



360.252 - COMPUTATIONAL SCIENCE ON MANY-CORE ARCHITECTURES

WS 2020 - EXERCISE 6

Christian GOLLMANN, 01435044

Last update: November 29, 2020

Contents

1	Dot Product with warp shuffles - shared memory	1
2	Dot Product with warp shuffles - warp shuffles	4
3	Dot Product with warp shuffles - performance comparison	6

1 Dot Product with warp shuffles - shared memory

I tested my code setting $N = 5$ which gave me the result shown in figure 1. I decided to implement everything with integers since the `atomicMin()` and `atomicMax()` would need some additional code snippets to work for double which, of course, I could have copied from the cuda documentation. But I wanted to avoid more overhead since I wanted to only focus on the principals here.

Compile output

[Compilation successful]

Run output

```
Input
-2|-1|0|1|2|
Sum of all entries: 0
Sum of maximum values: 6
Sum of squares: 10
Max-norm: 2
minimum value: -2
maximum value: 2
number of zeros: 1
```

Figure 1: Output of shared memory version

Listing 1: Calculations done with shared memory

```
1 #include <iostream>
2
3 --global-- void sharedMemoryKernel(const int* x, int* y, const int N) {
4     --shared-- int sharedMemory[7][256];
5     int sum = 0;
6     int maxSum = 0;
7     int sqrSum = 0;
8     int maxMod = 0;
9     int min = x[0];
10    int max = 0;
11    int zeros = 0;
12
13    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
14         tid < N; tid += gridDim.x * blockDim.x) {
15        int val = x[tid];
16
17        sum += val;
18        maxSum += std::abs(val);
19        sqrSum += val*val;
20        maxMod = std::abs(val) > maxMod ? val : maxMod;
21        min = val < min ? val : min;
22        max = val > max ? val : max;
23        zeros += val == 0 ? 1 : 0;
24    }
25
26    int tid = threadIdx.x;
27    if (tid < N) {
28        sharedMemory[0][threadIdx.x] = sum;
```

```

29     sharedMemory[1][threadIdx.x] = maxSum;
30     sharedMemory[2][threadIdx.x] = sqrSum;
31     sharedMemory[3][threadIdx.x] = maxMod;
32     sharedMemory[4][threadIdx.x] = min;
33     sharedMemory[5][threadIdx.x] = max;
34     sharedMemory[6][threadIdx.x] = zeros;
35
36     __syncthreads();
37     // blockDim.x needs to be a power of 2 in order for this to work
38     for (int i = blockDim.x/2; i != 0; i /= 2) {
39         __syncthreads();
40         if (tid < i) {
41             sharedMemory[0][tid] += sharedMemory[0][tid + i];
42             sharedMemory[1][tid] += sharedMemory[1][tid + i];
43             sharedMemory[2][tid] += sharedMemory[2][tid + i];
44             sharedMemory[3][tid] = sharedMemory[3][tid] > sharedMemory[3][tid + i]
45             ? sharedMemory[3][tid] : sharedMemory[3][tid + i];
46             sharedMemory[4][tid] = sharedMemory[4][tid] < sharedMemory[4][tid + i]
47             ? sharedMemory[4][tid] : sharedMemory[4][tid + i];
48             sharedMemory[5][tid] = sharedMemory[5][tid] > sharedMemory[5][tid + i]
49             ? sharedMemory[5][tid] : sharedMemory[5][tid + i];
50             sharedMemory[6][tid] += sharedMemory[6][tid + i];
51         }
52     }
53 }
54
55 if (tid == 0) {
56     atomicAdd(y, sharedMemory[0][0]);
57     atomicAdd(y+1, sharedMemory[1][0]);
58     atomicAdd(y+2, sharedMemory[2][0]);
59     atomicMax(y+3, sharedMemory[3][0]);
60     atomicMin(y+4, sharedMemory[4][0]);
61     atomicMax(y+5, sharedMemory[5][0]);
62     atomicAdd(y+6, sharedMemory[6][0]);
63 }
64 }
65
66 template <typename T>
67 void printContainer(T container, int N) {
68     for (int i = 0; i < N; i++) {
69         std::cout << container[i] << " | ";
70     }
71 }
72
73
74 int main() {
75
76     int N = 5;
77
78     int *x = (int *)malloc(sizeof(int) * N);
79     int *y = (int *)malloc(sizeof(int) * 7);
80
81     for (int i = 0; i < N; i++) {
82         x[i] = i - N/2;
83     }
84
85     int *cuda_x;

```

```

86     int *cuda_y;
87     cudaMalloc(&cuda_x, sizeof( int) * N);
88     cudaMalloc(&cuda_y, sizeof( int) * 7);
89
90     cudaMemcpy(cuda_x, x, sizeof( int) * N, cudaMemcpyHostToDevice);
91
92     sharedMemoryKernel<<<256, 256>>>(cuda_x, cuda_y, N);
93
94     cudaMemcpy(y, cuda_y, sizeof( int) * 7, cudaMemcpyDeviceToHost);
95
96     std::cout << "Input" << std::endl;
97     printContainer(x, N);
98     std::cout << std::endl;
99
100    std::cout << "Sum of all entries: " << y[0] << std::endl;
101    std::cout << "Sum of maximum values: " << y[1] << std::endl;
102    std::cout << "Sum of squares: " << y[2] << std::endl;
103    std::cout << "Max-norm: " << y[3] << std::endl;
104    std::cout << "minimum value: " << y[4] << std::endl;
105    std::cout << "maximum value: " << y[5] << std::endl;
106    std::cout << "number of zeros: " << y[6] << std::endl;
107
108    return EXIT_SUCCESS;
109 }

```

2 Dot Product with warp shuffles - warp shuffles

I tested my code by comparing to the version in point 1.

Listing 2: Calculations done with warp shuffles

```
1  #include <iostream>
2
3  __global__ void shuffleKernel(const int* x, int* y, const int N) {
4
5      int sum                = 0;
6      int maxSum             = 0;
7      int sqrSum             = 0;
8      int maxMod             = 0;
9      int min                = x[0];
10     int max                 = 0;
11     int zeros               = 0;
12
13     for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
14          tid < N; tid += gridDim.x * blockDim.x) {
15         int val = x[tid];
16
17         sum      += val;
18         maxSum   += std::abs(val);
19         sqrSum   += val*val;
20         maxMod   = std::abs(val) > maxMod ? val : maxMod;
21         min      = val < min ? val : min;
22         max      = val > max ? val : max;
23         zeros    += val == 0 ? 1 : 0;
24     }
25
26     int tid = threadIdx.x;
27     for (int i = warpSize / 2; i != 0; i /= 2) {
28         sum      += __shfl_down_sync(0xffffffff, sum, i);
29         maxSum   += __shfl_down_sync(0xffffffff, maxSum, i);
30         sqrSum   += __shfl_down_sync(0xffffffff, sqrSum, i);
31         int temporary = __shfl_down_sync(0xffffffff, maxMod, i);
32         maxMod   = temporary > maxMod ? temporary : maxMod;
33         temporary = __shfl_down_sync(0xffffffff, min, i);
34         min      = temporary < min ? temporary : min;
35         temporary = __shfl_down_sync(0xffffffff, max, i);
36         max      = temporary > max ? temporary : max;
37         zeros    += __shfl_down_sync(0xffffffff, zeros, i);
38     }
39     __syncthreads();
40     if (tid % warpSize == 0) {
41         atomicAdd(y, sum);
42         atomicAdd(y+1, maxSum);
43         atomicAdd(y+2, sqrSum);
44         atomicMax(y+3, maxMod);
45         atomicMin(y+4, min);
46         atomicMax(y+5, max);
47         atomicAdd(y+6, zeros);
48     }
49 }
50
51 template <typename T>
```

```

52 void printContainer(T container, int N) {
53     for (int i = 0; i < N; i++) {
54         std::cout << container[i] << " | ";
55     }
56 }
57
58
59 int main() {
60
61     int N = 100000;
62
63     int *x = (int *)malloc(sizeof(int) * N);
64     int *y = (int *)malloc(sizeof(int) * 7);
65
66     for (int i = 0; i < N; i++) {
67         x[i] = i - N/2;
68     }
69
70     int *cuda_x;
71     int *cuda_y;
72     cudaMalloc(&cuda_x, sizeof(int) * N);
73     cudaMalloc(&cuda_y, sizeof(int) * 7);
74
75     cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);
76
77     shuffleKernel<<<<N/256, 128>>>>(cuda_x, cuda_y, N);
78
79     cudaMemcpy(y, cuda_y, sizeof(int) * 7, cudaMemcpyDeviceToHost);
80
81     //std::cout << "Input" << std::endl;
82     //printContainer(x, N);
83     //std::cout << std::endl;
84
85     std::cout << "Sum of all entries: " << y[0] << std::endl;
86     std::cout << "Sum of maximum values: " << y[1] << std::endl;
87     std::cout << "Sum of squares: " << y[2] << std::endl;
88     std::cout << "Max-norm: " << y[3] << std::endl;
89     std::cout << "minimum value: " << y[4] << std::endl;
90     std::cout << "maximum value: " << y[5] << std::endl;
91     std::cout << "number of zeros: " << y[6] << std::endl;
92
93     return EXIT_SUCCESS;
94 }

```

3 Dot Product with warp shuffles - performance comparison

In order to compare the performances, I launched every kernel with $[N/256, 128]$. I additionally implemented the dot product in shared and shuffled version. The results can be found in figure 2. It can be seen that the shuffled versions perform a little better than the shared memory versions unless for one data point at the last N . I cannot really explain this behaviour. I doublechecked my tests and didn't find any errors in there.

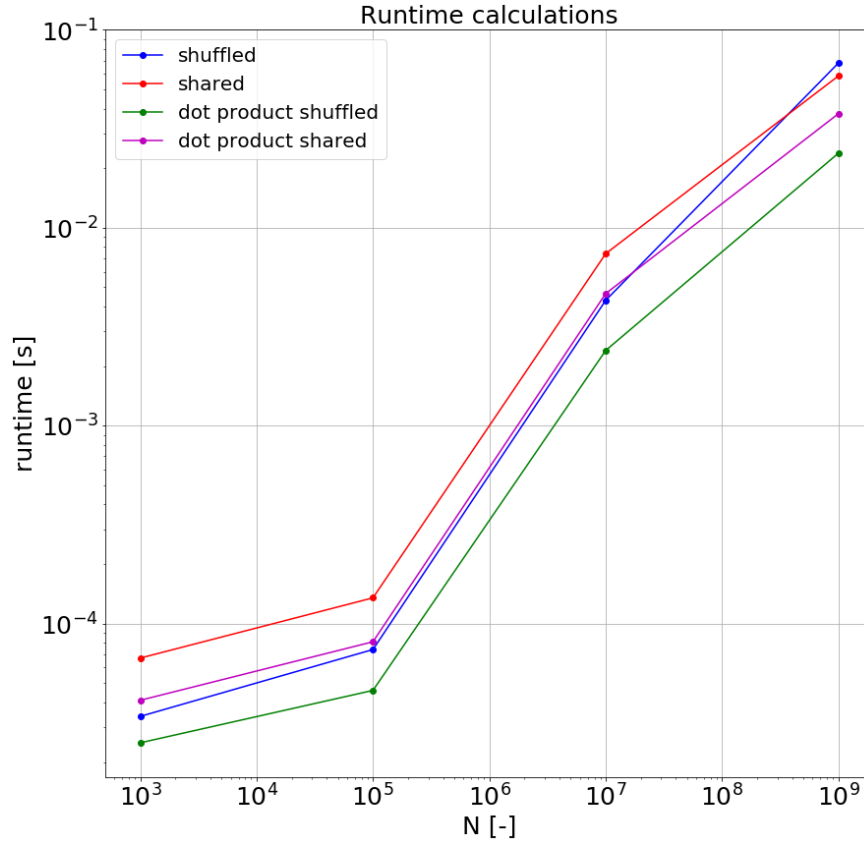


Figure 2: Runtimes for different versions

Listing 3: Dot product using shared memory

```

1  __global__ void dot_product(int* x, int* y, int* dot, int N) {
2
3      int index = threadIdx.x + blockDim.x * blockIdx.x;
4      int stride = blockDim.x * gridDim.x;
5
6      __shared__ int cache[128];
7
8      int temp = 0;
9      while (index < N) {
10         temp += x[index] * y[index];
11         index += stride;
12     }
13
14     cache[threadIdx.x] = temp;
15
16     __syncthreads();

```



```

17
18     for (int i = blockDim.x/2; i > 0; i/= 2) {
19         __syncthreads();
20         if (threadIdx.x < i)
21             cache[threadIdx.x] += cache[threadIdx.x + i];
22     }
23
24     if (threadIdx.x == 0)
25         atomicAdd(dot, cache[0]);
26
27 }

```

Listing 4: Dot product using warp shuffles

```

1  __global__ void dot_product_shuffle(int* x, int* y, int* dot, int N) {
2
3     int index = threadIdx.x + blockDim.x * blockIdx.x;
4     int stride = blockDim.x * gridDim.x;
5
6     int temp = 0;
7     while (index < N) {
8         temp += x[index] * y[index];
9         index += stride;
10    }
11
12    for (int i = warpSize / 2; i != 0; i /= 2) {
13        temp += __shfl_down_sync(0xffffffff, temp, i);
14    }
15
16    __syncthreads();
17
18    int tid = threadIdx.x;
19
20    if (tid % warpSize == 0) {
21        atomicAdd(dot, temp);
22    }
23
24 }

```
