

Shared Memory with More than 1 Block

Imagine we launch four blocks in a kernel, each block with 8 threads, as shown below.

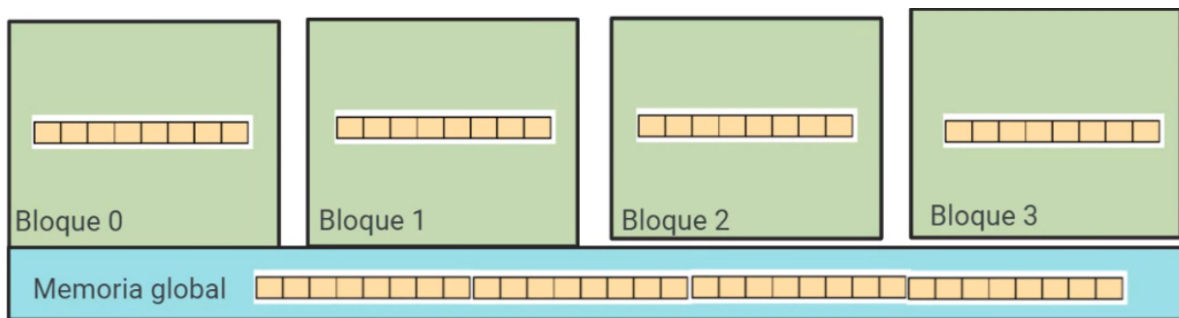


Figure 1: img

Thus, we will manipulate a vector of size 32 (4x8), the same size as the total threads launched.

In Global Memory, we have the vector thanks to `cudaMalloc()`. Global Memory is a memory that all threads from all blocks can see.

Last time, in the kernel we typed `__shared__ int vector[32]`, so that in a block there was a 32 vector in shared memory. But now, when launch many blocks the shared memory vector `__shared__ int vector[32]` will not be reserved 4 times, but the vector size will be split among all the blocks launched: each block has now a vector of size 8.

But now we need to copy from the Global Memory vector of size 32 into the shared vector size 8 in each block: the first block copies the first 8 values, and so on. The `gId` is computed:

```
int gId = threadIdx.x + blockDim.x * blockIdx.x;
```

So that it looks as follows:

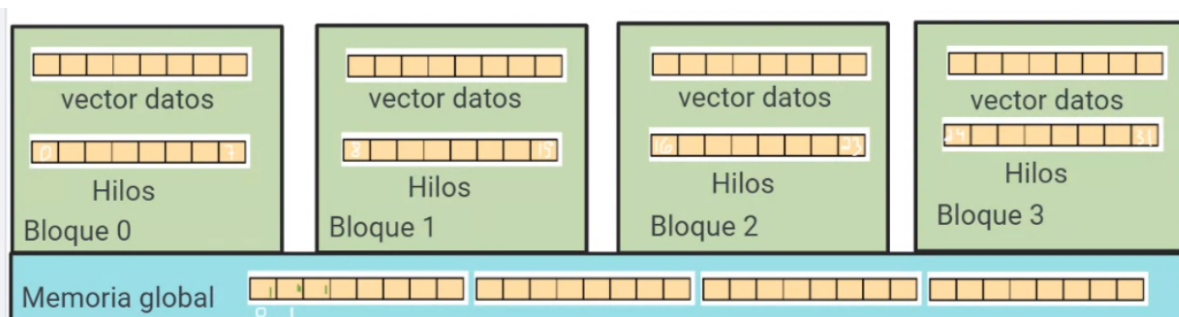


Figure 2: img

The block 1 cannot write anything on the shared vector of another block. Then, how to manage the shared vector indices in order to copy the global vector to the shared one?

The shared vector can be accessed with the **same** indices as if it was on global memory, because the reservation is global from the kernel, and thus the indices have continuity among blocks.

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5  }

```

Now, instead of partitioning the full 32 size vector, now we split into half the shared vector of size 8 that is in each block, and do the same thing over and over to perform a parallel reduction.

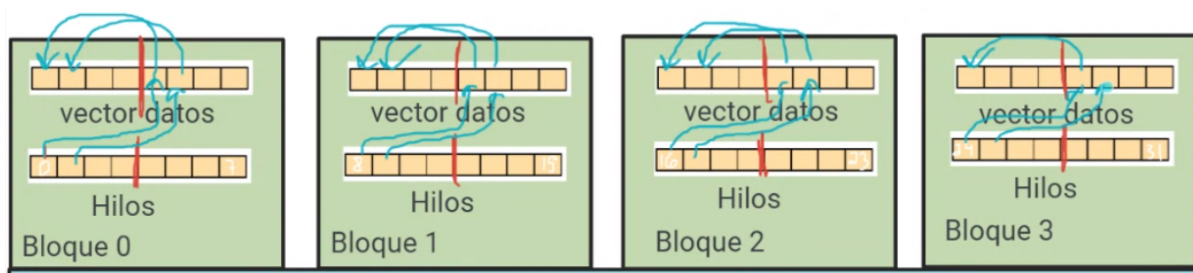


Figure 3: img

But what will be our step variable? Instead of $32 / 2$, we need $8 / 2$ initially. Consider that we have the following defined macros (that are known to the kernel):

```

1  #define numThreadsPerBlock 8
2  #define numBlocks 4
3  #define vecSize numThreadsPerBlock*numBlocks

```

Thus,

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5      int step = numThreadsPerBlock / 2;
6  }

```

and now, how to choose the first 4 threads of each block, in order to begin the parallel reduction step?



Figure 4: img

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5      int step = numThreadsPerBlock / 2;
6      if (threadIdx.x < step){ // the first 4 threads per block:
7          0 1 2 3
8          // use gId to move from block to block spaces in the
9          // shared vector (always the first 4)
10         // then add step to take the next half value and sum
11         // it up
12         vector[gId] += vector[gId + step];
13     }
14 }

```

But where is our final result? consider that the parallel reduction now is done per block:

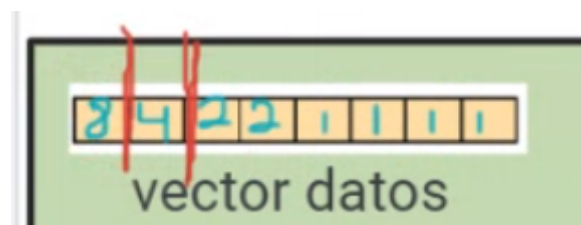


Figure 5: img

In the end, because the block is accessing only to its shared vector piece, thus the partial sum is in the first element of each block's vector of size 8, but we cannot access them across blocks. This is where the **global memory comes back**: we copy the number 8 in each block shared vector's first cell to the global memory vector, using the global id of such thread so that they write it in the global memory vector as:



Figure 6: img

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5      int step = numThreadsPerBlock / 2;
6      if (threadIdx.x < step){ // the first 4 threads per block:
7          0 1 2 3
8          vector[gId] += vector[gId + step];
9      }
10 }

```

```

9      if (threadIdx.x == 0){ // copy the partial results to the
      global mem vector
10         v[gId] = vector[gId];
11     }
12 }

```

Now back in the GM vector, we need to apply again a parallel reduction again, but to do that we need to copy all the 8's to the first 4 cells of the GM vector:



Figure 7: img

We will choose these 0 1 2 3 threads by choosing the threads with the gIds that are **smaller** than the number of blocks:

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5      int step = numThreadsPerBlock / 2;
6      if (threadIdx.x < step){ // the first 4 threads per block:
          0 1 2 3
7          vector[gId] += vector[gId + step];
8      }
9      if (threadIdx.x == 0){ // copy the partial results to the
      global mem vector
10         v[gId] = vector[gId];
11     }
12     if (gId < numBlocks){ // choose the first 4 threads to
      copy in the first 4 cells the partial sums
13         v[gId] = v[gId*numThreadsPerBlock]; // 0 <- 0*8, 1 <-
          1*8 ...
14     }
15 }

```

Now that we have the partial sums in the beginning cells of the GM vector, we need to add all these using parallel reduction. The new step now is:

```
int step = numBlocks / 2;
```

```

1  __global__ void kernel(int *v){
2      int gId = threadIdx.x + blockDim.x * blockIdx.x;
3      __shared__ int vector[32];
4      vector[gId] = v[gId];
5      // synch
6      int step = numThreadsPerBlock / 2;
7      if (threadIdx.x < step){ // the first 4 threads per block:
          0 1 2 3

```

```

8      vector[gId] += vector[gId + step];
9      // synch
10     }
11     if (threadIdx.x == 0){ // copy the partial results to the
        global mem vector
12         v[gId] = vector[gId];
13         // synch
14     }
15     if (gId < numBlocks){ // choose the first 4 threads to
        copy in the first 4 cells the partial sums
16         v[gId] = v[gId*numThreadsPerBlock]; // 0 <- 0*8, 1 <-
            1*8 ...
17         // apply parallel reduction over v
18         step = numBlocks / 2;
19         if (gId < step){
20             v[gId] += v[gId + step];
21         }
22     }
23     if (gId == 0){
24         *sum = v[0];
25     }
26 }

```

In the end now we have the total result of the sum in the first cell of the GM vector. Remember to synchronize threads after each shared memory vector usage.

Note: requisite for parallel reduction is that the vector size is an even number (divisible by 2)

Implementation

```

1  #include "cuda_runtime.h"
2  #include "device_launch_parameters.h"
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <iostream>
7
8  #define numBlocks 8
9  #define threadsPerBlock 1024
10
11 using namespace std;
12
13 __host__ void checkCUDAError(const char* msg) {
14     cudaError_t error;
15     cudaDeviceSynchronize();
16     error = cudaGetLastError();
17     if (error != cudaSuccess) {
18         printf("ERROR %d: %s (%s)\n", error,
            cudaGetErrorString(error), msg);

```

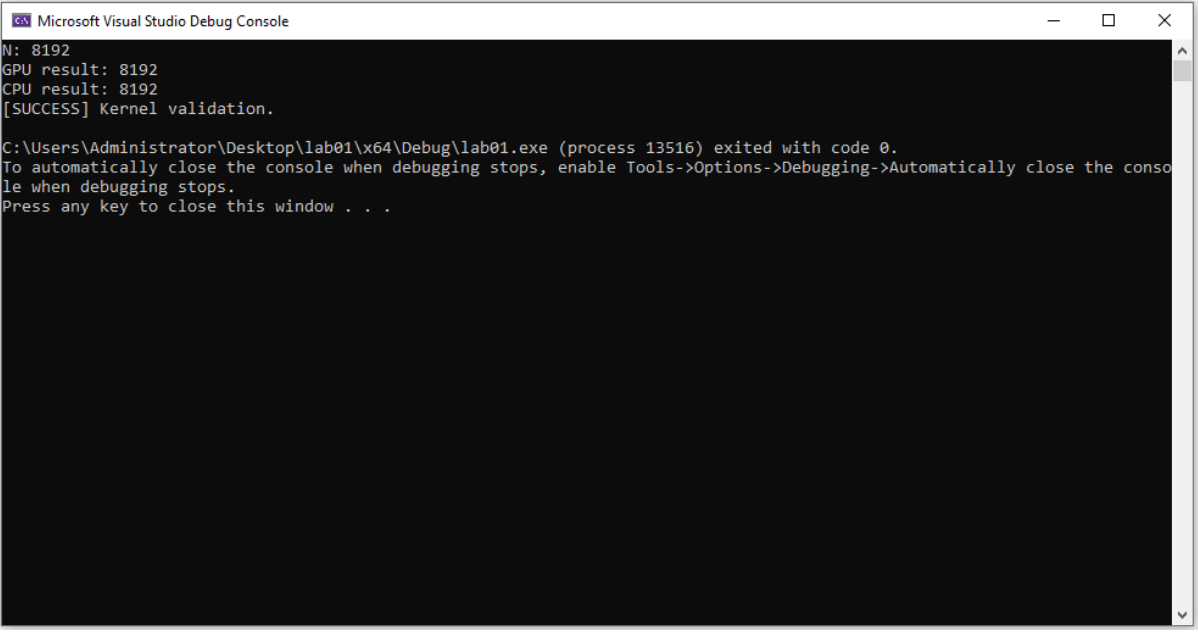
```

19     }
20 }
21
22 __host__ void validate(int* result_CPU, int* result_GPU) {
23     if (*result_CPU != *result_GPU) {
24         printf("[FAILED] Kernel validation.\n");
25         return;
26     }
27     printf("[SUCCESS] Kernel validation.\n");
28     return;
29 }
30
31 __host__ void CPU_reduction(int* v, int* sum) {
32     for (int i = 0; i < numBlocks * threadsPerBlock; i++) {
33         *sum += v[i];
34     }
35 }
36
37 __global__ void GPU_reduction(int* v, int* sum) {
38     __shared__ int vector[numBlocks * threadsPerBlock];
39     int gId = threadIdx.x + blockDim.x * blockIdx.x;
40
41     vector[gId] = v[gId];
42     __syncthreads();
43     int step = threadsPerBlock / 2;
44     while (step) {
45         if (threadIdx.x < step) {
46             vector[gId] = vector[gId] + vector[gId + step];
47             __syncthreads();
48         }
49         step = step / 2;
50     }
51     __syncthreads();
52     if (threadIdx.x == 0) { // copy the partial results to the
                           // global mem vector
53         //printf("SM->vector[%d]: %d\n", gId, vector[gId]);
54         v[gId] = vector[gId];
55         __syncthreads();
56         //printf("GM->v[%d]: %d\n", gId, v[gId]);
57     }
58     if (gId < numBlocks) { // choose the first 4 threads to
                           // copy in the first 4 cells the partial sums
59         v[gId] = v[gId * threadsPerBlock]; // 0 <- 0*8, 1 <-
                           // 1*8 ...
60         __syncthreads();
61         //printf("%d<-%d\n", v[gId], v[gId * threadsPerBlock])
62         ;
63     }
64     int new_step = numBlocks / 2;
65     while (new_step) {
66         if (gId < new_step) {

```

```
66         v[gId] += v[gId + new_step];
67     }
68     new_step = new_step / 2;
69 }
70 __syncthreads();
71 if (gId == 0) {
72     *sum = v[gId];
73 }
74 }
75
76 int main() {
77
78     int* dev_a, * dev_sum;
79     int host_sum = 0, CPU_sum = 0;
80     int* host_a = (int*)malloc(sizeof(int) * numBlocks *
81         threadsPerBlock);
82     cudaMalloc((void**)&dev_a, sizeof(int) * numBlocks *
83         threadsPerBlock);
84     cudaMalloc((void**)&dev_sum, sizeof(int));
85
86     for (int i = 0; i < numBlocks * threadsPerBlock; i++) {
87         host_a[i] = 1;
88     }
89
90     cudaMemcpy(dev_a, host_a, sizeof(int) * numBlocks *
91         threadsPerBlock, cudaMemcpyHostToDevice);
92
93     dim3 grid(numBlocks, 1, 1);
94     dim3 block(threadsPerBlock, 1, 1);
95     GPU_reduction << < grid, block >> > (dev_a, dev_sum);
96     cudaDeviceSynchronize();
97     checkCUDAError("Error at kernel");
98
99     printf("N: %d\n", numBlocks * threadsPerBlock);
100    cudaMemcpy(&host_sum, dev_sum, sizeof(int),
101        cudaMemcpyDeviceToHost);
102    printf("GPU result: %d\n", host_sum);
103
104    CPU_reduction(host_a, &CPU_sum);
105    printf("CPU result: %d\n", CPU_sum);
106
107    validate(&CPU_sum, &host_sum);
108
109    return 0;
110 }
```

Output



```
Microsoft Visual Studio Debug Console
N: 8192
GPU result: 8192
CPU result: 8192
[SUCCESS] Kernel validation.
C:\Users\Administrator\Desktop\lab01\x64\Debug\lab01.exe (process 13516) 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 8: img