360.252 - Computational Science on Many-Core Architectures

WS 2020 - Exercise 6

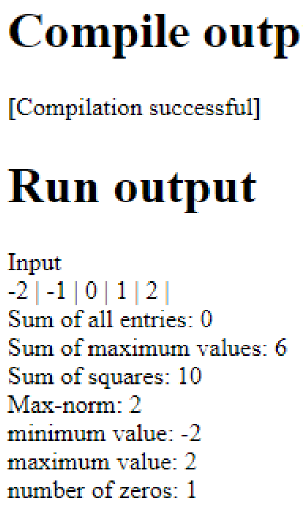
Christian Gollmann, 01435044

Last update: November 29, 2020

Contents

1 Dot Product with warp shuffles - shared memory

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

```
#include <iostream>

__global__ void sharedMemoryKernel(const int* x, int* y, const int N) {
    __shared__ int sharedMemory[7][256];
     int sum = 0;
     int maxSum = 0;
     int sqrSum = 0;
     int maxMod = 0;
     int min = x[0];
     int max = 0;
     int zeros = 0;

    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
    tid < N; tid += gridDim.x * blockDim.x) {
        int val = x[tid];

        sum += val;
        maxSum += std::abs(val);
        sqrSum += val*val;
        maxMod = std::abs(val) > maxMod ? val : maxMod;
        min = val < min ? val : min;
        max = val > max ? val :max;
        zeros += val == 0 ? 1 : 0;
    }

    int tid = threadIdx.x;
    if (tid < N) {
        sharedMemory[0][threadIdx.x] = sum;
```

```
            sharedMemory[1][threadIdx.x] = maxSum;
            sharedMemory[2][threadIdx.x] = sqrSum;
            sharedMemory[3][threadIdx.x] = maxMod;
            sharedMemory[4][threadIdx.x] = min;
            sharedMemory[5][threadIdx.x] = max;
            sharedMemory[6][threadIdx.x] = zeros;

            __syncthreads();
            // blockDim.x needs to be a power of 2 in order for this to work
            for (int i = blockDim.x/2; i != 0; i /= 2) {
                __syncthreads();
                if (tid < i) {
                    sharedMemory[0][tid] += sharedMemory[0][tid + i];
                    sharedMemory[1][tid] += sharedMemory[1][tid + i];
                    sharedMemory[2][tid] += sharedMemory[2][tid + i];
                    sharedMemory[3][tid] = sharedMemory[3][tid] > sharedMemory[3][tid + i]
                    ? sharedMemory[3][tid] : sharedMemory[3][tid + i];
                    sharedMemory[4][tid] = sharedMemory[4][tid] < sharedMemory[4][tid + i]
                    ? sharedMemory[4][tid] : sharedMemory[4][tid + i];
                    sharedMemory[5][tid] = sharedMemory[5][tid] > sharedMemory[5][tid + i]
                    ? sharedMemory[5][tid] : sharedMemory[5][tid + i];
                    sharedMemory[6][tid] += sharedMemory[6][tid + i];
                }
            }
        }

        if (tid == 0) {
            atomicAdd(y, sharedMemory[0][0]);
            atomicAdd(y+1, sharedMemory[1][0]);
            atomicAdd(y+2, sharedMemory[2][0]);
            atomicMax(y+3, sharedMemory[3][0]);
            atomicMin(y+4, sharedMemory[4][0]);
            atomicMax(y+5, sharedMemory[5][0]);
            atomicAdd(y+6, sharedMemory[6][0]);
        }
}

template <typename T>
 void printContainer(T container, int N) {
     for (int i = 0; i < N; i++) {
         std::cout << container[i] << " | ";
     }
}


int main() {

    int N = 5;

     int *x = (int *)malloc(sizeof(int) * N);
     int *y = (int *)malloc(sizeof(int) * 7);

    for (int i = 0; i < N; i++) {
        x[i] = i - N/2;
    }

     int *cuda_x;
```

```
     int *cuda_y;
    cudaMalloc(&cuda_x, sizeof( int) * N);
    cudaMalloc(&cuda_y, sizeof( int) * 7);

    cudaMemcpy(cuda_x, x, sizeof( int) * N, cudaMemcpyHostToDevice);

    sharedMemoryKernel<<<256, 256>>>(cuda_x, cuda_y, N);

    cudaMemcpy(y, cuda_y, sizeof( int) * 7, cudaMemcpyDeviceToHost);

    std::cout << "Input" << std::endl;
    printContainer(x, N);
    std::cout << std::endl;

    std::cout << "Sum of all entries: " << y[0] << std::endl;
    std::cout << "Sum of maximum values: " << y[1] << std::endl;
    std::cout << "Sum of squares: " << y[2] << std::endl;
    std::cout << "Max-norm: " << y[3] << std::endl;
    std::cout << "minimum value: " << y[4] << std::endl;
    std::cout << "maximum value: " << y[5] << std::endl;
    std::cout << "number of zeros: " << y[6] << std::endl;

    return EXIT_SUCCESS;
}
```

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

```
#include <iostream>

__global__ void shuffleKernel(const int* x, int* y, const int N) {

    int sum           = 0;
    int maxSum        = 0;
    int sqrSum        = 0;
    int maxMod        = 0;
    int min           = x[0];
    int max           = 0;
    int zeros         = 0;

    for (int tid = blockDim.x * blockIdx.x + threadIdx.x;
    tid < N; tid += gridDim.x * blockDim.x) {
        int val = x[tid];

        sum         += val;
        maxSum      += std::abs(val);
        sqrSum      += val*val;
        maxMod      = std::abs(val) > maxMod ? val : maxMod;
        min         = val < min ? val : min;
        max         = val > max ? val :max;
        zeros       += val == 0 ? 1 : 0;
    }

    int tid = threadIdx.x;
    for (int i = warpSize / 2; i != 0; i /= 2) {
        sum         += __shfl_down_sync(0xffffffff, sum, i);
        maxSum      += __shfl_down_sync(0xffffffff, maxSum, i);
        sqrSum      += __shfl_down_sync(0xffffffff, sqrSum, i);
        int temporary = __shfl_down_sync(0xffffffff, maxMod, i);
        maxMod      = temporary > maxMod ? temporary : maxMod;
        temporary   = __shfl_down_sync(0xffffffff, min, i);
        min         = temporary < min ? temporary : min;
        temporary   = __shfl_down_sync(0xffffffff, max, i);
        max         = temporary > max ? temporary : max;
        zeros       += __shfl_down_sync(0xffffffff, zeros, i);
    }
    __syncthreads();
    if (tid % warpSize == 0) {
        atomicAdd(y,   sum);
        atomicAdd(y+1, maxSum);
        atomicAdd(y+2, sqrSum);
        atomicMax(y+3, maxMod);
        atomicMin(y+4, min);
        atomicMax(y+5, max);
        atomicAdd(y+6, zeros);
    }
}

template <typename T>
```

```
void printContainer(T container, int N) {
    for (int i = 0; i < N; i++) {
        std::cout << container[i] << " | ";
    }
}


int main() {

    int N = 100000;

    int *x = (int *)malloc(sizeof(int) * N);
    int *y = (int *)malloc(sizeof(int) * 7);

    for (int i = 0; i < N; i++) {
        x[i] = i - N/2;
    }

    int *cuda_x;
    int *cuda_y;
    cudaMalloc(&cuda_x, sizeof(int) * N);
    cudaMalloc(&cuda_y, sizeof(int) * 7);

    cudaMemcpy(cuda_x, x, sizeof(int) * N, cudaMemcpyHostToDevice);

    shuffleKernel<<<N/256, 128>>>(cuda_x, cuda_y, N);

    cudaMemcpy(y, cuda_y, sizeof(int) * 7, cudaMemcpyDeviceToHost);

    //std::cout << "Input" << std::endl;
    //printContainer(x, N);
    //std::cout << std::endl;

    std::cout << "Sum of all entries: " << y[0] << std::endl;
    std::cout << "Sum of maximum values: " << y[1] << std::endl;
    std::cout << "Sum of squares: " << y[2] << std::endl;
    std::cout << "Max-norm: " << y[3] << std::endl;
    std::cout << "minimum value: " << y[4] << std::endl;
    std::cout << "maximum value: " << y[5] << std::endl;
    std::cout << "number of zeros: " << y[6] << std::endl;

    return EXIT_SUCCESS;
}
```

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.

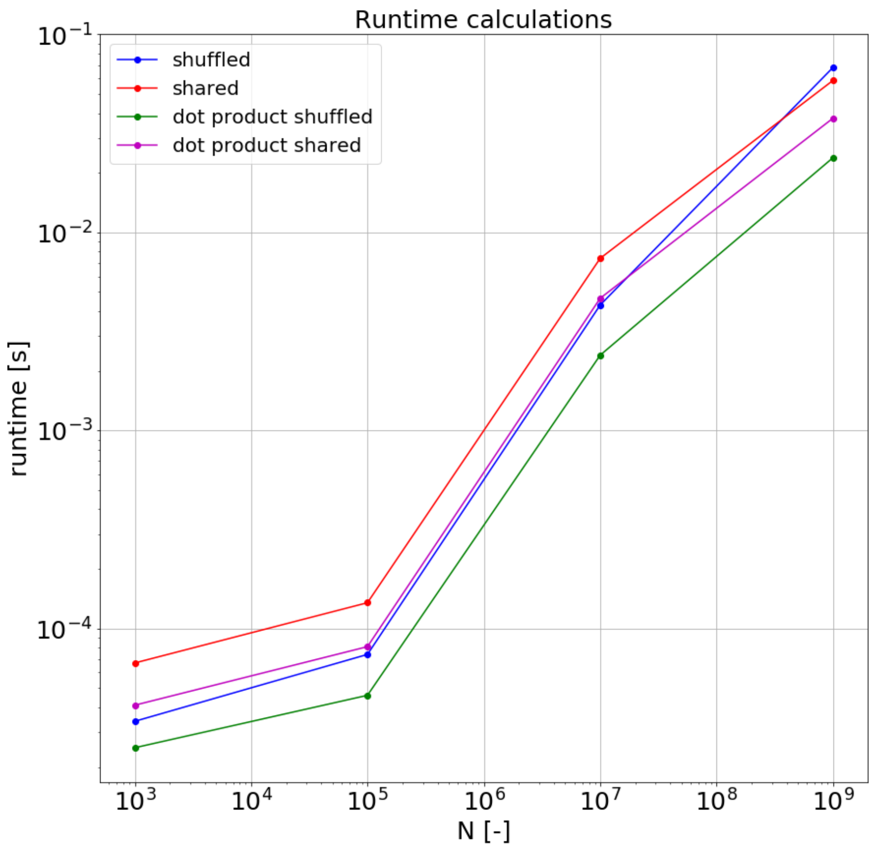


Figure 2: Runtimes for different versions

Listing 3: Dot product using shared memory

```
__global__ void dot_product(int* x, int* y, int* dot, int N) {

    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;

    __shared__ int cache[128];

    int temp = 0;
    while (index < N) {
        temp += x[index] * y[index];
        index += stride;
    }

    cache[threadIdx.x] = temp;

    __syncthreads();
```

```

    for (int i = blockDim.x/2; i > 0; i/= 2) {
        __syncthreads();
        if (threadIdx.x < i)
            cache[threadIdx.x] += cache[threadIdx.x + i];
    }

    if (threadIdx.x == 0)
        atomicAdd(dot, cache[0]);

}
```

Listing 4: Dot product using warp shuffles

```
__global__ void dot_product_shuffle(int* x, int* y, int* dot, int N) {

    int index = threadIdx.x + blockDim.x * blockIdx.x;
    int stride = blockDim.x * gridDim.x;

    int temp = 0;
    while (index < N) {
        temp += x[index] * y[index];
        index += stride;
    }

    for (int i = warpSize / 2; i != 0; i /= 2) {
        temp        += __shfl_down_sync(0xffffffff, temp, i);
    }

    __syncthreads();

    int tid = threadIdx.x;

    if (tid % warpSize == 0) {
        atomicAdd(dot,    temp);
    }

}
```