Computational Science on Many-Core Architectures: Exercise 9

Viktor Beck, 11713110

December 2022

Task 1: Libraries

Below we can see the comparison of the performances of the different libraries and the custom CUDA and OpenCL implementations. The time was measured such that only the computation of the dot product and the addition / subtraction of the vectors x and y is taken into account.

It seems that the custom CUDA implementation (from exercise 2) is the fastest but the amount of code written for such a small task is quite a lot in comparison to the few lines which were necessary for ViennaCL. The custom OpenCL implementation has the most lines of code and still performs second worst. In general one could say that the ViennaCL library has the best "effort to performance" ratio - at least in this case.

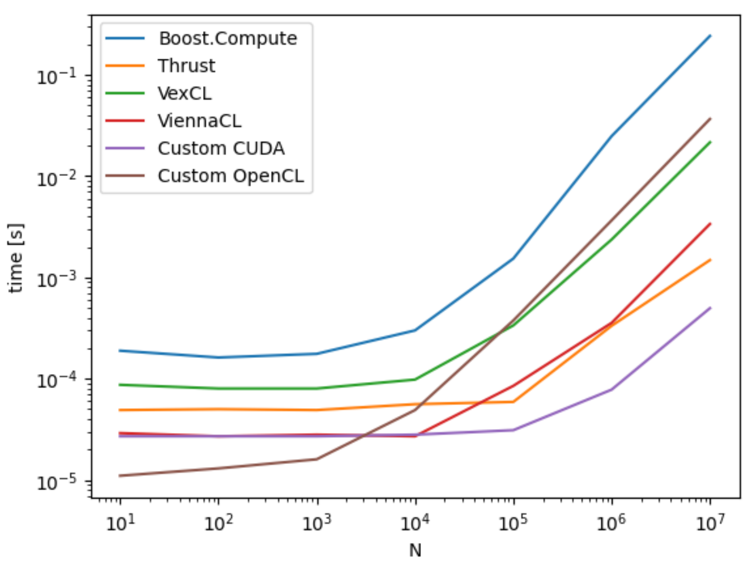

Figure 1: Comparing execution times for different libraries and custom implementations of a dot product for different vector sizes N on the RTX GPU

Task 2: HIP

Somehow, I was not able to install and use HIP therefore, and converted it manually as described in lecture 9 (with good old "ctrl+F"). Also, I could not test it... You can find the respective code snippet in the appendix.

Appendix

Task 1:

Boost.Compute

```

// specify use of OpenCL 1.2:
#define CL_TARGET_OPENCL_VERSION  120
#define CL_MINIMUM_OPENCL_VERSION 120

#include <vector>
#include <algorithm>
#include <iostream>

#include "timer.hpp"

#include <boost/compute/algorithm/transform.hpp>
#include <boost/compute/algorithm/inner_product.hpp>
#include <boost/compute/container/vector.hpp>
#include <boost/compute/functional/math.hpp>

namespace compute = boost::compute;

int main()
{

    // get default device and setup context
    compute::device device = compute::system::default_device();
    compute::context context(device);
    compute::command_queue queue(context, device);

    Timer timer;

    for (int k = 0; k <=7; k++){

        int N = pow(10, k);



        // generate data on the host
        std::vector<float> h_x(N);
        std::fill(h_x.begin(), h_x.end(), 1);
        std::vector<float> h_y(N);
        std::fill(h_y.begin(), h_y.end(), 2);

        // create a vector on the device
        compute::vector<float> d_x (N, context);
        compute::vector<float> d_y (N, context);

        // transfer data from the host to the device
        compute::copy(h_x.begin(), h_x.end(), d_x.begin(), queue);
        compute::copy(h_y.begin(), h_y.end(), d_y.begin(), queue);
```

```cpp

            timer.reset();

            // x+y
            compute::vector<float> x_plus_y(N, context);

            compute::transform(
                d_x.begin(),
                d_x.end(),
                d_y.begin(),
                x_plus_y.begin(),
                compute::plus<float>(),
                queue
            );

            // x-y
            compute::vector<float> x_minus_y(N, context);
            compute::transform(
                d_x.begin(),
                d_x.end(),
                d_y.begin(),
                x_minus_y.begin(),
                compute::minus<float>(),
                queue
            );

            float res = compute::inner_product(x_plus_y.begin(), x_plus_y.end(),
                x_minus_y.begin(), 0.f, queue);

            //std::cout << res << std::endl;

            std::cout << timer.get() << std::endl;
        }

        return 0;
    }
```

Thrust

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/transform.h>
#include <thrust/inner_product.h>
#include <cstdlib>

#include <vector>
#include "timer.hpp"

int main(void)
{
  Timer timer;

  for (int k = 0; k <= 7; k++)
  {
    int N = pow(10, k);

    // x and y on host
    thrust::host_vector<int> h_x(N);
    thrust::fill(h_x.begin(), h_x.end(), 1);
```

```
    thrust::host_vector<int> h_y(N);
    thrust::fill(h_y.begin(), h_y.end(), 2);

    // transfer data to the device
    thrust::device_vector<int> d_x = h_x;
    thrust::device_vector<int> d_y = h_y;

    timer.reset();

    // x+y
    thrust::device_vector<float> x_plus_y(N);

    thrust::transform(
        d_x.begin(),
        d_x.end(),
        d_y.begin(),
        x_plus_y.begin(),
        thrust::plus<float>());

    // x-y
    thrust::device_vector<float> x_minus_y(N);

    thrust::transform(
        d_x.begin(),
        d_x.end(),
        d_y.begin(),
        x_minus_y.begin(),
        thrust::minus<float>());

    float res = thrust::inner_product(x_plus_y.begin(), x_plus_y.end(), x_minus_y
        .begin(), 0.f);

    // std::cout << res << std::endl;

    std::cout << timer.get() << std::endl;
  }

  return 0;
}
```

VexCL

```

// The following three defines are necessary to pick the correct OpenCL version
    on the machine:
#define VEXCL_HAVE_OPENCL_HPP
#define CL_HPP_TARGET_OPENCL_VERSION 120
#define CL_HPP_MINIMUM_OPENCL_VERSION 120

#include <iostream>
#include <stdexcept>
#include <vexcl/vexcl.hpp>

#include "timer.hpp"
#include <vector>

int main()
{
    vex::Context ctx(vex::Filter::GPU && vex::Filter::DoublePrecision);

```

```
    std::cout << ctx << std::endl; // print list of selected devices

    Timer timer;

    for (int k = 0; k <= 7; k++)
    {

        int N = pow(10, k);

        std::vector<double> h_x(N, 1.0), h_y(N, 2.0);

        vex::vector<double> d_x(ctx, h_x);
        vex::vector<double> d_y(ctx, h_y);


        timer.reset();

        vex::vector<double> x_plus_y = d_x + d_y;
        vex::vector<double> x_minus_y = d_x - d_y;

        vex::Reductor<double, vex::SUM> sum(ctx);
        double res = sum(x_plus_y * x_minus_y);

        // std::cout << res << std::endl;

        std::cout << timer.get() << std::endl;
    }
    return 0;
}
```

ViennaCL

```

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

#define VIENNACL_WITH_CUDA

#include "viennacl/vector.hpp"
#include "viennacl/linalg/inner_prod.hpp"

int main()
{
  Timer timer;

  for (int k = 0; k <= 7; k++)
  {

    int N = pow(10, k);

    viennacl::vector<double> x = viennacl::scalar_vector<double>(N, 1.0);
    viennacl::vector<double> y = viennacl::scalar_vector<double>(N, 2.0);

    timer.reset();

    viennacl::vector<double> x_plus_y = x + y;
    viennacl::vector<double> x_minus_y = x - y;

    double res = viennacl::linalg::inner_prod(x_plus_y, x_minus_y);
```

```

    // std::cout << res << std::endl;

    std::cout << timer.get() << std::endl;
  }

  return EXIT_SUCCESS;
}
```

Task 2:

```
#include "poisson2d.hpp"
#include "timer.hpp"
#include <algorithm>
#include <iostream>
#include <stdio.h>

// y = A * x
__global__ void hip_csr_matvec_product(int N, int *csr_rowoffsets,
                                        int *csr_colindices, double *csr_values,
                                        double *x, double *y)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
      hipBlockDim_x * hipGridDim_x)
  {
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
    {
      sum += csr_values[k] * x[csr_colindices[k]];
    }
    y[i] = sum;
  }
}

// x <- x + alpha * y
__global__ void hip_vecadd(int N, double *x, double *y, double alpha)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
      hipBlockDim_x * hipGridDim_x)
    x[i] += alpha * y[i];
}

// x <- y + alpha * x
__global__ void hip_vecadd2(int N, double *x, double *y, double alpha)
{
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
      hipBlockDim_x * hipGridDim_x)
    x[i] = y[i] + alpha * x[i];
}

// result = (x, y)
__global__ void hip_dot_product(int N, double *x, double *y, double *result)
{
  __shared__ double shared_mem[512];

  double dot = 0;
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
      hipBlockDim_x * hipGridDim_x)
  {
    dot += x[i] * y[i];
```

```
  }

  shared_mem[hipThreadIdx_x] = dot;
  for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (hipThreadIdx_x < k)
    {
      shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
    }
  }

  if (hipThreadIdx_x == 0)
    atomicAdd(result, shared_mem[0]);
}

////////////// CG KERNEL 1: //////////////

__global__ void hip_cg_1(int N,
                          double alpha,
                          double beta,
                          double *x,
                          double *r,
                          double *p,
                          double *Ap,
                          double *result_rr)
{

  __shared__ double shared_mem[512];
  double dot = 0;

  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i +=
      hipBlockDim_x * hipGridDim_x)
  {
    // line 2,3,4: get x, r, p
    x[i] = x[i] + alpha * p[i];
    r[i] = r[i] - alpha * Ap[i];
    p[i] = r[i] + beta * p[i];

    // line 6: get dot(r,r)
    dot += r[i] * r[i];
  }

  __syncthreads();
  shared_mem[hipThreadIdx_x] = dot;

  for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (hipThreadIdx_x < k)
    {
      shared_mem[hipThreadIdx_x] += shared_mem[hipThreadIdx_x + k];
    }
  }

  if (hipThreadIdx_x == 0)
  {
    atomicAdd(result_rr, shared_mem[0]);
    // printf("%g", r[0]);
  }
}
```

```
////////////// CG KERNEL 2: //////////////
__global__ void hip_cg_2(int N,
                         int *csr_rowoffsets,
                         int *csr_colindices,
                         double *csr_values,
                         double *p,
                         double *Ap,
                         double *result1,
                         double *result2)
{

  __shared__ double shared_mem1[512];
  __shared__ double shared_mem2[512];

  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x * hipGridDim_x)
  {
    // line 5: get Ap
    double sum = 0;
    for (int k = csr_rowoffsets[i]; k < csr_rowoffsets[i + 1]; k++)
    {
      sum += csr_values[k] * p[csr_colindices[k]];
    }
    Ap[i] = sum;
  }

  // line 6: get dot(Ap,Ap) and dot(p,Ap)
  double dot1 = 0;
  double dot2 = 0;
  for (int i = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x; i < N; i += hipBlockDim_x * hipGridDim_x)
  {
    dot1 += Ap[i] * Ap[i];
    dot2 += p[i] * Ap[i];
  }

  shared_mem1[hipThreadIdx_x] = dot1;
  shared_mem2[hipThreadIdx_x] = dot2;
  for (int k = hipBlockDim_x / 2; k > 0; k /= 2)
  {
    __syncthreads();
    if (hipThreadIdx_x < k)
    {
      shared_mem1[hipThreadIdx_x] += shared_mem1[hipThreadIdx_x + k];
      shared_mem2[hipThreadIdx_x] += shared_mem2[hipThreadIdx_x + k];
    }
  }

  if (hipThreadIdx_x == 0)
  {
    atomicAdd(result1, shared_mem1[0]);
    atomicAdd(result2, shared_mem2[0]);
  }
}

/** Implementation of the conjugate gradient algorithm.
 *
 *  The control flow is handled by the CPU.
 *  Only the individual operations (vector updates, dot products, sparse
 * matrix-vector product) are transferred to hip kernels.
 *
 *  The temporary arrays p, r, and Ap need to be allocated on the GPU for use
```

```
 * with hip. Modify as you see fit.
 */
void conjugate_gradient(int N, // number of unknows
                        int *csr_rowoffsets, int *csr_colindices,
                        double *csr_values, double *rhs, double *solution)
//, double *init_guess)   // feel free to add a nonzero initial guess as needed
{
  // initialize timer
  Timer timer;

  // clear solution vector (it may contain garbage values):
  std::fill(solution, solution + N, 0);

  // initialize work vectors:
  double alpha, beta, residual_norm_squared, dot_pAp, dot_ApAp;
  double *hip_solution, *hip_p, *hip_r, *hip_Ap, *hip_scalar, *hip_dot_pAp, *hip_dot_ApAp;
  hipMalloc(&hip_p, sizeof(double) * N);
  hipMalloc(&hip_r, sizeof(double) * N);
  hipMalloc(&hip_Ap, sizeof(double) * N);
  hipMalloc(&hip_solution, sizeof(double) * N);
  hipMalloc(&hip_scalar, sizeof(double));

  hipMalloc(&hip_dot_ApAp, sizeof(double));
  hipMalloc(&hip_dot_pAp, sizeof(double));

  hipMemcpy(hip_p, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(hip_r, rhs, sizeof(double) * N, hipMemcpyHostToDevice);
  hipMemcpy(hip_solution, solution, sizeof(double) * N, hipMemcpyHostToDevice);

  // get residual_norm_squared
  const double zero = 0;
  hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
  hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_r, hip_r, hip_scalar);
  hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);

  double initial_residual_squared = residual_norm_squared;

  // line 1: get alpha0, beta0, Ap0
  hipLaunchKernelGGL(hip_csr_matvec_product, 512, 512, 0,0, N, csr_rowoffsets, csr_colindices, csr_values, hip_p, hip_Ap);

  hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
  hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_p, hip_Ap, hip_scalar);
  hipMemcpy(&dot_pAp, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);
  alpha = residual_norm_squared / dot_pAp;

  hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);
  hipLaunchKernelGGL(hip_dot_product, 512, 512, 0,0, N, hip_Ap, hip_Ap, hip_scalar);
  hipMemcpy(&dot_ApAp, hip_scalar, sizeof(double), hipMemcpyDeviceToHost);

  beta = (alpha * alpha * dot_ApAp - residual_norm_squared) / residual_norm_squared;



  int iters = 0;
  hipDeviceSynchronize();
```

```
timer.reset();
while (1)
{

  hipMemcpy(hip_scalar, &zero, sizeof(double), hipMemcpyHostToDevice);

  // std::cout << alpha << ", " << residual_norm_squared << std::endl;
  hipLaunchKernelGGL(hip_cg_1, 512, 512, 0,0, N, alpha, beta, hip_solution,
      hip_r, hip_p, hip_Ap, hip_scalar);

  hipMemcpy(&residual_norm_squared, hip_scalar, sizeof(double),
      hipMemcpyDeviceToHost);

  // std::cout << residual_norm_squared << std::endl;

  hipMemcpy(hip_dot_ApAp, &zero, sizeof(double), hipMemcpyHostToDevice);
  hipMemcpy(hip_dot_pAp, &zero, sizeof(double), hipMemcpyHostToDevice);

  // std::cout << dot_ApAp << ", " << dot_pAp << ", " << residual_norm_squared
      << std::endl;

  hipLaunchKernelGGL(hip_cg_2, 512, 512, 0,0, N, csr_rowoffsets, csr_colindices
      , csr_values,
                        hip_p, hip_Ap, hip_dot_ApAp, hip_dot_pAp);

  hipMemcpy(&dot_ApAp, hip_dot_ApAp, sizeof(double), hipMemcpyDeviceToHost);
  hipMemcpy(&dot_pAp, hip_dot_pAp, sizeof(double), hipMemcpyDeviceToHost);
  // std::cout << alpha << ", " << residual_norm_squared << std::endl;

  // line 7:
  alpha = residual_norm_squared / dot_pAp;

  // line 8:
  beta = (alpha * alpha * dot_ApAp - residual_norm_squared) /
      residual_norm_squared;

  // std::cout << dot_ApAp << ", " << dot_pAp << std::endl;

  // check for convergence
  if (std::sqrt(residual_norm_squared / initial_residual_squared) < 1e-6)
  {
    break;
  }

  if (iters > 10000)
    break; // solver didn't converge
  ++iters;
}
hipMemcpy(solution, hip_solution, sizeof(double) * N, hipMemcpyDeviceToHost);

hipDeviceSynchronize();
/*
std::cout << "Time elapsed: " << timer.get() << " (" << timer.get() / iters <<
    " per iteration)" << std::endl;

if (iters > 10000)
  std::cout << "Conjugate Gradient did NOT converge within 10000 iterations"
            << std::endl;
else
  std::cout << "Conjugate Gradient converged in " << iters << " iterations."
            << std::endl;
*/
```

```cpp

  std::cout << timer.get() / iters << "," << std::endl;

  hipFree(hip_p);
  hipFree(hip_r);
  hipFree(hip_Ap);
  hipFree(hip_solution);
  hipFree(hip_scalar);
}

/** Solve a system with `points_per_direction * points_per_direction` unknowns
 */
void solve_system(int points_per_direction)
{

  int N = points_per_direction *
          points_per_direction; // number of unknows to solve for

  // std::cout << "Solving Ax=b with " << N << " unknowns." << std::endl;

  //
  // Allocate CSR arrays.
  //
  // Note: Usually one does not know the number of nonzeros in the system matrix
  // a-priori.
  //       For this exercise, however, we know that there are at most 5 nonzeros
  //       per row in the system matrix, so we can allocate accordingly.
  //
  int *csr_rowoffsets = (int *)malloc(sizeof(double) * (N + 1));
  int *csr_colindices = (int *)malloc(sizeof(double) * 5 * N);
  double *csr_values = (double *)malloc(sizeof(double) * 5 * N);

  int *hip_csr_rowoffsets, *hip_csr_colindices;
  double *hip_csr_values;
  //
  // fill CSR matrix with values
  //
  generate_fdm_laplace(points_per_direction, csr_rowoffsets, csr_colindices,
                       csr_values);

  //
  // Allocate solution vector and right hand side:
  //
  double *solution = (double *)malloc(sizeof(double) * N);
  double *rhs = (double *)malloc(sizeof(double) * N);
  std::fill(rhs, rhs + N, 1);

  //
  // Allocate hip-arrays //
  //
  hipMalloc(&hip_csr_rowoffsets, sizeof(double) * (N + 1));
  hipMalloc(&hip_csr_colindices, sizeof(double) * 5 * N);
  hipMalloc(&hip_csr_values, sizeof(double) * 5 * N);
  hipMemcpy(hip_csr_rowoffsets, csr_rowoffsets, sizeof(double) * (N + 1),
      hipMemcpyHostToDevice);
  hipMemcpy(hip_csr_colindices, csr_colindices, sizeof(double) * 5 * N,
      hipMemcpyHostToDevice);
  hipMemcpy(hip_csr_values, csr_values, sizeof(double) * 5 * N,
      hipMemcpyHostToDevice);

  //
  // Call Conjugate Gradient implementation with GPU arrays
```

```
  //
  conjugate_gradient(N, hip_csr_rowoffsets, hip_csr_colindices, hip_csr_values,
      rhs, solution);

  //
  // Check for convergence:
  //
  /*
  double residual_norm = relative_residual(N, csr_rowoffsets, csr_colindices,
      csr_values, rhs, solution);
  std::cout << "Relative residual norm: " << residual_norm
            << " (should be smaller than 1e-6)" << std::endl;
  */
  hipFree(hip_csr_rowoffsets);
  hipFree(hip_csr_colindices);
  hipFree(hip_csr_values);
  free(solution);
  free(rhs);
  free(csr_rowoffsets);
  free(csr_colindices);
  free(csr_values);
}

int main()
{
  std::vector<int> N_vec = {10, 25, 50, 75, 100, 250, 500, 750, 1000};
  for (const auto &N : N_vec)
  {
    solve_system(N); // solves a system with 100*100 unknowns
  }

  return EXIT_SUCCESS;
}
```