

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 $(N - 1)/(\text{BLOCKSIZE})$, average in post-reduction has $(N - 1 - \log_2(N))/(\text{BLOCKSIZE})$**

- (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 initial phase where each thread handles more than two inputs could be done.**

- (10) Suppose a you want to scan using 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 in the sections demarcated with `//@@`. Students expected the other code unchanged. The tutorial page describes the functionality of the `wb*` methods.

```

1 // Given a list (lst) of length n
2 // Output its prefix sum = {lst[0], lst[0] + lst[1], lst[0] + lst[1] + ...
3 // +
4 // lst[n-1]}
5
6 #include <wb.h>
7
8 #define BLOCK_SIZE 512 //@@ You can change this
9
10 #define wbCheck(stmt) \
11     do { \
12         cudaError_t err = stmt; \
13         if (err != cudaSuccess) { \
14             wbLog(ERROR, "Failed to run stmt ", #stmt); \
15             wbLog(ERROR, "Got CUDA error ... ", cudaGetErrorString(err)); \
16             return -1; \
17         } \
18     } while (0)
19
20 __global__ void scan(float *input, float *output, int len) {
21     //@@ Modify the body of this function to complete the functionality of

```

```

22  ///@ the scan on the device
23  ///@ You may need multiple kernel calls; write your kernels before this
24  ///@ function and call them from here
25  }
26
27  int main(int argc, char **argv) {
28      wbArg_t args;
29      float *hostInput; // The input 1D list
30      float *hostOutput; // The output list
31      float *deviceInput;
32      float *deviceOutput;
33      int numElements; // number of elements in the list
34
35      args = wbArg_read(argc, argv);
36
37      wbTime_start(Generic, "Importing data and creating memory on host");
38      hostInput = (float *)wbImport(wbArg_getInputFile(args, 0), &numElements);
39      hostOutput = (float *)malloc(numElements * sizeof(float));
40      wbTime_stop(Generic, "Importing data and creating memory on host");
41
42      wbLog	TRACE, "The number of input elements in the input is ",
43              numElements);
44
45      wbTime_start(GPU, "Allocating GPU memory.");
46      wbCheck(cudaMalloc((void **)&deviceInput, numElements * sizeof(float)));
47      wbCheck(cudaMalloc((void **)&deviceOutput, numElements * sizeof(float)));
48      wbTime_stop(GPU, "Allocating GPU memory.");
49
50      wbTime_start(GPU, "Clearing output memory.");
51      wbCheck(cudaMemset(deviceOutput, 0, numElements * sizeof(float)));
52      wbTime_stop(GPU, "Clearing output memory.");
53
54      wbTime_start(GPU, "Copying input memory to the GPU.");
55      wbCheck(cudaMemcpy(deviceInput, hostInput, numElements * sizeof(float),
56                          cudaMemcpyHostToDevice));
57      wbTime_stop(GPU, "Copying input memory to the GPU.");
58
59      ///@ Initialize the grid and block dimensions here
60
61      wbTime_start(Compute, "Performing CUDA computation");
62      ///@ Modify this to complete the functionality of the scan
63      ///@ on the device
64
65      cudaDeviceSynchronize();
66      wbTime_stop(Compute, "Performing CUDA computation");
67
68      wbTime_start(Copy, "Copying output memory to the CPU");
69      wbCheck(cudaMemcpy(hostOutput, deviceOutput, numElements * sizeof(float),
70                          cudaMemcpyDeviceToHost));
71      wbTime_stop(Copy, "Copying output memory to the CPU");
72
73      wbTime_start(GPU, "Freeing GPU Memory");
74      cudaFree(deviceInput);

```

```

75     cudaFree(deviceOutput);
76     wbTime_stop(GPU, "Freeing GPU Memory");
77
78     wbSolution(args, hostOutput, numElements);
79
80     free(hostInput);
81     free(hostOutput);
82
83     return 0;
84 }

```

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.

```

1  // Given a list (lst) of length n
2  // Output its prefix sum = {lst[0], lst[0] + lst[1], lst[0] + lst[1] + ...
3  // + lst[n-1]}
4
5  #include <wb.h>
6
7  #define BLOCK_SIZE 512 /// You can change this
8
9  #define wbCheck(stmt) \
10     do { \
11         cudaError_t err = stmt; \
12         if (err != cudaSuccess) { \
13             wbLog(ERROR, "Failed to run stmt ", #stmt); \
14             return -1; \
15         } \
16     } while (0)
17
18 __global__ void fixup(float *input, float *aux, int len) {
19     unsigned int t = threadIdx.x, start = 2 * blockIdx.x * BLOCK_SIZE;
20     if (blockIdx.x) {
21         if (start + t < len)
22             input[start + t] += aux[blockIdx.x - 1];
23         if (start + BLOCK_SIZE + t < len)
24             input[start + BLOCK_SIZE + t] += aux[blockIdx.x - 1];
25     }
26 }
27
28 __global__ void scan(float *input, float *output, float *aux, int len) {
29     // Load a segment of the input vector into shared memory
30     __shared__ float scan_array[BLOCK_SIZE << 1];
31     unsigned int t = threadIdx.x, start = 2 * blockIdx.x * BLOCK_SIZE;
32     if (start + t < len)
33         scan_array[t] = input[start + t];
34     else
35         scan_array[t] = 0;

```

```

36     if (start + BLOCK_SIZE + t < len)
37         scan_array[BLOCK_SIZE + t] = input[start + BLOCK_SIZE + t];
38     else
39         scan_array[BLOCK_SIZE + t] = 0;
40     __syncthreads();
41
42     // Reduction
43     int stride;
44     for (stride = 1; stride <= BLOCK_SIZE; stride <= 1) {
45         int index = (t + 1) * stride * 2 - 1;
46         if (index < 2 * BLOCK_SIZE)
47             scan_array[index] += scan_array[index - stride];
48         __syncthreads();
49     }
50
51     // Post reduction
52     for (stride = BLOCK_SIZE >> 1; stride; stride >>= 1) {
53         int index = (t + 1) * stride * 2 - 1;
54         if (index + stride < 2 * BLOCK_SIZE)
55             scan_array[index + stride] += scan_array[index];
56         __syncthreads();
57     }
58
59     if (start + t < len)
60         output[start + t] = scan_array[t];
61     if (start + BLOCK_SIZE + t < len)
62         output[start + BLOCK_SIZE + t] = scan_array[BLOCK_SIZE + t];
63
64     if (aux && t == 0)
65         aux[blockIdx.x] = scan_array[2 * BLOCK_SIZE - 1];
66 }
67
68 int main(int argc, char **argv) {
69     wbArg_t args;
70     float *hostInput; // The input 1D list
71     float *hostOutput; // The output list
72     float *deviceInput;
73     float *deviceOutput;
74     float *deviceAuxArray, *deviceAuxScannedArray;
75     int numElements; // number of elements in the list
76
77     args = wbArg_read(argc, argv);
78
79     wbTime_start(Generic, "Importing data and creating memory on host");
80     hostInput = (float *)wbImport(wbArg_getInputFile(args, 0), &numElements);
81     cudaHostAlloc(&hostOutput, numElements * sizeof(float),
82                  cudaHostAllocDefault);
83     wbTime_stop(Generic, "Importing data and creating memory on host");
84
85     wbLog(TRACE, "The number of input elements in the input is ",
86          numElements);
87
88     wbTime_start(GPU, "Allocating GPU memory.");

```

```

89     wbCheck(cudaMalloc((void **)&deviceInput, numElements * sizeof(float)));
90     wbCheck(cudaMalloc((void **)&deviceOutput, numElements * sizeof(float)));
91
92     // XXX the size is fixed for ease of implementation.
93     cudaMalloc(&deviceAuxArray, (BLOCK_SIZE << 1) * sizeof(float));
94     cudaMalloc(&deviceAuxScannedArray, (BLOCK_SIZE << 1) * sizeof(float));
95     wbTime_stop(GPU, "Allocating GPU memory.");
96
97     wbTime_start(GPU, "Clearing output memory.");
98     wbCheck(cudaMemset(deviceOutput, 0, numElements * sizeof(float)));
99     wbTime_stop(GPU, "Clearing output memory.");
100
101     wbTime_start(GPU, "Copying input memory to the GPU.");
102     wbCheck(cudaMemcpy(deviceInput, hostInput, numElements * sizeof(float),
103                     cudaMemcpyHostToDevice));
104     wbTime_stop(GPU, "Copying input memory to the GPU.");
105
106     /// Initialize the grid and block dimensions here
107     int numBlocks = ceil((float)numElements / (BLOCK_SIZE << 1));
108     dim3 dimGrid(numBlocks, 1, 1);
109     dim3 dimBlock(BLOCK_SIZE, 1, 1);
110     wbLog(TRACE, "The number of blocks is ", numBlocks);
111
112     wbTime_start(Compute, "Performing CUDA computation");
113     /// Modify this to complete the functionality of the scan
114     /// on the device
115     scan<<<dimGrid, dimBlock>>>(deviceInput, deviceOutput, deviceAuxArray,
116                               numElements);
117     cudaDeviceSynchronize();
118     scan<<<dim3(1, 1, 1), dimBlock>>>(deviceAuxArray, deviceAuxScannedArray,
119                                       NULL, BLOCK_SIZE << 1);
120     cudaDeviceSynchronize();
121     fixup<<<dimGrid, dimBlock>>>(deviceOutput, deviceAuxScannedArray,
122                               numElements);
123
124     cudaDeviceSynchronize();
125     wbTime_stop(Compute, "Performing CUDA computation");
126
127     wbTime_start(Copy, "Copying output memory to the CPU");
128     wbCheck(cudaMemcpy(hostOutput, deviceOutput, numElements * sizeof(float),
129                     cudaMemcpyDeviceToHost));
130     wbTime_stop(Copy, "Copying output memory to the CPU");
131
132     wbTime_start(GPU, "Freeing GPU Memory");
133     cudaFree(deviceInput);
134     cudaFree(deviceOutput);
135     cudaFree(deviceAuxArray);
136     cudaFree(deviceAuxScannedArray);
137     wbTime_stop(GPU, "Freeing GPU Memory");
138
139     wbSolution(args, hostOutput, numElements);
140
141     free(hostInput);

```

```
142     cudaFreeHost(hostOutput);  
143  
144     return 0;  
145 }
```

© ⓘ ⓘ This work is licensed by UIUC and NVIDIA (2016) under a [Creative Commons Attribution-NonCommercial 4.0 License](#).