High Performance Computer Architectures Practical Course - Exercise 4 -

Tutorium 1

David Jordan (6260776)
Florian Rüffer (7454628)
Michael Samjatin (7485765)

May 16, 2023

1 Matrix

In this task we had to use the Vc implementation to compute the element-wise square root of the matrix. We iterate over the elements of the matrix using nested loops. The outermost loop (variable k) controls the number of iterations (NIter). The second loop (variable l) iterates over the rows of the matrix, and the innermost loop (variable m) iterates over the columns of the matrix.

Within the innermost loop, the scalar float elements of the a and c?_simdVc arrays are converted to Vc float vector type (float_v) using the reinterpret_cast operation. The function f is then applied to the Vc float vector (aVec) using element-wise operations, and the result is stored in the Vc float vector cVec. Here is a part of the modified Matrix.cpp file:

```
1   /// Vc
2   TStopwatch timerVc;
3   // Iterate NIter times
4   for (int k = 0; k < NIter; k++){
5     // Iterate over all rows of the matrix
6     for(int l = 0; l < N; l++){
7       // Itertae over all columns of the matrix
8       for(int m = 0; m < N; m+= float_v::Size) {
9         // Using reinterpret_cast we convert scalar
              float array elements to  Vc float vector
              type
10        float_v &aVec = (reinterpret_cast<float_v&>(a[
              l][m]));
11        // Using reinterpret_cast we convert c_simdVc
              array elements to  Vc float vector type
12        float_v &cVec = (reinterpret_cast<float_v&>(
              c_simdVc[l][m]));
13        // Applying the square root template function
              to the Vc float  vector and storing it in
              cVec
14        cVec = f(aVec);
```

File 1: Matrix.cpp

The resulting output shows a marked speed up of 4.89 resulting from the use of the Vc implementation.

```
File  Edit  View  Terminal  Tabs  Help
hpc@hpc-vm:~/Desktop/Exercise/Exercises/3_Vc/1_Matrix$ ls
Matrix.cpp  Matrix_solution.cpp
hpc@hpc-vm:~/Desktop/Exercise/Exercises/3_Vc/1_Matrix$ g++ Matrix.cpp -O3 -lVc; ./a.out
Time scalar:  105.272 ms
Time headers: 22.9981 ms, speed up 4.57743
Time Vc:      21.507 ms, speed up 4.89478
SIMD and scalar results are the same.
SIMD and scalar results are the same.
hpc@hpc-vm:~/Desktop/Exercise/Exercises/3_Vc/1_Matrix$
```

Figure 1: Output

2 Quadratic Equation

In this exercise we had to complete code solving a set of quadratic equations using both scalar operations and SIMD operations using the Vc library. In the scalar calculations section, a loop iterates over a set of quadratic equations, and for each equation, the code calculates the discriminant and the root using the quadratic formula. In the SIMD sections, there are two loops, one using Vc and operating on vectors of 'a', 'b', and 'c' coefficients. It copies the coefficients from the input data structure, performs the calculations using Vc vectors, and then copies the roots back to the input data structure. The second loop also utilizes Vc, but instead of copying the coefficients, it reinterprets the memory of the input data structure as Vc vectors directly. This avoids the need for explicit copying and improves performance. The calculations are performed using the SIMD operations, and the roots are stored back in the input data structure. Here is part of the modified code:

```
1  // SIMD calculations with Vc using dataSIMD3 as input
       (use cast here)
2   TStopwatch timerSIMD3;
3   for (int io = 0; io < NIterOut; io++)
4     for (int i = 0; i < NVectors; i++)
5     {
6       // Using reinterpret_cast we reinterpret the
            memory representation of the input data-
            structure as Vc vectors
7       float_v &aV = *(reinterpret_cast<float_v *>(
            dataSIMD3.data[i].a));
8       float_v &bV = *(reinterpret_cast<float_v *>(
            dataSIMD3.data[i].b));
9       float_v &cV = *(reinterpret_cast<float_v *>(
            dataSIMD3.data[i].c));
10      float_v &xV = *(reinterpret_cast<float_v *>(
            dataSIMD3.data[i].x));
11
12      // We calculate the discriminant and roots here
```

```
          using SIMD operations
13        const float_v det = bV * bV - 4 * aV * cV;
14        xV = (-bV + sqrt(det)) / (2 * aV);
15      }
16    timerSIMD3.Stop();
```

File 2: QuadraticEquation.cpp

The resulting output reveals that there is a sizeable speedup compared to the scalar calculations, demonstrating the effectiveness of the use of SIMD operations and the Vc library to efficiently solve multiple quadratic equations in parallel, taking advantage of vectorized instructions for improved performance.

```
-vectorize -g -msse -lVc; ./a.out
Time scalar:   74.6019 ms
Time Vc AOS:   66.6821 ms, speed up 1.11877
Time Vc SOA:   18.4002 ms, speed up 4.05441
Time Vc AOSOA: 44.409 ms, speed up 1.67988
SIMD 1 and scalar results are the same.
SIMD 2 and scalar results are the same.
SIMD 3 and scalar results are the same.
hpc@hpc-vm:~/Desktop/Exercise/Exercises/3_Vc/2_QuadraticEquation$
```

Figure 2: Output

3 Newton

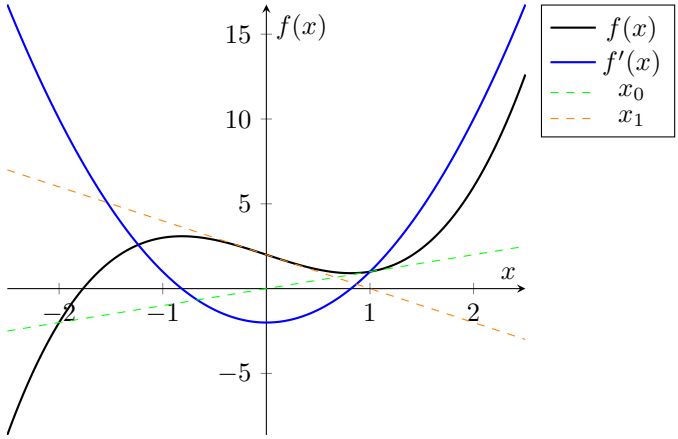
In this task we have to vectorize the Newton method. As mentioned in the task description the Newton Method is an interative process to approximate the x-axis intersection.

```
1     float_v FindRootVector(const float_v& p1, const
          float_v& p2)
2     {
3       float_v x = 1.f, x_new = 0.f;
4       float_m mask(true);
5       for( ; !mask.isEmpty(); ) {
6           // for(int i = 0; i < 1000; ++i){
7         x = x_new;
8         x_new(mask) = x - F(x,p1,p2) / Fd(x,p1,p2);
9         mask = abs((x_new - x)/x_new) > P;
10      }
11      return x_new;
12    }
```

File 3: Newton.cpp

The main idea of vectorizing the Newton method, is the calculation of multiple roots at once. This comes along with an issue, because if multiple roots are calculate, then also some of those will reach the desired precision earlier than others. To solve this problem we will use masks. The underlying principle of masks is a vector with each value mapped to a boolean. Each boolean value corresponds to the value of another vector. Using this methodology we can assign 'false' to a corresponding number, which already has our desired precision and therefore exclude it from further unneccessary calculations.

The first iteration will start with a mask of only 'true' values, as no value has yet reached the desired precision. This means every value will be updated accordingly with $x - F(x, p1, p2)/Fd(x, p1, p2)$. The next iteration will only update those values, which need further refinement ($\rightarrow$ mask value equals 'true') with $mask = abs((x_new - x)/x_new) > P$.



4 Random Access

The first part of this exercise is done by gathering the data from the input arry to the tmp float_v variable according to the index array. This is done by initializing a uint_v vector and it's corresponding indexes. If this is done, the goal of the subtask can be achieved by running the gather function:

```
1    tmp.gather(input, ind);
```

File 4: gathering data

Now we want to filter the data - from the tmp variable every value larger than 0.5. Therefore we create a specific mask with the following code: float_m mask = tmp > 0.5f;
With this mask, and a new initialized variable, where we want to safe the data, we can run the "gather" function again, with adapted parameters:

```
1    tmp2.gather(input, ind, mask);
```

File 5: gathering filtered data

The last part of this exercise is to put the data from the tmp float_v variable to the ouput array, when it satisfies a given condition. The condition is, that all values from tmp, that are smaller than 0.5, are put to the array "ouput" at the places given by indices.
We could create a new mask, but because this mask's type is identical to the mask above (and because the solutions does it like this), we reuse the mask (of course with adjusted filter < 0.5).
Lastly one can just use the scatter function with the added parameters to get the wished results to the tmp variable.

```
1    tmp.scatter(input, ind, mask);
```

File 6: scattering