INSTITUTE OF MICROELECTRONICS

360.252 Computational Science on Many-Core Architectures

Author:
Peter Holzner, 01426733

Submission: November 3, 2020

Contents

1 Basic CUDA

1.1 Allocation

Allocation of memory is quite fast, although the first allocation of a program is much slower than the following allocations. The reason for this is likely some form of device/driver/runtime initialization that needs to take place first.

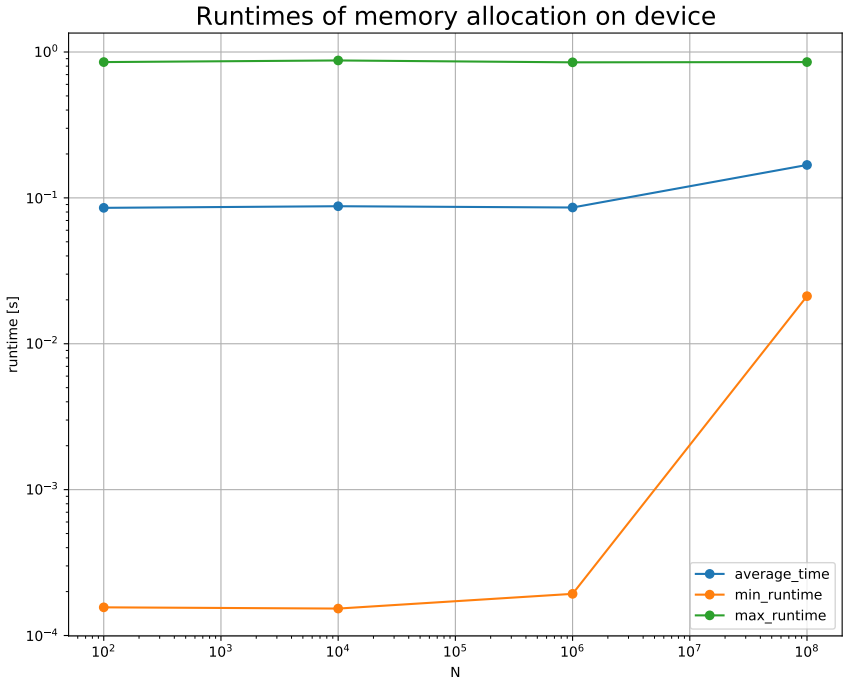

Fig. 1. Runtimes for allocation of a vector of different sizes on a CUDA device.

1.2 Initialization

Two vectors were initialized with the given schemes,

$$vec1 : 0, 1, ..., N-2, N-1 \tag{1}$$

$$vec2 : N-1, N-2, ..., 1, 0 \tag{2}$$

, and the runtimes of three different initialization implementations were recorded, see Fig.2. The estimated bandwidths are given in Fig.3. Note that the second version, memcpy called per element, did not work for vectors bigger than 10^3 elements. Initialization within a CUDA kernel was the fastest.

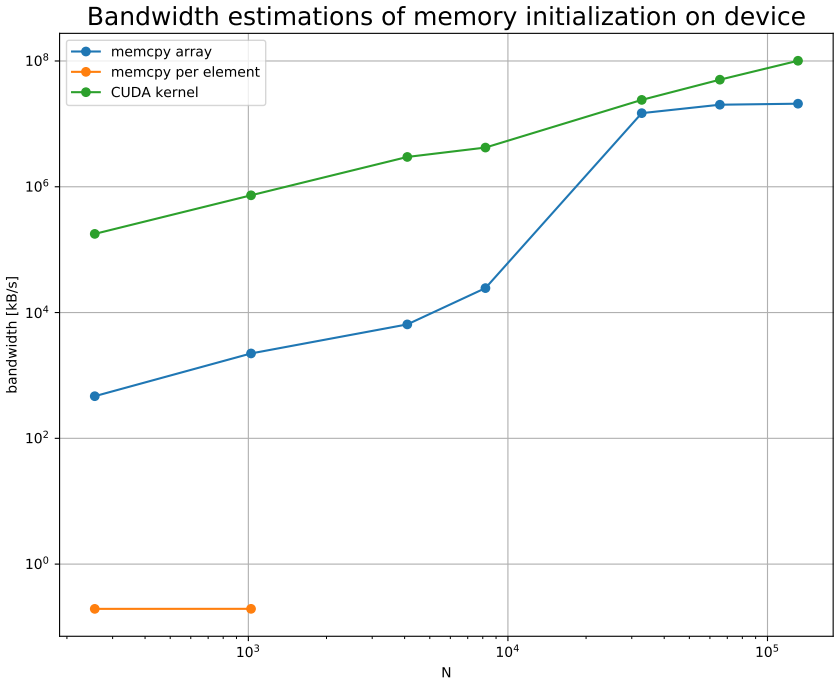

Fig. 3. Bandwidths for initialization of 2 vector according to the given scheme for different sizes.

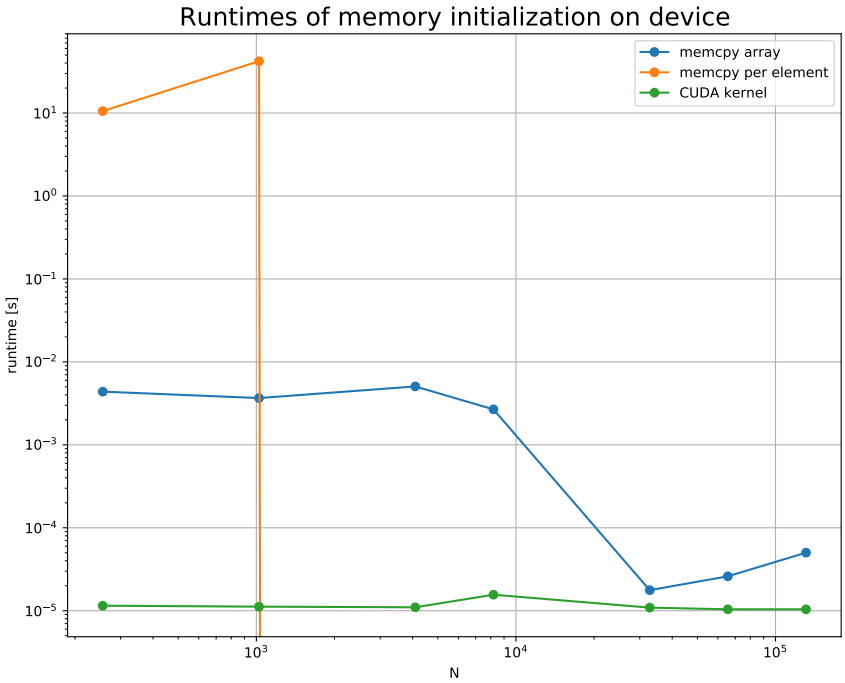

Fig. 2. Runtimes for initialization of 2 vector according to the given scheme for different sizes.

1.3 Vector addition

Surprisingly, the there isn't much of a difference between different block sizes. However, the algorithm scales with $\mathcal{O}(1)$ up until size $N = 10^7$ and only afterwards takes much longer to complete, as can be seen in Fig.4.

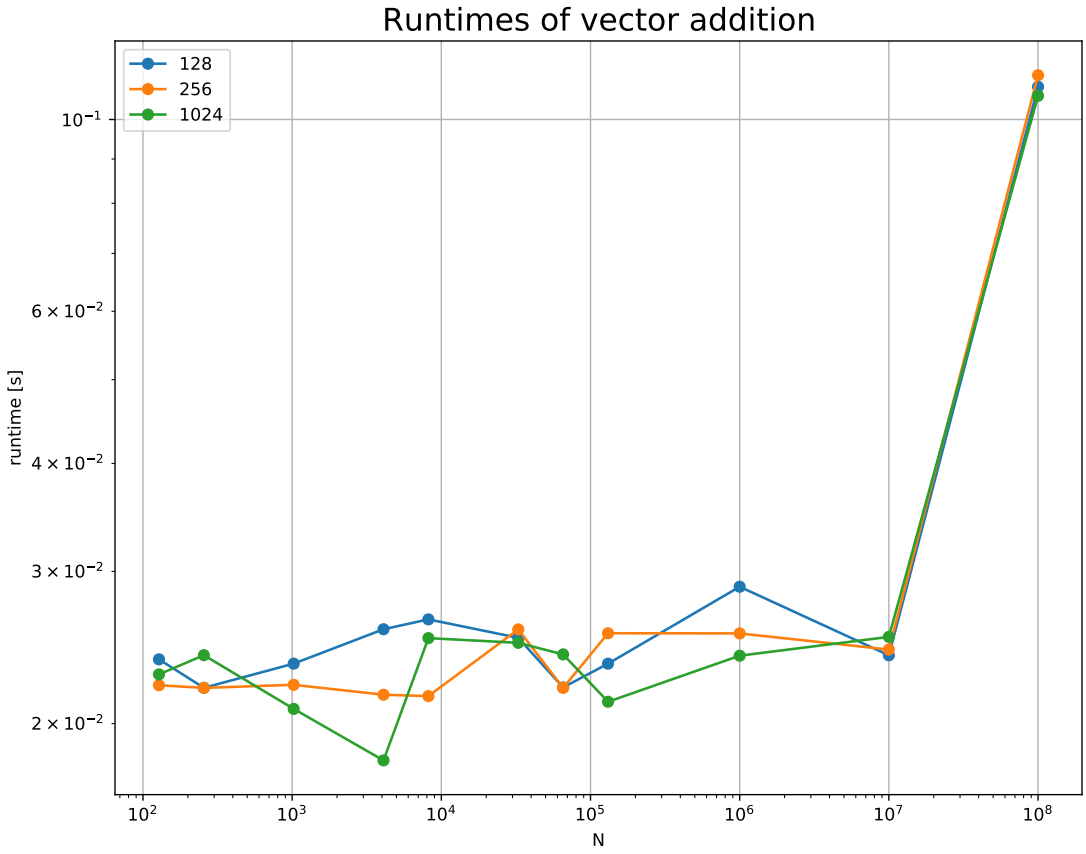

Fig. 4. Runtimes for addition of 2 vectors with kernels using different block sizes (see legend).

1.4 Vector Addition with different Block And Grid Sizes

This time, I compared runtimes for the same vector addition kernel using different combinations of grid and block sizes. Most combinations seem to work similarly well for a large ($N = 10^7$) vector, although combinations using (small, small), e.g. (16, 16), or (large, large), e.g. (1024, 1024), work notably worse.

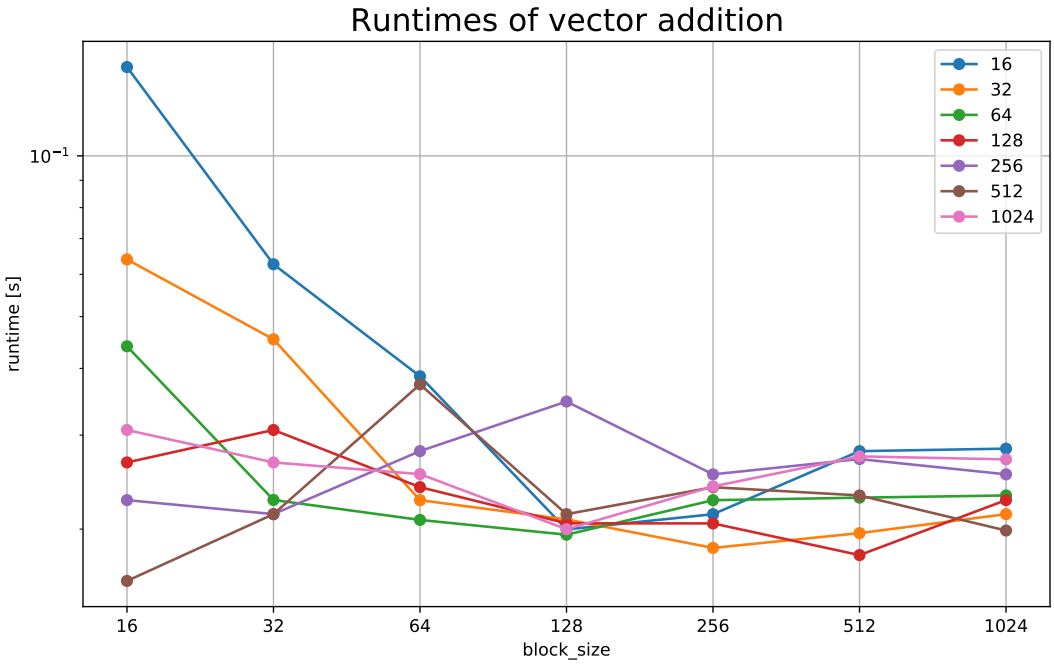

Fig. 5. Runtimes for different implementations of the dot product of two vectors. Block size and number of blocks used was 256.

All kernels used can be found at the end of the report. The source files are also provided in the submission package, as well as the data collected and the scripts used for plotting.

2 Dot Product

After some failed attempts of implementing these kernels, I looked for 'inspiration' online. I found resources here: `https://bitbucket.org/jsandham/algorithms_in_cuda/src/master/dot_product/`, that helped me to correct some mistakes I made. The final kernels are listed in the subsection Kernls below.

In Fig.6 and Fig.7, we can see the runtimes of the three different kernels/implementations. The block size and number of blocks (gridDim) were both set to the same number, 256 in the first and 1024 in the second picture. The results are nearly identical for both block sizes. The `atomic CUDA` and `CUDA + CPU` clearly outperform the two stage `2 CUDA kernels` version in terms of speed. However, the mixed `CUDA + CPU` version did not produce correct results for vectors larger than $N > 1e5$, hence they are not shown in the graph. Whether this issue happens due to a poor implementation, poor choice of values for the vectors - both input vectors were initialized to contain 1.0 in every element for easier checking of the correct result - or due to constraints of the benchmark machine, that I don't know. I heavily lean towards a poor implementation on my part, though I did not have enough time to investigate and solve the problem.

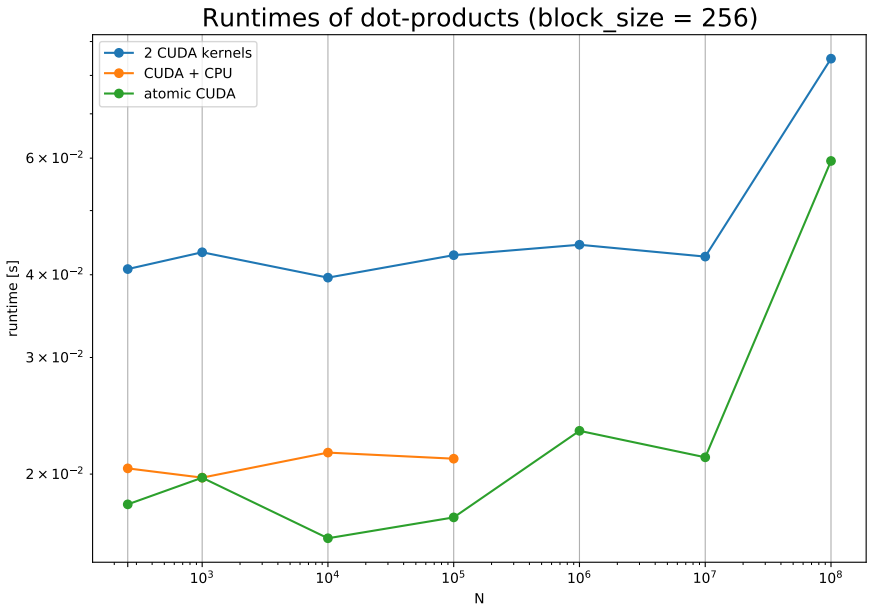

Fig. 6. Runtimes for different implementations of the dot product of two vectors. Block size and number of blocks used was 256.

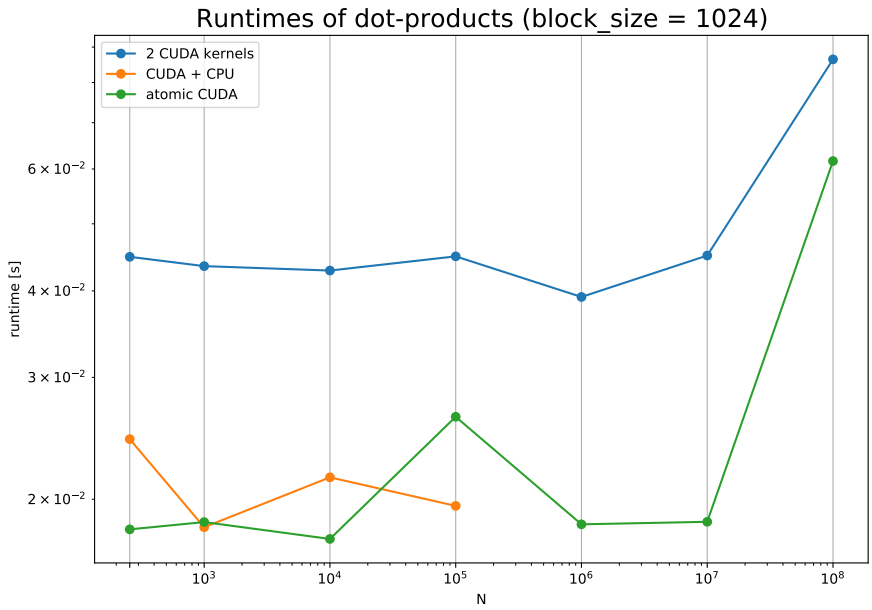

Fig. 7. Runtimes for different implementations of the dot product of two vectors. Block size and number of blocks used was 1024.

In general, the results are mostly as expected. Although I suspected that the `atomic CUDA` version would perform worse than the others, specifically the `2 CUDA kernel` version, for larger N - this was not the case, however. See Fig.**??** and Fig.**??** for reference.

I also suspected that the `CUDA + CPU` version might outperform the pure CUDA versions for larger N, since the resulting vector containing the sums of each block will be comparatively small. Summing over each element of the vector might then actually be faster to do in a purely serial way on the CPU without the overhead of threads. I was not able to confirm or deny my suspicion due to missing results, as explained above.

Sidenote: The block size was set via a `#define BLOCK_SIZE xxxx` at the start of the source file, because shared memory arrays can apparently only be reserved with constant size parameters (even using a `const int` did not work). There must be a better way to do this, but I didn't have enough time to research further.

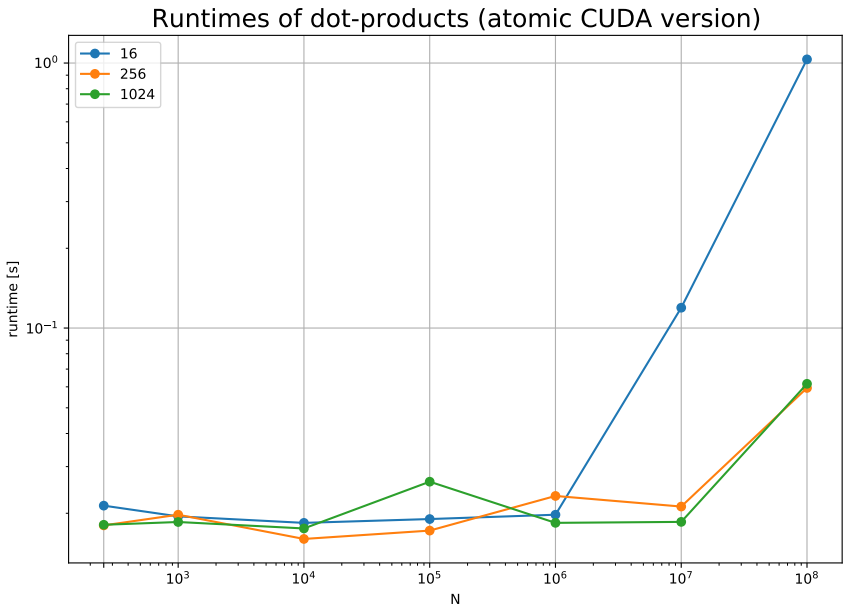

Fig. 8. Runtimes for the atomic CUDA version of the dot product with different block numbers and sizes.

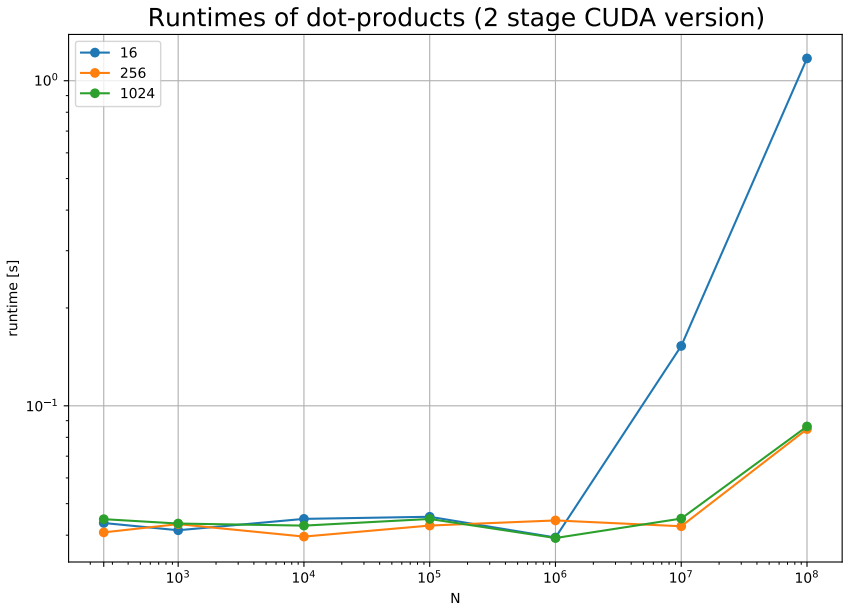

Fig. 9. Runtimes for the 2 stage CUDA version of the dot product with different block numbers and sizes.

3 CUDA Kernels

```
#include <iostream>
#include "timer.hpp"

//for (; p<=3; p++);
  //       cout << *p;

int main(void)
{
  int N = 1000;
  int num_tests = 10;
  int i = 0;

  Timer timer;
  double total_time = 0.0;
  double runtime = 0.0, max_runtime = 0.0, min_runtime = 100.0;

  for (; i<num_tests; i++) {
    double *d_x;
    timer.reset();

    cudaMalloc(&d_x, N*sizeof(double));
    cudaFree(d_x);
    cudaDeviceSynchronize();

    runtime = timer.get();
    std::cout << "(" << i+1 << ") Elapsed: " << runtime << std::endl;
    total_time += runtime;
    if (runtime > max_runtime) {
        max_runtime = runtime;
    }
    if (runtime < min_runtime) {
      min_runtime = runtime;
  }
    if (total_time > 1.) {
        break;
    }
  }
  std::cout << "num_tests;min_runtime;average_time;max_runtime" << std::endl;
  std::cout << num_tests << ";" << min_runtime << ";" << total_time/i << ";" << max_runtime
      << std::endl;

  return EXIT_SUCCESS;
}
```

Listing 1: "Ex2.1.a) Basic CUDA"

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"

__global__ void initKernel(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x + stride;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid - 1;
    arr2[tid] = N - 1 - tid;
  }
}

int main(void)
{
  std::vector<int> N_vec{ 256, 1024, 4096, 8192, 32768, 65536, 131072};

  int option = 1;
  int num_tests = 10;
```

```
int tests_done = num_tests;
std::string mode = "csv";

std::cout << "tests_done;N;total_time;average_time;max_runtime;min_runtime;hosttime" <<
    std::endl;
for (int & N: N_vec)
{
  //if (N != 256) break; // Break clause for testing purposes!
  //int N = 1024;
  int i = 0;

  Timer timer;
  double total_time = 0.0;
  double runtime = 0.0, max_runtime = 0.0, min_runtime = 100.0, hosttime = -1.;
  double *d_x, *d_y;
  std::vector<double> x(N, 0);
  std::vector<double> y(N, 0);

  timer.reset();
  for (i = 0; i < N; ++i)
  {
    x[i] = i;
    y[i] = N - 1 - i;
  }
  hosttime = timer.get();

  cudaMalloc(&d_x, N*sizeof(double));
  cudaMalloc(&d_y, N*sizeof(double));

  for (i = 0; i<num_tests; i++)
  {
    timer.reset();
    if (option == 1)
    {
      cudaMemcpy(d_x, x.data(), N*sizeof(double), cudaMemcpyHostToDevice);
      cudaMemcpy(d_y, y.data(), N*sizeof(double), cudaMemcpyHostToDevice);
      cudaDeviceSynchronize();
      tests_done = i+1;
    }
    else if (option == 2)
    {
      if (N < 1025)
      {
        double* px = &x[0];
        double* py = &y[0];
        for(int k=0; k < N; k++)
        {
          cudaMemcpy(d_x+k, px+k, sizeof(double), cudaMemcpyHostToDevice);
          cudaMemcpy(d_y+k, py+k, sizeof(double), cudaMemcpyHostToDevice);
        }
        cudaDeviceSynchronize();
        tests_done = i+1;
      }
    }
    else if (option == 3)
    {
      initKernel<<<(N+255)/256, 256>>>(d_x, d_y, N);
      tests_done = i+1;
    }
    else
    {
      std::cout << "No valid option selected!" << std::endl;
    }

    runtime = timer.get();
    //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
    total_time += runtime;

    cudaFree(d_x);
    cudaFree(d_y);
    cudaDeviceSynchronize();
```

```
      if (runtime > max_runtime)
      {
        max_runtime = runtime;
      }
      if (runtime < min_runtime)
      {
        min_runtime = runtime;
      }
      if (total_time > 1.)
      {
        break;
      }
    }

    if (mode == "readable")
    {
      std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
      std::cout << "Total runtime: " << total_time << std::endl;
      std::cout << "Average runtime; " << total_time/tests_done << std::endl;
      std::cout << "Maximum runtime: " << max_runtime << std::endl;
      std::cout << "Minimum runtime: " << min_runtime << std::endl;
      std::cout << "\nTime needed for vector init on host: " << hosttime << std::endl;
      std::cout << "\n\n";
      for (int j=0; j < 4; j++)
      {
        std::cout << j << ": " << x[j] << " | " << y[j] << std::endl;
      }
    }
    if (mode == "csv")
    {
      std::cout << tests_done << ";" << N << ";" << total_time << ";" << total_time/
          tests_done << ";" << max_runtime << ";" << min_runtime << ";" << hosttime << std::
          endl;
    }

  }
  return EXIT_SUCCESS;
}
```

Listing 2: Ex2.1.a) Basic CUDA

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"

__global__ void initKernel(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x + stride;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid - 1;
    arr2[tid] = N - 1 - tid;
  }
}

int main(void)
{
  std::vector<int> N_vec{ 256, 1024, 4096, 8192, 32768, 65536, 131072};

  int option = 1;
  int num_tests = 10;
  int tests_done = num_tests;
  std::string mode = "csv";

  std::cout << "tests_done;N;total_time;average_time;max_runtime;min_runtime;hosttime" <<
      std::endl;
  for (int & N: N_vec)
  {
```

```
//if (N != 256) break; // Break clause for testing purposes!
//int N = 1024;
int i = 0;

Timer timer;
double total_time = 0.0;
double runtime = 0.0, max_runtime = 0.0, min_runtime = 100.0, hosttime = -1.;
double *d_x, *d_y;
std::vector<double> x(N, 0);
std::vector<double> y(N, 0);

timer.reset();
for (i = 0; i < N; ++i)
{
  x[i] = i;
  y[i] = N - 1 - i;
}
hosttime = timer.get();

cudaMalloc(&d_x, N*sizeof(double));
cudaMalloc(&d_y, N*sizeof(double));

for (i = 0; i<num_tests; i++)
{
  timer.reset();
  if (option == 1)
  {
    cudaMemcpy(d_x, x.data(), N*sizeof(double), cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y.data(), N*sizeof(double), cudaMemcpyHostToDevice);
    cudaDeviceSynchronize();
    tests_done = i+1;
  }
  else if (option == 2)
  {
    if (N < 1025)
    {
      double* px = &x[0];
      double* py = &y[0];
      for(int k=0; k < N; k++)
      {
        cudaMemcpy(d_x+k, px+k, sizeof(double), cudaMemcpyHostToDevice);
        cudaMemcpy(d_y+k, py+k, sizeof(double), cudaMemcpyHostToDevice);
      }
      cudaDeviceSynchronize();
      tests_done = i+1;
    }
  }
  else if (option == 3)
  {
    initKernel<<<(N+255)/256, 256>>>(d_x, d_y, N);
    tests_done = i+1;
  }
  else
  {
    std::cout << "No valid option selected!" << std::endl;
  }

  runtime = timer.get();
  //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
  total_time += runtime;

  cudaFree(d_x);
  cudaFree(d_y);
  cudaDeviceSynchronize();


  if (runtime > max_runtime)
  {
    max_runtime = runtime;
  }
  if (runtime < min_runtime)
  {
```

```
        min_runtime = runtime;
      }
      if (total_time > 1.)
      {
        break;
      }
    }

    if (mode == "readable")
    {
      std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
      std::cout << "Total runtime: " << total_time << std::endl;
      std::cout << "Average runtime; " << total_time/tests_done << std::endl;
      std::cout << "Maximum runtime: " << max_runtime << std::endl;
      std::cout << "Minimum runtime: " << min_runtime << std::endl;
      std::cout << "\nTime needed for vector init on host: " << hosttime << std::endl;
      std::cout << "\n\n";
      for (int j=0; j < 4; j++)
      {
        std::cout << j << ": " << x[j] << " | " << y[j] << std::endl;
      }
    }
    if (mode == "csv")
    {
      std::cout << tests_done << ";" << N << ";" << total_time << ";" << total_time/
          tests_done << ";" << max_runtime << ";" << min_runtime << ";" << hosttime << std::
          endl;
    }

  }
  return EXIT_SUCCESS;
}
```

Listing 3: Ex2.1.b) Basic CUDA

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"

__global__ void initKernel2(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
  }
}

__global__ void initKernel3(double* arr, double* arr2, double* arr3, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
    arr3[tid] = 0;
  }
}

__global__ void addKernel(double* x, double* y, double* res, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx . x * blockDim . x;

  for(; tid < N; tid += stride)
  {
```

```
    res[tid] = x[tid] + y[tid];
    //res[tid] += 1;
  }
}

int main(void)
{

  double *d_x, *d_y, *d_res;

  std::vector<int> N_vec{ 128, 256, 1024, 4096, 8192, 32768, 65536, 131072, int(1e6), int(1
      e7), int(1e8)}};
  int num_tests = 10;
  int tests_done = num_tests;
  std::string mode = "readable";

  for (int & N: N_vec)
  {
    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_res, N*sizeof(double));
    cudaDeviceSynchronize();
    //if (N > 256) break; // Break clause for testing purposes!
    //int N = 1024;
    int i = 0;
    int block_size = 256;
    int blocks = (int)(N+255)/block_size;

    Timer timer;
    double total_time = 0.0;
    double runtime = 0.0, max_runtime = 0.0, min_runtime = 100.0;
    std::vector<double> x(N, 0);
    std::vector<double> y(N, 0);
    std::vector<double> res(N, 0);

    for (i = 0; i < N; ++i)
    {
      x[i] = i;
      y[i] = N - 1 - i;
    }

    //initKernel3<<<(N+255)/256, 256>>>(d_x, d_y, d_res, N);
    initKernel3<<<blocks, block_size>>>(d_x, d_y, d_res, N);
    cudaDeviceSynchronize();

    for (i = 0; i<num_tests; i++)
    {
      tests_done = i+1;
      timer.reset();

      //sumVectors<<<blocks, block_size>>>(d_x, d_y, d_res, N);
      addKernel<<<blocks, block_size>>>(d_x, d_y, d_res, N);
      cudaDeviceSynchronize();

      runtime = timer.get();
      //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
      total_time += runtime;

      if (runtime > max_runtime)
      {
        max_runtime = runtime;
      }
      if (runtime < min_runtime)
      {
        min_runtime = runtime;
      }
      if (total_time > 1.)
      {
        break;
      }
    }
```

```
    if (mode == "readable")
    {
      cudaMemcpy(x.data(), d_x, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaMemcpy(y.data(), d_y, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaMemcpy(res.data(), d_res, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
      std::cout << "Total runtime: " << total_time << std::endl;
      std::cout << "Average runtime; " << total_time/tests_done << std::endl;
      std::cout << "Maximum runtime: " << max_runtime << std::endl;
      std::cout << "Minimum runtime: " << min_runtime << std::endl;
      std::cout << "\n\n";
      for (int j=0; j < 4; j++)
      {
        std::cout << j << ": " << x[j] << " | " << y[j] << " | " << res[j] << std::endl;
      }
      for (int j=0; j < 4; j++)
      {
        int from_end = N - 1 - j;
        std::cout << from_end << ": " << x[from_end] << " | " << y[from_end] << " | " << res
            [from_end] << std::endl;
      }
      std::cout << "\n\n";
    }
    if (mode == "csv")
    {
      std::cout << tests_done << ";" << N << ";" << total_time << ";" << total_time/
          tests_done << ";" << max_runtime << ";" << min_runtime << std::endl;
    }

    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_res);
    cudaDeviceSynchronize();
  }

  return EXIT_SUCCESS;
}
```

Listing 4: Ex2.1.c)+d) Basic CUDA

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"

__global__ void initKernel2(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
  }
}

__global__ void initKernel3(double* arr, double* arr2, double* arr3, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
    arr3[tid] = 0;
  }
}

__global__ void addKernel(double* x, double* y, double* res, const size_t N)
```

```
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx . x * blockDim . x;

  for(; tid < N; tid += stride)
  {
    res[tid] = x[tid] + y[tid];
    //res[tid] += 1;
  }
}

int main(void)
{

  double *d_x, *d_y, *d_res;

  std::vector<int> block_size_vec{ 16, 32, 64, 128, 256, 512, 1024 };
  int num_tests = 10;
  int tests_done = num_tests;
  int N = (int)1e8;
  std::string mode = "csv";

  for (int & block_size: block_size_vec)
  {
    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_res, N*sizeof(double));
    cudaDeviceSynchronize();
    //if (N > 256) break; // Break clause for testing purposes!
    //int N = 1024;
    int i = 0;
    //int block_size = 256;
    //int blocks = (int)(N+block_size-1)/block_size;
    int blocks = (int)(N/block_size)+1;

    Timer timer;
    double total_time = 0.0;
    double runtime = 0.0, max_runtime = 0.0, min_runtime = 100.0;
    std::vector<double> x(N, 0);
    std::vector<double> y(N, 0);
    std::vector<double> res(N, 0);

    for (i = 0; i < N; ++i)
    {
      x[i] = i;
      y[i] = N - 1 - i;
    }

    //initKernel3<<<(N+255)/256, 256>>>(d_x, d_y, d_res, N);
    initKernel3<<<blocks, block_size>>>(d_x, d_y, d_res, N);
    cudaDeviceSynchronize();

    for (i = 0; i<num_tests; i++)
    {
      tests_done = i+1;
      timer.reset();

      addKernel<<<blocks, block_size>>>(d_x, d_y, d_res, N);
      cudaDeviceSynchronize();

      runtime = timer.get();
      //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
      total_time += runtime;

      if (runtime > max_runtime)
      {
        max_runtime = runtime;
      }
      if (runtime < min_runtime)
      {
        min_runtime = runtime;
      }
```

```
      if (total_time > 1.)
      {
        break;
      }
    }

    if (mode == "readable")
    {
      cudaMemcpy(x.data(), d_x, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaMemcpy(y.data(), d_y, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaMemcpy(res.data(), d_res, N*sizeof(double), cudaMemcpyDeviceToHost);
      cudaDeviceSynchronize();
      std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
      std::cout << "Total runtime: " << total_time << std::endl;
      std::cout << "Average runtime; " << total_time/tests_done << std::endl;
      std::cout << "Maximum runtime: " << max_runtime << std::endl;
      std::cout << "Minimum runtime: " << min_runtime << std::endl;
      std::cout << "\n\n";
      for (int j=0; j < 4; j++)
      {
        std::cout << j << ": " << x[j] << " | " << y[j] << " | " << res[j] << std::endl;
      }
      for (int j=0; j < 4; j++)
      {
        int from_end = N - 1 - j;
        std::cout << from_end << ": " << x[from_end] << " | " << y[from_end] << " | " << res
            [from_end] << std::endl;
      }
      std::cout << "\n\n";
    }
    if (mode == "csv")
    {
      std::cout << blocks << ";" << block_size << ";" << tests_done << ";" << N << ";" <<
          total_time << ";" << total_time/tests_done << ";" << max_runtime << ";" <<
          min_runtime << std::endl;
    }
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_res);
    cudaDeviceSynchronize();
  }

  return EXIT_SUCCESS;
}
```

Listing 5: Ex2.1.e) Basic CUDA

```
#include <iostream>
#include <string>
#include <vector>
#include "timer.hpp"
#include <cmath>

#define BLOCK_SIZE 256

__global__ void initKernel1(double* arr, const double value, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = value;
  }
}

__global__ void initKernel2(double* arr, double* arr2, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
```

```
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
  }
}

__global__ void initKernel3(double* arr, double* arr2, double* arr3, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    arr[tid] = tid;
    arr2[tid] = N - 1 - tid;
    arr3[tid] = 0;
  }
}

__global__ void addKernel(double* x, double* y, double* res, const size_t N)
{
  const int stride = blockDim.x * gridDim.x;
  int tid = threadIdx.x +  blockIdx.x * blockDim.x;

  for(; tid < N; tid += stride)
  {
    res[tid] = x[tid] + y[tid];
    //res[tid] += 1;
  }
}

__global__ void dot_A_1(double* x, double* y, double* block_sums, const size_t N)
{
  uint tid = threadIdx.x + blockDim.x* blockIdx.x;
  uint stride = blockDim.x* gridDim.x;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
  }
  cache[threadIdx.x] = tid_sum;

  __syncthreads();
  for (uint i = blockDim.x/2; i != 0; i /=2)
  {
    if (threadIdx.x < i) //lower half does smth, rest idles
    {
      cache[threadIdx.x] += cache[threadIdx.x + i]; //lower looks up by stride and sums up
    }
    __syncthreads();
  }

  if(threadIdx.x == 0) // cache[0] now contains block_sum
  {
    block_sums[blockIdx.x] = cache[0];
  }
  __syncthreads();
}

__global__ void dot_A_2(double* block_sums)
{
  int tid = threadIdx.x; // only one block, so this is fine!
  for (int i = blockDim.x / 2; i >= 1; i /=2) // same principal as above
  {
    if ( tid < i )
    {
      block_sums[tid] += block_sums[tid + i];
    }
    __syncthreads();
```

```
  }
}

__global__ void dot_Atomic(double* x, double* y, double* result, const size_t N)
{
  uint tid = threadIdx.x + blockDim.x* blockIdx.x ;
  uint stride = blockDim.x* gridDim.x ;

  __shared__ double cache[BLOCK_SIZE];

  double tid_sum = 0.0;
  for (; tid < N; tid += stride)
  {
    tid_sum += x[tid] * y[tid];
  }
  tid = threadIdx.x;
  cache[tid] = tid_sum;

  __syncthreads();
  for (uint i = blockDim.x/2; i != 0; i /=2)
  {
    __syncthreads();
    if (tid < i) //lower half does smth, rest idles
    {
      cache[tid] += cache[tid + i ]; //lower looks up by stride and sums up
    }
  }

  if(tid == 0) // cache[0] now contains block_sum
  {
    atomicAdd(result, cache[0]);
  }
}

int main(void)
{
  const int num_tests = 10;
  int tests_done = num_tests;
  const int block_size = BLOCK_SIZE;
  const int mode = 1;
  const int option = 2;

  std::cout << "N;blocks;block_size;tests_done;total_time;time_per_test;check" << std::endl;
  size_t N = 256;
  {
    //int blocks = (int)(N+block_size-1)/block_size;
    int blocks = block_size;
    double result = 0.0;
    double* presult = new double;
    presult = &result;
    double result_true = N;
    double* pnull = new double;
    *pnull = 0.0;
    double *d_x, *d_y, *d_block_sums, *d_result;
    double* h_block_sums;
    cudaMalloc(&d_result, sizeof(double));
    cudaMalloc(&d_x, N*sizeof(double));
    cudaMalloc(&d_y, N*sizeof(double));
    cudaMalloc(&d_block_sums, blocks*sizeof(double));
    cudaDeviceSynchronize();
    h_block_sums = new double[blocks];

    int i = 0;
    Timer timer;
    double total_time = 0.0;

    initKernel1<<<blocks, block_size>>>(d_block_sums, 0.0, block_size);
    initKernel1<<<blocks, block_size>>>(d_x, 1.0, N);
    initKernel1<<<blocks, block_size>>>(d_y, 1.0, N);
    //initKernel2<<<blocks, block_size>>>(d_x, d_y, N);
    cudaDeviceSynchronize();
```

```
timer.reset();
for (i = 0; i<num_tests; i++)
{
  tests_done = i+1;
  if (option == 1)
  {
    dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
    cudaDeviceSynchronize();
    dot_A_2<<<1, block_size>>>(d_block_sums);
    cudaDeviceSynchronize();
  }
  if (option == 2)
  {
    dot_A_1<<<blocks, block_size>>>(d_x, d_y, d_block_sums,  N);
    cudaDeviceSynchronize();
    cudaMemcpy(h_block_sums, d_block_sums, blocks*sizeof(double), cudaMemcpyDeviceToHost
        );
    //std::cout << h_block_sums[0] << " =? " << h_block_sums[blocks-1] << std::endl;
    cudaDeviceSynchronize();
    result = 0.0;
    for (int j = 0; j < blocks/2; j+=4)
    {
      result += h_block_sums[j] + h_block_sums[j+1] + h_block_sums[j+2] + h_block_sums[j
          +3];
    }
  }
  if (option == 3)
  {
    cudaMemcpy(d_result, pnull, sizeof(double), cudaMemcpyHostToDevice);
    dot_Atomic<<<blocks, block_size>>>(d_x, d_y, d_result, N);
  }

  //std::cout << "(" << i+1 << ") Elapsed: " << runtime << " s" << std::endl;
}
total_time = timer.get();
size_t check;
if (option == 1)
{
  cudaMemcpy(presult, d_block_sums, sizeof(double), cudaMemcpyDeviceToHost);
  check = result - result_true;
}
if (option == 2)
{
  check = result - result_true;
}
if (option == 3)
{
  cudaMemcpy(presult, d_result, sizeof(double), cudaMemcpyDeviceToHost);
  check = result - result_true;
}

if (mode == 0)
{
  std::cout << std::endl << "Results after " << tests_done << " tests:" << std::endl;
  std::cout << "Total runtime: " << total_time << std::endl;
  std::cout << "Average runtime; " << total_time/tests_done << std::endl;
  std::cout << "Check: " << result << " ?= " << result_true << std::endl;
  std::cout << "\n\n";
}
if (mode == 1)
{
  std::cout << N << ";"
  << blocks << ";"
  << block_size << ";"
  << tests_done << ";"
  << total_time << ";"
  << total_time/tests_done << ";"
  << check << std::endl;
}
cudaFree(d_x);
cudaFree(d_y);
cudaFree(d_block_sums);
```

```
    cudaFree(d_result);
    cudaDeviceSynchronize();
    free(presult);
    free(pnull);
    free(h_block_sums);
    if (N==256)
    {
      N=100;
    }
  }
  cudaDeviceSynchronize();
  return EXIT_SUCCESS;
}
```

Listing 6: Ex2.2. Dot Product