Vienna University of Technology

360.252 Computational Science on Many Core Architectures

Institute for Microelectronics

Exercise 4

Authors:
Camilo Tello Fachin
12127084

Supervisor:
Dipl.-Ing. Dr.techn. Karl Rupp

November 15, 2022

TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

Abstract

Here documented the results of exercise 4.

Contents

1 Dot Product with Warp Shuffles (3/4 Points) **1**

1.1 Task 1a 1

1.2 Task 1b 2

1.3 Task 1c 2

2 Multiple Dot Products (4 Points total) **3**

2.1 Task 2a 3

2.2 Task 2b 5

2.3 Task 2c 8

2.4 Task 2d 8

References **8**

1 Dot Product with Warp Shuffles (3/4 Points)

1.1 Task 1a

In The listing below see the code part for the Cuda Kernel for Exercise 4.1.a

C++ Cuda Code for 1a Kernel

```
// result  = (sum, abssum, squares, zero)
__global__ void cuda_1a(int N, double *x, double *sum, double *abssum, double *squares, double *zeros)
{
   __shared__ double shared_mem_sum[512];
   __shared__ double shared_mem_abssum[512];
   __shared__ double shared_mem_squares[512];
   __shared__ double shared_mem_zeros[512];

  double sum_thr = 0;
  double abssum_thr = 0;
  double squares_thr  = 0;
  double zeros_thr  = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x;  i < N; i += blockDim.x * gridDim.x) {
    sum_thr  += x[i];
    abssum_thr += abs(x[i]);
     squares_thr  += pow(x[i],2);
     zeros_thr  += (double)x[i]==0;
  }

  shared_mem_sum[threadIdx.x] = sum_thr;
  shared_mem_abssum[threadIdx.x] = abssum_thr;
  shared_mem_squares[threadIdx.x]  = squares_thr;
  shared_mem_zeros[threadIdx.x]  = zeros_thr;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_sum[threadIdx.x] += shared_mem_sum[threadIdx.x + k];
      shared_mem_abssum[threadIdx.x] += shared_mem_abssum[threadIdx.x + k];
      shared_mem_squares[threadIdx.x]  += shared_mem_squares[threadIdx.x + k];
      shared_mem_zeros[threadIdx.x]  += shared_mem_zeros[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0) {
    atomicAdd(sum, shared_mem_sum[0]);
    atomicAdd(abssum, shared_mem_abssum[0]);
    atomicAdd(squares, shared_mem_squares[0]);
    atomicAdd(zeros, shared_mem_zeros[0]);

  }
}
```

1.2 Task 1b

In the b part of Exercise 1, the warpshuffles had to be made use of instead of the shared memory. See below the listing with the code snippet for the CUDA Kernel with warpshuffles.

C++ Cuda Code for 1b Kernel

```
__global__ void cuda_1b(int N, double *x, double *sum, double *abssum, double *squares, double *zeros)
{
  double sum_thr = 0;
  double abssum_thr = 0;
  double squares_thr  = 0;
  double zeros_thr  = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    sum_thr  += x[i];
    abssum_thr += abs(x[i]);
     squares_thr  += pow(x[i],2);
     zeros_thr  += (double)x[i]==0;
  }

  for (int i=warpSize/2; i>0; i=i/2){
      sum_thr += __shfl_down_sync(-1, sum_thr, i);
      abssum_thr += __shfl_down_sync(-1, abssum_thr, i);
       squares_thr  += __shfl_down_sync(-1, squares_thr,  i);
       zeros_thr  += __shfl_down_sync(-1, zeros_thr,  i);
  }

    if ((threadIdx.x &(warpSize-1))== 0) {
    atomicAdd(sum, sum_thr);
    atomicAdd(abssum, abssum_thr);
    atomicAdd(squares,  squares_thr);
    atomicAdd(zeros,  zeros_thr);
    }
  }
```

1.3 Task 1c

In exercise 4.1.b, the same had to be done as before, but with warp shuffles only with one thread per entry. This can be done by changing the arguments of the Kernel invocation.

C++ Cuda Code for 1c Kernel Invocation

```
    cuda_1c<<<((N+255)/256), 256>>>(N, cuda_x, cuda_sum, cuda_abssum, cuda_squares, cuda_zeros);
```

2 Multiple Dot Products (4 Points total)

2.1 Task 2a

In this Section it was asked specifically to ”write a Kernel”, so that is what you will find below, just a Kernel without any useful implementation!

C++ Cuda Code Kernel for 2a

```
__global__ void cuda_2a(int N, double *x, double *y, double *result)
{
  __shared__ double shared_m1[256];
  __shared__ double shared_m2[256];
  __shared__ double shared_m3[256];
  __shared__ double shared_m4[256];
  __shared__ double shared_m5[256];
  __shared__ double shared_m6[256];
  __shared__ double shared_m7[256];
  __shared__ double shared_m8[256];

  double dot1=0, dot2=0, dot3=0, dot4=0, dot5=0, dot6=0, dot7=0, dot8=0;
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += blockDim.x * gridDim.x) {
    dot1 += x[i] * y[i];
    dot2 += x[i] * y[i+N];
    dot3 += x[i] * y[i+2*N];
    dot4 += x[i] * y[i+3*N];
    dot5 += x[i] * y[i+4*N];
    dot6 += x[i] * y[i+5*N];
    dot7 += x[i] * y[i+6*N];
    dot8 += x[i] * y[i+7*N];
  }

  shared_m1[threadIdx.x] = dot1;
  shared_m2[threadIdx.x] = dot2;
  shared_m3[threadIdx.x] = dot3;
  shared_m4[threadIdx.x] = dot4;
  shared_m5[threadIdx.x] = dot5;
  shared_m6[threadIdx.x] = dot6;
  shared_m7[threadIdx.x] = dot7;
  shared_m8[threadIdx.x] = dot8;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_m1[threadIdx.x] += shared_m1[threadIdx.x + k];
      shared_m2[threadIdx.x] += shared_m2[threadIdx.x + k];
      shared_m3[threadIdx.x] += shared_m3[threadIdx.x + k];
      shared_m4[threadIdx.x] += shared_m4[threadIdx.x + k];
      shared_m5[threadIdx.x] += shared_m5[threadIdx.x + k];
      shared_m6[threadIdx.x] += shared_m6[threadIdx.x + k];
      shared_m7[threadIdx.x] += shared_m7[threadIdx.x + k];
      shared_m8[threadIdx.x] += shared_m8[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0){
    atomicAdd(&result[0], shared_m1[0]);
    atomicAdd(&result[1], shared_m2[0]);
    atomicAdd(&result[2], shared_m3[0]);
    atomicAdd(&result[3], shared_m4[0]);
    atomicAdd(&result[4], shared_m5[0]);
    atomicAdd(&result[5], shared_m6[0]);
    atomicAdd(&result[6], shared_m7[0]);
    atomicAdd(&result[7], shared_m8[0]);
  }
}
```

2.2 Task 2b

"Add a loop around that kernel to call it K/8 times to correctly compute the K dot products" This was achieved by adjusting the algorithm in the following way:

C++ Cuda Code adaptation 2b

```
void Loop_function(int N, int K, double *x, double **y, double **results)
{
    int h= K/8;
    for (; h>0; h--)
    {
        cuda_task2<<<256, 256>>>(N, x, y[h-1], results[h-1]);
    }
}

float avg_vec(std::vector<float> timings_vec)
{
    float avg = std::accumulate(timings_vec.begin(), timings_vec.end(), 0.0);
    avg /= timings_vec.size();
    return avg;
}

int main(void)
{

  Timer timer;

  for (int N = 1000; N < 1000001; N *= 10)
  {
      for (size_t K = 8; K < 33; K += 8)
      {

      std::vector<float> timings;

      //
      // allocate host memory:
      //

      double *x = (double*)malloc(sizeof(double) * N);
      double *results   = (double*)malloc(sizeof(double) * K);
      double **results3  = (double**)malloc(sizeof(double) * K/8);
      for (size_t i=0; i<K/8; ++i) {
        results3[i] = (double*)malloc(sizeof(double) * 8);
      }

      double **v = (double**)malloc(sizeof(double*) * K/8);
      for (size_t i=0; i<K/8; ++i) {
        v[i] = (double*)malloc(sizeof(double) * N*8);
      }

      //
    // allocate device memory
    //

    double *cuda_x; cudaMalloc((&cuda_x), sizeof(double)*N);
    double **cuda_v = (double**)malloc(sizeof(double*) * K/8);
    for (size_t i=0; i<K/8; ++i) {
      cudaMalloc( (void **)(&cuda_v[i]), sizeof(double)*N*8);
    }

    double **cuda_results3 = (double**)malloc(sizeof(double*) * K/8);
    for (size_t i=0; i<K/8; ++i) {
      cudaMalloc( (void **)(&cuda_results3[i]), sizeof(double)*8);
    }

    //
    // initialize v
    //

    std::fill(x, x + N, 1.0);
    for (size_t i=0; i<K/8; ++i) {
      for (size_t j=0; j<N*8; ++j) {
        v[i][j] = 1 + rand() / (1.1 * RAND_MAX);
      }
    }

    //
    // Reference calculation on CPU:
    //

    for (size_t i=0; i<K; ++i) {results[i]=0;}
    // dot product
    for (size_t i=0; i<K/8; ++i) {
      for (size_t j=0; j<N*8; ++j) {
        results[i*8+j/N] += x[j%N] * v[i][j];
      }
    }

    //
    // Copy data to GPU
    //

    cudaMemcpy(cuda_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);
    for (size_t i=0; i<K/8; ++i) {
      cudaMemcpy(cuda_v[i], v[i], sizeof(double)*N*8, cudaMemcpyHostToDevice);
    }

    for (int i = 0; i < 11; i++)
    {
      //
      // CUDA implementation
      //

      CUDA_ERRCHK(cudaDeviceSynchronize());
      timer.reset();
      Loop_function(N, K, cuda_x, cuda_v, cuda_results3);
      CUDA_ERRCHK(cudaDeviceSynchronize());
      float elapsed_time_0 = timer.get();
      if (i > 0)
      {
          timings.push_back(elapsed_time_0);
      }

    }

    std::cout << N << "," << K << "," << avg_vec(timings) << std::endl;

    //
    // Copy data to Host
    //

    for (size_t i=0; i<K/8; ++i) {
      cudaMemcpy(results3[i], cuda_results3[i], sizeof(double)*8, cudaMemcpyDeviceToHost);
    }
```

2.3 Task 2c

In figure 1 below, the plot for task 2c.

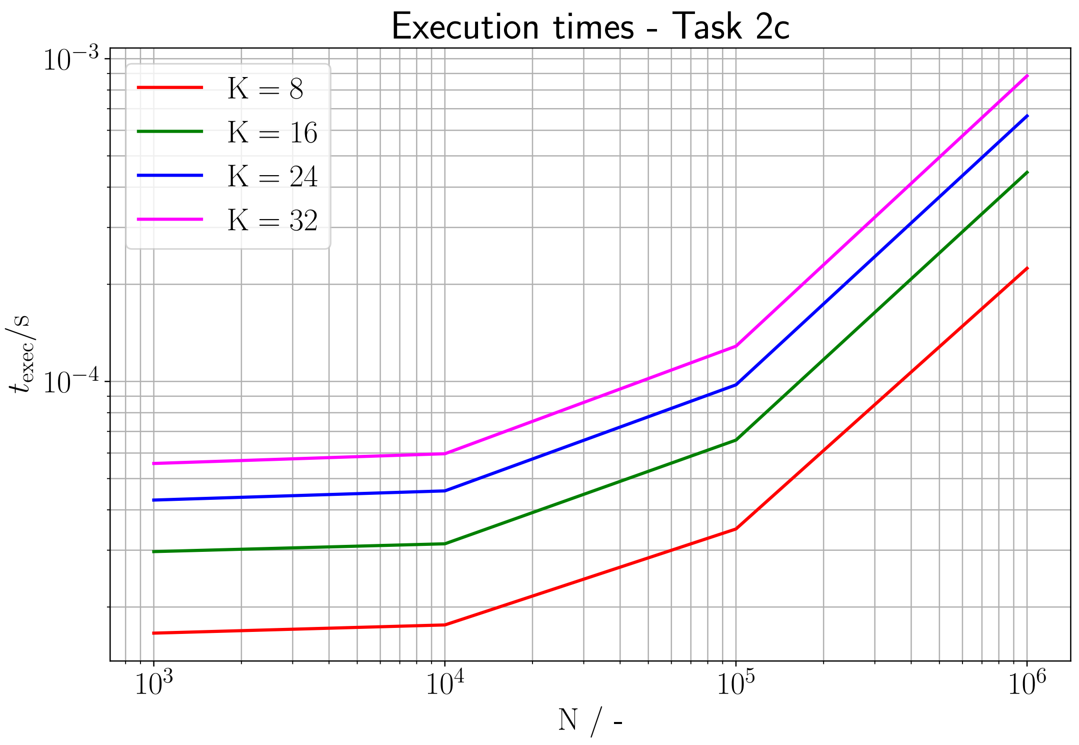

Figure 1: Execution Times of concurrent dot products for different K and N

2.4 Task 2d

By checking if k mod(8) is 0, if not find a scheme to additionally start another kernel for a single dot product to balance out, otherwise proceed as before.