE, "The number of input elements in the input is ",
        numElements);

  wbTime_start(GPU, "Allocating GPU memory.");
  wbCheck(cudaMalloc((void **)&deviceInput, numElements * sizeof(float)));
  wbCheck(cudaMalloc((void **)&deviceOutput, numElements * sizeof(float)));

  // XXX the size is fixed for ease of implementation.
  cudaMalloc(&deviceAuxArray, (BLOCK_SIZE << 1) * sizeof(float));
  cudaMalloc(&deviceAuxScannedArray, (BLOCK_SIZE << 1) * sizeof(float));
  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),
                     cudaMemcpyHostToDevice));
  wbTime_stop(GPU, "Copying input memory to the GPU.");

  //@@ Initialize the grid and block dimensions here
  int numBlocks = ceil((float)numElements / (BLOCK_SIZE << 1));
  dim3 dimGrid(numBlocks, 1, 1);
  dim3 dimBlock(BLOCK_SIZE, 1, 1);
  wbLog(TRACE, "The number of blocks is ", numBlocks);

  wbTime_start(Compute, "Performing CUDA computation");
  //@@ Modify this to complete the functionality of the scan
  //@@ on the deivce
  scan<<<dimGrid, dimBlock>>>(deviceInput, deviceOutput, deviceAuxArray,
                              numElements);
  cudaDeviceSynchronize();
  scan<<<dim3(1, 1, 1), dimBlock>>>(deviceAuxArray, deviceAuxScannedArray,
                                    NULL, BLOCK_SIZE << 1);
  cudaDeviceSynchronize();
  fixup<<<dimGrid, dimBlock>>>(deviceOutput, deviceAuxScannedArray,
                               numElements);

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

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

  wbTime_start(GPU, "Freeing GPU Memory");
  cudaFree(deviceInput);
  cudaFree(deviceOutput);
  cudaFree(deviceAuxArray);
  cudaFree(deviceAuxScannedArray);
  wbTime_stop(GPU, "Freeing GPU Memory");

  wbSolution(args, hostOutput, numElements);

  free(hostInput);
  cudaFreeHost(hostOutput);

  return 0;
}
```