# 360.252 - Computational Science on Many-Core Architectures

## WS 2020 - Exercise 4

Christian Gollmann, 01435044

Last update: November 14, 2020

## Contents

# 1 Multiple Dot Products 1

The following kernel computes 8 dot products simultaneously. I first wanted to go for some kind of 2D-array which I pass to the kernel but somehow I wasn't able to make that work, I always had segmentation failures. So I decided to go with this rather clumsy implementation which gets the job done as well in this case. However, I will reflect more on what one could to make this more efficient at the end of this exercise point.

Listing 1: Kernel to compute 8 dot products simultaneously

```
__global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
double *y2, double *y3, double *y4, double *y5, double *y6, double *y7,
double *result)
{
  __shared__ double shared_mem_0[512];
  __shared__ double shared_mem_1[512];
  __shared__ double shared_mem_2[512];
  __shared__ double shared_mem_3[512];
  __shared__ double shared_mem_4[512];
  __shared__ double shared_mem_5[512];
  __shared__ double shared_mem_6[512];
  __shared__ double shared_mem_7[512];

  double dot_0 = 0;
  double dot_1 = 0;
  double dot_2 = 0;
  double dot_3 = 0;
  double dot_4 = 0;
  double dot_5 = 0;
  double dot_6 = 0;
  double dot_7 = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x;
  i < N; i += blockDim.x * gridDim.x) {
    double val = x[i];
    dot_0 += val * y0[i];
    dot_1 += val * y1[i];;
    dot_2 += val * y2[i];;
    dot_3 += val * y3[i];;
    dot_4 += val * y4[i];;
    dot_5 += val * y5[i];;
    dot_6 += val * y6[i];;
    dot_7 += val * y7[i];;
  }

  shared_mem_0[threadIdx.x] = dot_0;
  shared_mem_1[threadIdx.x] = dot_1;
  shared_mem_2[threadIdx.x] = dot_2;
  shared_mem_3[threadIdx.x] = dot_3;
  shared_mem_4[threadIdx.x] = dot_4;
  shared_mem_5[threadIdx.x] = dot_5;
  shared_mem_6[threadIdx.x] = dot_6;
  shared_mem_7[threadIdx.x] = dot_7;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
```

```
    if (threadIdx.x < k) {
      shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
      shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
      shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
      shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
      shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
      shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
      shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
      shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0){
      atomicAdd(result+0, shared_mem_0[0]);
      atomicAdd(result+1, shared_mem_1[0]);
      atomicAdd(result+2, shared_mem_2[0]);
      atomicAdd(result+3, shared_mem_3[0]);
      atomicAdd(result+4, shared_mem_4[0]);
      atomicAdd(result+5, shared_mem_5[0]);
      atomicAdd(result+6, shared_mem_6[0]);
      atomicAdd(result+7, shared_mem_7[0]);
  }
}
```

## 2 Multiple Dot Products 2 + 3

It can be seen that the runtimes are very, let's call it regular, and all follow the same trend. (K=32) takes approximately four times as long as (K=8) which I kind of expected. Also do they grow as N grows, so no really big surprises here.

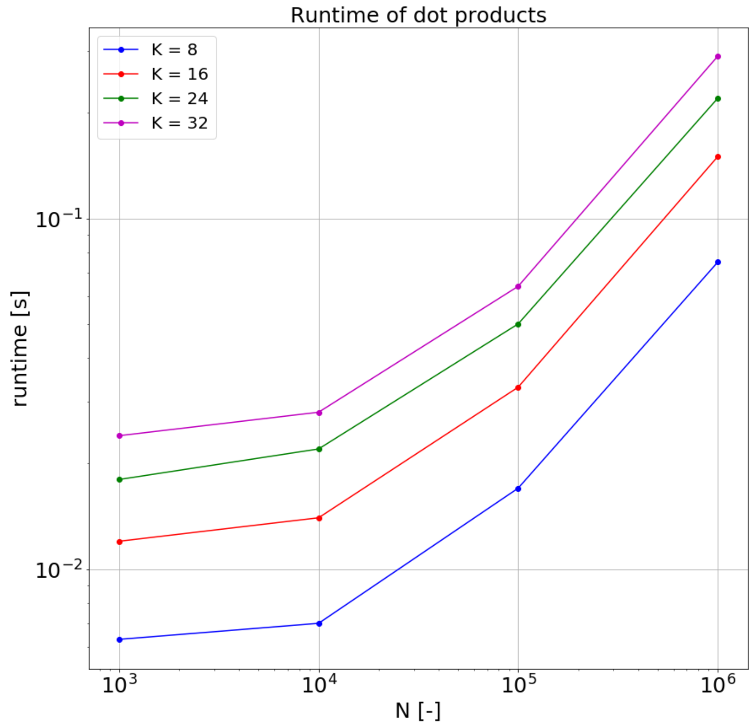


Figure 1: Runtime for dot product of vectors with N entries

Listing 2: Code for points 2 and 3

```
#include <cuda_runtime.h>
#include <cublas_v2.h>
#include <stdio.h>
#include <cmath>
#include <iostream>
#include "timer.hpp"
#include <algorithm>
#include <vector>

 __global__ void cuda_many_dot_product(int N, double *x, double *y0, double *y1,
 double *y2, double *y3, double *y4, double *y5, double *y6,
 double *y7, double *result)
{
  __shared__ double shared_mem_0[512];
  __shared__ double shared_mem_1[512];
  __shared__ double shared_mem_2[512];
  __shared__ double shared_mem_3[512];
  __shared__ double shared_mem_4[512];
  __shared__ double shared_mem_5[512];
  __shared__ double shared_mem_6[512];
  __shared__ double shared_mem_7[512];

  double dot_0 = 0;
```

```
  double dot_1 = 0;
  double dot_2 = 0;
  double dot_3 = 0;
  double dot_4 = 0;
  double dot_5 = 0;
  double dot_6 = 0;
  double dot_7 = 0;

  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N;
  i += blockDim.x * gridDim.x) {
    double val = x[i];
    dot_0 += val * y0[i];
    dot_1 += val * y1[i];;
    dot_2 += val * y2[i];;
    dot_3 += val * y3[i];;
    dot_4 += val * y4[i];;
    dot_5 += val * y5[i];;
    dot_6 += val * y6[i];;
    dot_7 += val * y7[i];;
  }

  shared_mem_0[threadIdx.x] = dot_0;
  shared_mem_1[threadIdx.x] = dot_1;
  shared_mem_2[threadIdx.x] = dot_2;
  shared_mem_3[threadIdx.x] = dot_3;
  shared_mem_4[threadIdx.x] = dot_4;
  shared_mem_5[threadIdx.x] = dot_5;
  shared_mem_6[threadIdx.x] = dot_6;
  shared_mem_7[threadIdx.x] = dot_7;

  for (int k = blockDim.x / 2; k > 0; k /= 2) {
    __syncthreads();
    if (threadIdx.x < k) {
      shared_mem_0[threadIdx.x] += shared_mem_0[threadIdx.x + k];
      shared_mem_1[threadIdx.x] += shared_mem_1[threadIdx.x + k];
      shared_mem_2[threadIdx.x] += shared_mem_2[threadIdx.x + k];
      shared_mem_3[threadIdx.x] += shared_mem_3[threadIdx.x + k];
      shared_mem_4[threadIdx.x] += shared_mem_4[threadIdx.x + k];
      shared_mem_5[threadIdx.x] += shared_mem_5[threadIdx.x + k];
      shared_mem_6[threadIdx.x] += shared_mem_6[threadIdx.x + k];
      shared_mem_7[threadIdx.x] += shared_mem_7[threadIdx.x + k];
    }
  }

  if (threadIdx.x == 0){
        atomicAdd(result+0, shared_mem_0[0]);
        atomicAdd(result+1, shared_mem_1[0]);
        atomicAdd(result+2, shared_mem_2[0]);
        atomicAdd(result+3, shared_mem_3[0]);
        atomicAdd(result+4, shared_mem_4[0]);
        atomicAdd(result+5, shared_mem_5[0]);
        atomicAdd(result+6, shared_mem_6[0]);
        atomicAdd(result+7, shared_mem_7[0]);
  }
}

int main(void)
```

```
{

    Timer timer;
    const size_t N = 100000;
    const size_t K = 16;


    //
    // allocate host memory:
    //
    std::cout << "Allocating host arrays..." << std::endl;
    double  *x = (double*)malloc(sizeof(double) * N);
    double **y = (double**)malloc(sizeof(double*) * K);
    for (size_t i=0; i<K; ++i) {
      y[i] = (double*)malloc(sizeof(double) * N);
    }
    double *results  = (double*)malloc(sizeof(double) * K);
    double *results2 = (double*)malloc(sizeof(double) * 8);


    //
    // allocate device memory
    //
    std::cout << "Allocating CUDA arrays..." << std::endl;
    double *cuda_x; cudaMalloc( (void **)(&cuda_x), sizeof(double)*N);
    double *cuda_y0; cudaMalloc( (void **)(&cuda_y0), sizeof(double)*N);
    double *cuda_y1; cudaMalloc( (void **)(&cuda_y1), sizeof(double)*N);
    double *cuda_y2; cudaMalloc( (void **)(&cuda_y2), sizeof(double)*N);
    double *cuda_y3; cudaMalloc( (void **)(&cuda_y3), sizeof(double)*N);
    double *cuda_y4; cudaMalloc( (void **)(&cuda_y4), sizeof(double)*N);
    double *cuda_y5; cudaMalloc( (void **)(&cuda_y5), sizeof(double)*N);
    double *cuda_y6; cudaMalloc( (void **)(&cuda_y6), sizeof(double)*N);
    double *cuda_y7; cudaMalloc( (void **)(&cuda_y7), sizeof(double)*N);
    double *cuda_results2; cudaMalloc( (void **)(&cuda_results2), sizeof(double)*8);



    //
    // fill host arrays with values
    //
    for (size_t j=0; j<N; ++j) {
      x[j] = 1 + j%K;
    }
    for (size_t i=0; i<K; ++i) {
      for (size_t j=0; j<N; ++j) {
        y[i][j] = 1 + rand() / (1.1 * RAND_MAX);
      }
    }

    //
    // Reference calculation on CPU:
    //
    for (size_t i=0; i<K; ++i) {
      results[i] = 0;
      results2[i] = 0;
      for (size_t j=0; j<N; ++j) {
        results[i] += x[j] * y[i][j];
```

```
      }
    }

    //
    // Copy data to GPU
    //
    std::cout << "Copying data to GPU..." << std::endl;
    cudaMemcpy(cuda_x, x, sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y0, y[0], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y1, y[1], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y2, y[2], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y3, y[3], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y4, y[4], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y5, y[5], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y6, y[6], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_y7, y[7], sizeof(double)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);


    double *resultslarge  = (double*)malloc(sizeof(double) * K);

    std::vector<double> timings;
    timer.reset();

    for(int reps=0; reps < 10; ++reps)
    {

      std::cout << "Running dot products simultaneously..." << std::endl;
      for (size_t i=0; i<K/8; ++i) {
        cudaDeviceSynchronize();
        cudaMemcpy(cuda_y0, y[i*8+0], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y1, y[i*8+1], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y2, y[i*8+2], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y3, y[i*8+3], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y4, y[i*8+4], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y5, y[i*8+5], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y6, y[i*8+6], sizeof(double)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(cuda_y7, y[i*8+7], sizeof(double)*N, cudaMemcpyHostToDevice);
        cuda_many_dot_product<<<512, 512>>>(N, cuda_x, cuda_y0, cuda_y1, cuda_y2,
        cuda_y3, cuda_y4, cuda_y5, cuda_y6,
        cuda_y7, cuda_results2);
        cudaMemcpy(resultslarge+i*8, cuda_results2, sizeof(double)*8,
        cudaMemcpyDeviceToHost);
        for (int j = 0; j < 8; j++) {
          results2[j] = 0;
        }
        cudaMemcpy(cuda_results2, results2, sizeof(double)*8, cudaMemcpyHostToDevice);
      }

      //
      // Compare results
      //
      std::cout << "Copying results back to host..." << std::endl;
      for (size_t i=0; i<K; ++i) {
        std::cout << results[i] << " on CPU, " << resultslarge[i] << " on GPU.
        Relative difference: " << fabs(results[i] - resultslarge[i]) / results[i]
        << std::endl;
```

```
    }

    timings.push_back(timer.get());
  }

  std::sort(timings.begin(), timings.end());
  double time_elapsed = timings[10/2];


  printf("Dot product took %g seconds", time_elapsed);


  //
  // Clean up:
  //
  std::cout << std::endl << std::endl << "Cleaning up..." << std::endl;
  free(x);
  cudaFree(cuda_x);
  cudaFree(cuda_y0);
  cudaFree(cuda_y1);
  cudaFree(cuda_y2);
  cudaFree(cuda_y3);
  cudaFree(cuda_y4);
  cudaFree(cuda_y5);
  cudaFree(cuda_y6);
  cudaFree(cuda_y7);

  for (size_t i=0; i<K; ++i) {
    free(y[i]);
  }
  free(y);


  free(results);
  free(results2);
  free(resultslarge);

  return 0;
}
```

## 3 Multiple Dot Products 4

So, how could one make the multiple dot product applicable to general values of K? I think the best approach would be to pass the vectors y to the kernel in form of a matrix and then let the kernel automatically deduce how many columns the matrix has and therefore how many dot products should be computed. The kernel's variables then wouldn't be replicated but dynamically allocated and indexed, eg. dot[0] instead of dot_0. So generally speaking, one would have to shift from static copies to dynamically allocated indexing of variables.