

Parallel Reduction

It is a semi-parallel technique that consists in dividing the vector size into 2 and also divide the amount of threads into two: if you have a size 8 vector, you then use 4 threads to sum the index 0 and 4, 1 and 5, and so on. Then when the half vector is covered, the 4 first elements will have the sum of the first and last 4 elements. Then, you repeat the process for the 4 cells remaining: now 2 have two elements of the first and last 2 elements of the remaining vector, etc.

At the end, you will have one cell with the sum of all the vector elements. This cell will be the first cell of the original array, to then return this value in the kernel. Basically have to divide in two and add up the firsts and lasts, the divide that in two again, until you sum up all the elements.

It is not completely in parallel, but it is done in less iterations than a **sequential programming technique** such as a **for loop**.

- The only condition is that **the vector size is even**.

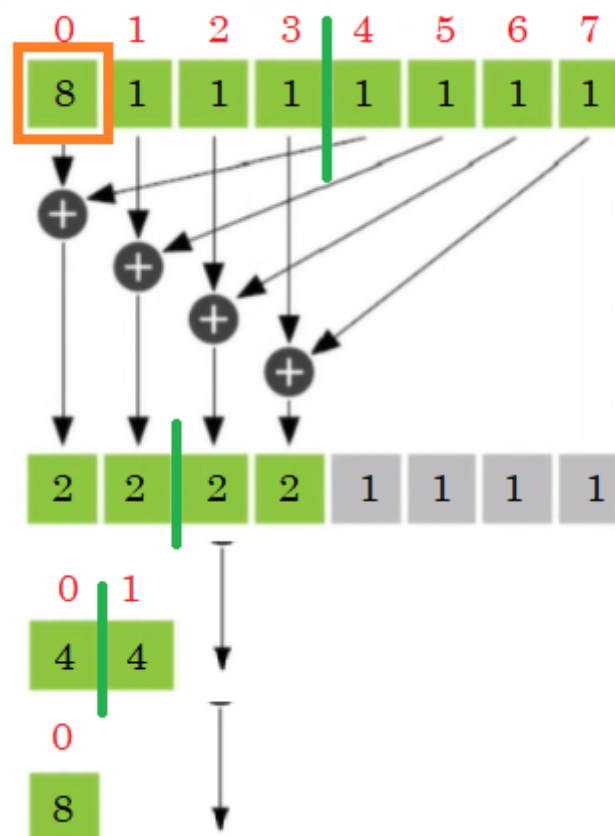


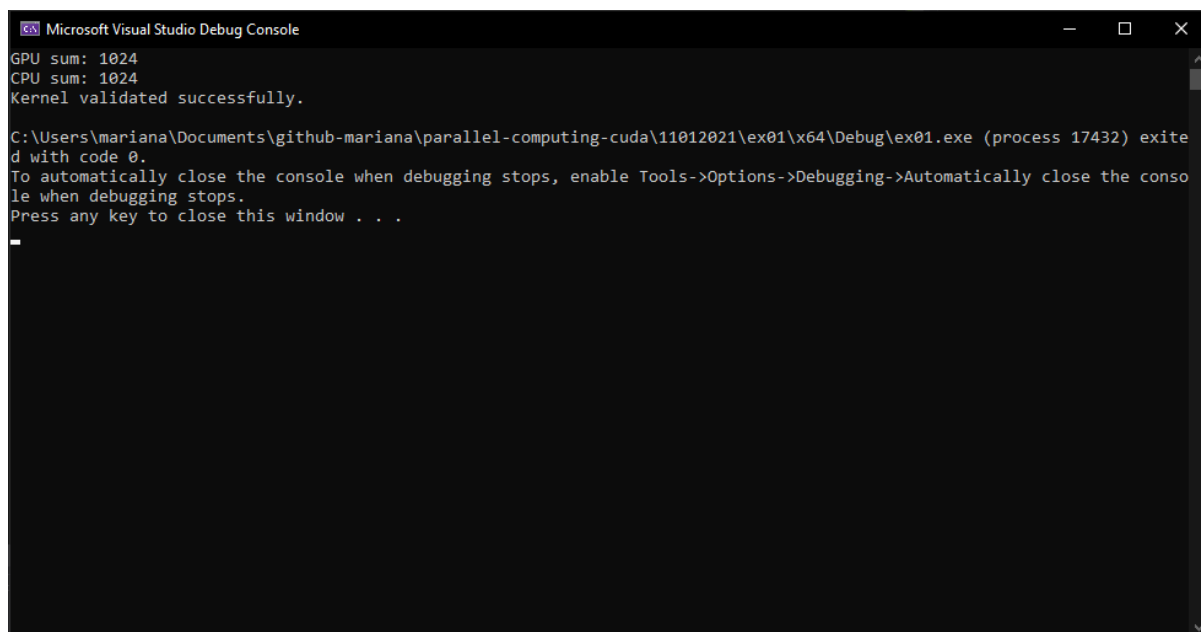
Figure 1: img

Solution

```
1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  __host__ void checkCUDAError(const char* msg) {
8      cudaError_t error;
9      cudaDeviceSynchronize();
10     error = cudaGetLastError();
11     if (error != cudaSuccess) {
12         printf("ERROR %d: %s (%s)\n", error,
13             cudaGetErrorString(error), msg);
14     }
15 }
16
17 __host__ void validate(int* result_CPU, int* result_GPU, int
18     size) {
19     if (*result_CPU != *result_GPU) {
20         printf("The results are not equal.\n");
21         return;
22     }
23     printf("Kernel validated successfully.\n");
24     return;
25 }
26
27 __host__ void CPU_fn(int *v, int* sum, const int size) {
28     for (int i = 0; i < size; i++) {
29         *sum += v[i];
30     }
31 }
32
33 __global__ void kernel(int* v, int* sum) {
34     int gId = threadIdx.x;
35     int step = blockDim.x;
36
37     while (step) {
38         if (gId < step) {
39             v[gId] = v[gId] + v[gId + step];
40         }
41         step = step / 2;
42     }
43     if (gId == 0) {
44         *sum = v[gId];
45     }
46 }
47
48 int main() {
49     const int size = 1024;
```

```
49     int* v = (int*)malloc(sizeof(int) * size);
50     int sumCPU = 0;
51     int sumGPU = 0;
52
53     int* dev_v, *sum;
54     cudaMalloc((void**)&dev_v, sizeof(int) * size);
55     cudaMalloc((void**)&sum, sizeof(int));
56
57     for (int i = 0; i < size; i++) {
58         v[i] = 1;
59     }
60
61     cudaMemcpy(dev_v, v, sizeof(int) * size,
62               cudaMemcpyHostToDevice);
63     cudaMemcpy(sum, &sumGPU, sizeof(int),
64               cudaMemcpyHostToDevice);
65
66     dim3 grid(1);
67     dim3 block(size);
68
69     kernel <<< grid, block >>> (dev_v, sum);
70     cudaMemcpy(&sumGPU, sum, sizeof(int),
71               cudaMemcpyDeviceToHost);
72     printf("GPU sum: %d\n", sumGPU);
73
74     CPU_fn(v, &sumCPU, size);
75     printf("CPU sum: %d\n", sumCPU);
76
77     validate(&sumCPU, &sumGPU, size);
78
79     return 0;
80 }
```

Output



```
Microsoft Visual Studio Debug Console
GPU sum: 1024
CPU sum: 1024
Kernel validated successfully.

C:\Users\mariana\Documents\github-mariana\parallel-computing-cuda\11012021\ex01\x64\Debug\ex01.exe (process 17432) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 2: img