Warp Divergence

Warp Divergence happens when you launch the threads in a warp to do **different things** among them. For example, if you, inside a kernel, differentiate the activities of the threads using their global ID and send, for example, the even threads to do this and the odds to do something else. Last time, in parallel reduction, you provoked this divergence.

The problem with this divergence is that it theoretically reduces **velocity by half** than the one your kernel would have if threads in a warp didn't diverge, **per if condition**: if you have one if statement, you reduce the speed by half; two if statements, you reduce that half speed by half again, and so on.

What can we do to avoid this? Instead of using the **gId**, use the **warpId** to divide the tasks among the threads. This is not warp divergence, because **all threads of each warp are doing the same thing**.

- Example

This has divergence:

```
if (gId % 2 == 0){
    // activity 1
} else {
    // activity 2
}
```

This has no divergence:

```
int warpId = gId / 32;
if (warpId % 2 == 0){
    // activity 1
} else {
    // activity 2
}
```

Especifically, in parallel reduction we cannot use this to avoid divergence. When you cannot avoid divergence, another alternative is to fix this divergence by using the space of memory called **Shared Memory**. Up until now, we have just used **global memory** when we have variables inside the kernel. The memory we used with `cudaMalloc()` was the memory called **Global Memory**.

Shared Memory is independent memory per block, where all threads in a block can access the memory of *only* the block they belong to. It is a faster memory in terms of access (read/write) when compared to global memory, because it is closer to the processor: therefore it is recommended when you have data that is constantly used or referenced inside a kernel. Parallel Redutcion is apt to this, because the elements of a vector are constantly referenced. The advise would be to store the vector in **Shared Memory**.

Nevertheless, we have less Shared Memory than the memory we have as Global Memory. You can check the capacity of these memories using the `cudaGetDeviceProperties()`, and it is given by block. Take care that your program does not surpass the capacity of Shared Memory when you write a kernel that requires it.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>

int main() {
    cudaDeviceProp prop;
    cudaGetDeviceProperties(&prop, 0);
    size_t sharedMemory = prop.sharedMemPerBlock;
    printf("sharedMemPerBlock: %zd bytes\n", sharedMemory); //
        %zd is used to print size_t values
}
```

Which outputs:

```
sharedMemPerBlock: 49152 bytes
```

Parallel Reduction With Divergence But Using Shared Memory Fix

Solution

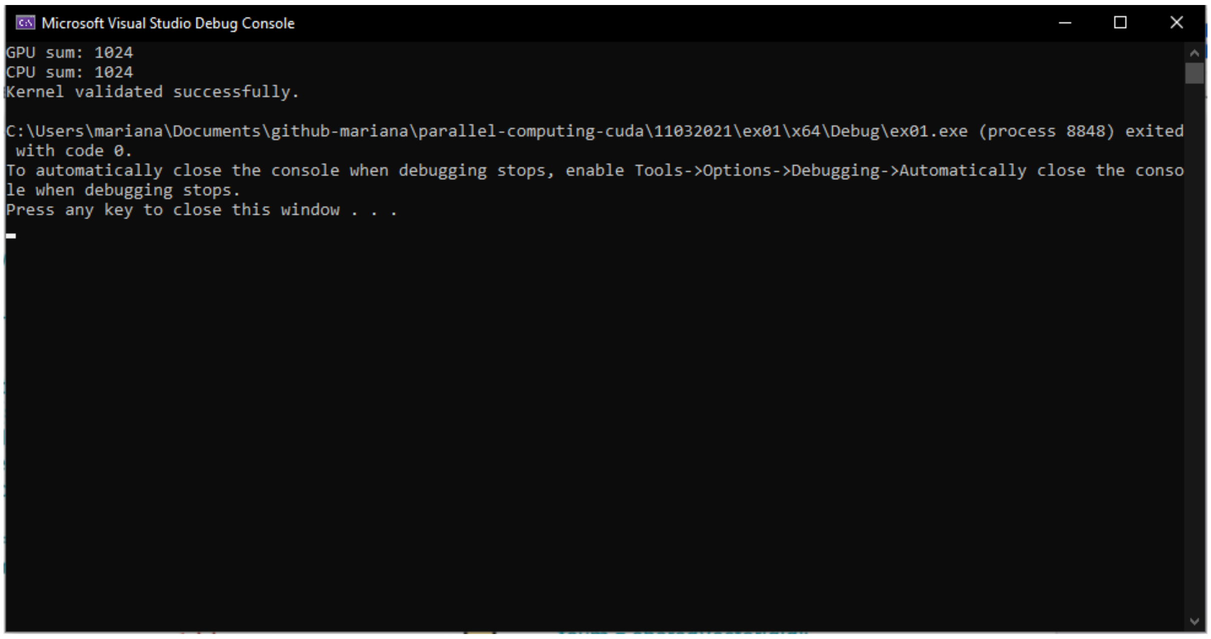
- 1D Grid with 1D block of 1024 threads in the x axis.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <stdlib.h>
#define vecSize 1024

__host__ void checkCUDAError(const char* msg) {
    cudaError_t error;
    cudaDeviceSynchronize();
    error = cudaGetLastError();
    if (error != cudaSuccess) {
        printf("ERROR %d: %s (%s)\n", error,
            cudaGetErrorString(error), msg);
    }
}

__host__ void validate(int* result_CPU, int* result_GPU, int
    size) {
```

```
    if (*result_CPU != *result_GPU) {
        printf("The results are not equal.\n");
        return;
    }
    printf("Kernel validated successfully.\n");
    return;
}

__host__ void CPU_fn(int* v, int* sum, const int size) {
    for (int i = 0; i < size; i++) {
        *sum += v[i];
    }
}

__global__ void kernel_divergent(int* v, int* sum) {
    int gId = threadIdx.x;
    int step = vecSize / 2;

    while (step) {
        if (gId < step) {
            v[gId] = v[gId] + v[gId + step];
        }
        step = step / 2;
    }
    if (gId == 0) {
        *sum = v[gId];
    }
}

__global__ void kernel_divergent_fixed(int* v, int* sum) {
    int gId = threadIdx.x;
    // vector in shared memory
    __shared__ int vectorShared[vecSize];
    vectorShared[gId] = v[gId];
    // old GPU's need thread synchronize when using shared
       memory: shared memory is visible for all blocks
    __syncthreads();
    int step = vecSize / 2; // avoid using blockDim to avoid
       unexpected behaviours

    while (step) {
        //printf("%d\n", step);
        if (gId < step) {
            vectorShared[gId] = vectorShared[gId] +
               vectorShared[gId + step];
        }
        step = step / 2;
    }
    if (gId == 0) {
        *sum = vectorShared[gId];
    }
```

```
}

int main() {

    const int size = 1024;
    int* v = (int*)malloc(sizeof(int) * size);
    int sumCPU = 0;
    int sumGPU = 0;

    int* dev_v, * sum;
    cudaMalloc((void**)&dev_v, sizeof(int) * size);
    cudaMalloc((void**)&sum, sizeof(int));

    for (int i = 0; i < size; i++) {
        v[i] = 1;
    }

    cudaMemcpy(dev_v, v, sizeof(int) * size,
       cudaMemcpyHostToDevice);
    cudaMemcpy(sum, &sumGPU, sizeof(int),
       cudaMemcpyHostToDevice);

    dim3 grid(1);
    dim3 block(size);

    kernel_divergent_fixed << < grid, block >> > (dev_v, sum);
    cudaMemcpy(&sumGPU, sum, sizeof(int),
       cudaMemcpyDeviceToHost);
    printf("GPU sum: %d\n", sumGPU);

    CPU_fn(v, &sumCPU, size);
    printf("CPU sum: %d\n", sumCPU);

    validate(&sumCPU, &sumGPU, size);

    return 0;
}
```

Output



Microsoft Visual Studio Debug Console

```
GPU sum: 1024
CPU sum: 1024
Kernel validated successfully.

C:\Users\mariana\Documents\github-mariana\parallel-computing-cuda\11032021\ex01\x64\Debug\ex01.exe (process 8848) exited
 with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure 1: img