2.3. Vector classes Library

Exercises are located at Exercises/3_Vc/
Solutions are located at Exercises/3_Vc/***/***_solution.cpp
To compile and run exercise programs use the line given in the head-comments in the code.
The results given here are obtained on Intel E7-4860 CPU with gcc4.7.3.

Vc Introduction

Vector classes (Vc) is a free software library to ease explicit vectorization of C++ code. It has an intuitive interface and provides portability between different compilers and compiler versions as well as portability between different vector instruction sets. Thus an application written with Vc can be compiled for AVX, SSE, XeonPhi (MIC) and others SIMD instructions[1].

Similar to the SIMD header files, it provides all basic arithmetic operations and functions, and much more in addition. All functionality, which the headers provide for **fvec** can be used similarly with the class **float_v**. The most useful functionalities of Vc are masks and random memory access.

The mask functionality allows conditioning calculations. For example, an absolute value of **a** can be calculated as:

float_m mask = (a < 0);   a(mask) = - a;

Here **mask** saves the comparison result for each entry of **a** in a form, which can be represented like [true;true;false;true] and **operator()** applies the assignment only to those entries, where the mask is true.

The random memory access functionality is provided by the **gather** and **scatter** functions. For example:

gather fills a vector **a** from an array **A** taking elements with indexes stored in a vector **I**:

a.gather(A, I); // A[I] -> a

scatter makes the opposite - fills an array entries from a vector:

a.scatter(A, I); // a -> A[I]

3_Vc/0_Matrix: description

The Matrix exercise requires to parallelize the square root extraction over a set of float variables arranged in a square matrix using the Vc library. The initial code implements scalar and vector parts using the SIMD header, the space for the Vc implementation is blank. Therefore the initial output shows 0 time for vector calculations and infinite speed up factor, that should be currently ignored.

[1] http://code.compeng.uni-frankfurt.de/projects/vc

Part of the source code of Matrix.cpp

```
float a[N][N] __attribute__ ((aligned(16)));
float c[N][N] __attribute__ ((aligned(16)));
float c_simd[N][N] __attribute__ ((aligned(16)));
float c_simdVc[N][N] __attribute__ ((aligned(16)));

template<typename T> // required calculations
T f(T x) {
  return sqrt(x);
}
...
int main() {

    // fill classes by random numbers
  for( int i = 0; i < N; i++ ) {
    for( int j = 0; j < N; j++ ) {
      a[i][j] = float(rand())/float(RAND_MAX); // put a random value, from 0 to 1
    }
  }

    /// -- CALCULATE --
    /// SCALAR
  TStopwatch timerScalar;
  for( int ii = 0; ii < NIter; ii++ )
    for( int i = 0; i < N; i++ ) {
      for( int j = 0; j < N; j++ ) {
        c[i][j] = f(a[i][j]);
      }
    }
  timerScalar.Stop();

    /// SIMD VECTORS
  TStopwatch timerSIMD;
  for( int ii = 0; ii < NIter; ii++ )
    for( int i = 0; i < N; i++ ) {
      for( int j = 0; j < N; j+=fvecLen ) {
          fvec &aVec = (reinterpret_cast<fvec&>(a[i][j]));
          fvec &cVec = (reinterpret_cast<fvec&>(c_simd[i][j]));
          cVec = f(aVec);
      }
    }
  timerSIMD.Stop();

   /// Vc
  TStopwatch timerVc;
  //TODO write the code using Vc
  timerVc.Stop();
```

```
Typical output
Time scalar:  798.745 ms
Time headers: 201.086 ms, speed up 3.97215
Time Vc:      0 ms, speed up inf
SIMD and scalar results are the same.
ERROR! SIMD and scalar results are not the same.
```

3_Vc/0_Matrix: solution

Since **float_v** is stored in memory in the same way as **fvec** and the function **f(…)** is a template, the part for Vc can be exactly the same, simply the name of the type must be changed:

```
Part of the source code of Matrix_solution.cpp
    /// Vc
  TStopwatch timerVc;
  for( int ii = 0; ii < NIter; ii++ )
    for( int i = 0; i < N; i++ ) {
      for( int j = 0; j < N; j+=float_v::Size ) {
          float_v &aVec = (reinterpret_cast<float_v&>(a[i][j]));
          float_v &cVec = (reinterpret_cast<float_v&>(c_simd[i][j]));
          cVec = f(aVec);
      }
    }
  timerVc.Stop();
```

```
Typical output after solution
Time scalar:  798.728 ms
Time headers: 201.078 ms, speed up 3.97223
Time Vc:      201.079 ms, speed up 3.97221
SIMD and scalar results are the same.
SIMD and scalar results are the same.
```

Since 4 float variables fit into a single SIMD vector, all calculations are done in parallel and no overhead operations are required, the expected speed-up factor should be 4.

3_Vc/1_QuadraticEquation: description

The QuadraticEquation exercise requires to vectorize using Vc the solution of a set of quadratic equations in three different ways, based on tree different data setups: (1) Array of Structures (AoS), (2) Structure of Arrays (SoA) (3) Array of Structures of Arrays (AoSoA).

It is recommended to compile the code with **-fno-tree-vectorize** option to prevent auto-vectorization of the scalar code, otherwise the comparison of the vectorized and scalar codes will not be direct.

The input data is given already in the required formats. An elemental structure **DataAOSElement**, which contains parameters of the equations and the resulting maximum root, is declared for the AoS format. **_mm_malloc** function is used to allocate aligned memory.

Part of the source code of QuadraticEquation.cpp

```
struct DataAOSElement {
  float a, b, c, // coefficients
    x; // a root
};

struct DataAOS {
  DataAOS(const int N) {
    data = (DataAOSElement*) _mm_malloc(sizeof(DataAOSElement)*N,16);
  }
  ~DataAOS() {
    if(data) _mm_free(data);
  }
  DataAOSElement *data;
};
```

The SoA format is declared as a whole structure, which contains dynamic arrays.

Part of the source code of QuadraticEquation.cpp

```
struct DataSOA {

  DataSOA(const int N) {
    a = (float*) _mm_malloc(sizeof(float)*N,16);
    b = (float*) _mm_malloc(sizeof(float)*N,16);
    c = (float*) _mm_malloc(sizeof(float)*N,16);
    x = (float*) _mm_malloc(sizeof(float)*N,16);
  }
  ~DataSOA()
  {
    if(a) _mm_free(a);
    if(b) _mm_free(b);
    if(c) _mm_free(c);
    if(x) _mm_free(x);
  }

  float *a, *b, *c, // coefficients
    *x; // a root
};
```

To define the AoSoA format an elemental structure **DataAOSOAElement** is declared to contain information about **float_v::Size** (4) equations, similarly to **DataSOA**. The elemental structures are packed together in the DataAOSOA structure similarly to **DataAOS**. Memory for all information is allocated in one go in **DataAOSOA** to ensure compact data allocation, afterwards the memory is distributed between elemental structures using **SetMemory** function.

Part of the source code of QuadraticEquation.cpp

```
struct DataAOSOAElement {

  void SetMemory(float *m) {
    a = m;
    b = m + float_v::Size;
    c = m + 2*float_v::Size;
    x = m + 3*float_v::Size;
  }

  float *a, *b, *c, // coefficients
    *x; // a root
};

struct DataAOSOA {
  DataAOSOA(const int N) {
    const int NVectors = N/float_v::Size;

    data = new DataAOSOAElement[NVectors];
    memory = (float*) _mm_malloc(sizeof(float)*4*N,16);

    float *memp = memory;
    for( int i = 0; i < NVectors; i++ ) {
      data[i].SetMemory(memp);
      memp += float_v::Size*4;
    }
  }
  ~DataAOSOA() {
    _mm_free(memory);
    delete[] data;
  }

  float *memory;
  DataAOSOAElement *data;
};
```

The given structures is filled by the same random data sample and scalar implementation is given in the exercise code.

Part of the source code of QuadraticEquation.cpp

```
  // fill parameters by random numbers
  for( int i = 0; i < N; i++ ) {
    float a = 0.01 + float(rand())/float(RAND_MAX); // put a random value, from 0.01 to 1.01 (a has not to be equal 0)
    float b = float(rand())/float(RAND_MAX);
    float c = -float(rand())/float(RAND_MAX);

    dataScalar.data[i].a = a;
    dataScalar.data[i].b = b;
    dataScalar.data[i].c = c;

    dataSIMD1.data[i].a = a;
    dataSIMD1.data[i].b = b;
    dataSIMD1.data[i].c = c;

    dataSIMD2.a[i] = a;
    dataSIMD2.b[i] = b;
    dataSIMD2.c[i] = c;

    const int nV = i/float_v::Size;
    const int iV = i%float_v::Size;
    dataSIMD3.data[nV].a[iV] = a;
    dataSIMD3.data[nV].b[iV] = b;
    dataSIMD3.data[nV].c[iV] = c;
  }

  /// -- CALCULATE --

  // scalar calculations
  TStopwatch timerScalar;
  for(int io=0; io<NIterOut; io++)
    for(int i=0; i<N; i++)
    {
      float &a = dataScalar.data[i].a;
      float &b = dataScalar.data[i].b;
      float &c = dataScalar.data[i].c;
      float &x = dataScalar.data[i].x;

      float det = b*b - 4*a*c;
      x = (-b+sqrt(det))/(2*a);
    }
  timerScalar.Stop();
```

Typical output

```
Time scalar:   567.055 ms
Time Vc AOS:   0 ms, speed up inf
Time Vc SOA:   0 ms, speed up inf
Time Vc AOSOA: 0 ms, speed up inf
ERROR! SIMD and scalar results are not the same. SIMD part 1.
ERROR! SIMD and scalar results are not the same. SIMD part 2.
ERROR! SIMD and scalar results are not the same. SIMD part 3.
```

3_Vc/1_QuadraticEquation: solution

(1) To vectorize the AoS format one needs to gather data from different instances of the **DataAOSElement** structure and pack it together (in groups by 4) in **float_v** variables. Since the same data (for example, the **a** variable) is placed in different parts of memory, data coping is necessary here. Ones data is grouped, the solution is found using exactly the same code as one given in the scalar part. The data is ungrouped back into the output **x** variable.

Part of the source code of QuadraticEquation_solution.cpp

```
for(int io=0; io<NIterOut; io++)
{
  for(int i=0; i<NVectors; i++)
  {
    // copy input data
    float_v aV;
    float_v bV;
    float_v cV;


    for(int iV=0; iV<float_v::Size; iV++)
    {
      aV[iV] = dataSIMD1.data[i*float_v::Size + iV].a;
      bV[iV] = dataSIMD1.data[i*float_v::Size + iV].b;
      cV[iV] = dataSIMD1.data[i*float_v::Size + iV].c;
    }

    const float_v det = bV*bV - 4*aV*cV;
    float_v xV = (-bV+sqrt(det))/(2*aV);

    // copy output data
    for(int iE=0; iE<float_v::Size; iE++)
      dataSIMD1.data[i*float_v::Size+iE].x = xV[iE];
  }
}
```

(2) The second task is vectorization with the SoA data format. Since in SoA similar data is placed near each other, the **reinterpret_cast** can be used. Ones the data is reinterpreted, the solution is found using exactly the same code, as the given one in the scalar part.

	Part of the source code of QuadraticEquation_solution.cpp

```
  for(int io=0; io<NIterOut; io++)
    for(int i=0; i<N; i+=float_v::Size)
    {
      float_v& aV = (reinterpret_cast<float_v&>(dataSIMD2.a[i]));
      float_v& bV = (reinterpret_cast<float_v&>(dataSIMD2.b[i]));
      float_v& cV = (reinterpret_cast<float_v&>(dataSIMD2.c[i]));
      float_v& xV = (reinterpret_cast<float_v&>(dataSIMD2.x[i]));

      const float_v det = bV*bV - 4*aV*cV;
      xV = (-bV+sqrt(det))/(2*aV);
    }
```

(3) The third task is vectorization with the AoSoA data format. The reinterpret cast can be used here in the same way as with the SoA format, just the dereference operator must be applied in addition. The calculations part remains the same as well.

	Part of the source code of QuadraticEquation_solution.cpp

```
  for(int io=0; io<NIterOut; io++)
    for(int i=0; i<NVectors; i++)
    {
      float_v& aV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].a));
      float_v& bV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].b));
      float_v& cV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].c));
      float_v& xV = *(reinterpret_cast<float_v*>(dataSIMD3.data[i].x));

      const float_v det = bV*bV - 4*aV*cV;
      xV = (-bV+sqrt(det))/(2*aV);
    }
```

	Typical output after solution

```
Time scalar:   566.821 ms
Time Vc AOS:   217.513 ms, speed up 2.60592
Time Vc SOA:   142.603 ms, speed up 3.97482
Time Vc AOSOA: 143.57 ms, speed up 3.94805
SIMD 1 and scalar results are the same.
SIMD 2 and scalar results are the same.
SIMD 3 and scalar results are the same.
```

The speed up factor of 4 is expected due to vectorization. It is achieved with the SoA and AoSoA data formats. With AoS additional data regrouping is required, that results in the smaller speed up of 2.6.

3_Vc/2_Newton: description

The Newton exercise requires to vectorize the Newton method for numerical solution of a group of equations.

The method can be explained graphically as in the Fig 1. The task is to find intersection of the curve and the X-axis. The algorithm starts with an initial approximation for root x_0 and takes a function value at

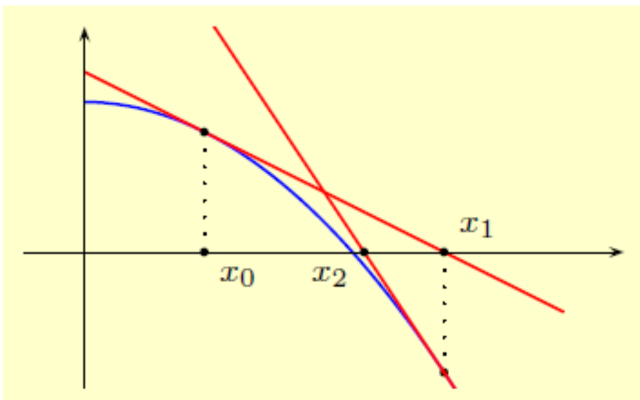

Fig. 1. Explanation of the Newton task.

this point. A tangent line is drown at this point, an intersection of the tangent line with the X-axis is the next approximation x_1:

$$x_k \equiv x_{k-1} - f(x_{k-1})/f'(x_{k-1})$$

Then the procedure is repeated until the approximation does not change within the required precision.

$$x_k - x_{k-1} < \text{epsilon}$$

A scalar version of the algorithm is given in the exercise. A solution is proposed to perform in two steps. First, vectorize the algorithm, which uses a fixed number of iterations (1000). Then make the number of iterations dependent on the required precision. The vectorized version must give exactly the same result as the scalar one.

Part of the source code of Newton.cpp

```
float FindRootScalar(const float& p1, const float& p2)
{
  float x = 1, x_new = 0;
  for( ; abs((x_new - x)/x_new) > P; ) {
      // for(int i = 0; i < 1000; ++i){
    x = x_new;
    x_new = x - F(x,p1,p2) / Fd(x,p1,p2);
  }
  return x_new;
```

Typical output

```
Scalar part:
Results are correct!
Time: 15.415 ms
SIMD part:
Results are NOT the same!
Time: 0.226021 ms
Speed up: 68.2015
```

3_Vc/2_Newton: solution

The parallelization is achieved by grouping 4 equations together. A complication in this case is that different equations will reach the required precision at different number of iterations. This problem can be solved using Vc masks: the loop exit condition should be stored as a mask. Then the loop is continued until the mask has at least one entry with the value **true**. A mask value should be also used during update of the root approximation in order to reproduce the scalar result.

Part of the source code of Newton_solution.cpp

```
float_v FindRootVector(const float_v& p1, const float_v& p2)
{
  float_v x = 1.f, x_new = 0.f;
  float_m mask(true);
  for( ; !mask.isEmpty(); ) {
      // for(int i = 0; i < 1000; ++i){
    x = x_new;
    x_new(mask) = x - F(x,p1,p2) / Fd(x,p1,p2);
    mask = abs((x_new - x)/x_new) > P;
  }
  return x_new;
}
```

Typical output after solution

```
Scalar part:
Results are correct!
Time: 15.3401 ms
SIMD part:
Results are the same!
Time: 4.49395 ms
Speed up: 3.4135
```

A typical speed up factor of vectorization is 4, but since mask is used, the parallelization is not full and at the final iterations only a part of the SIMD vector entries is used. Therefore, the speed up factor is about 3.5.

3_Vc/3_RandomAccess: description

The RandomAccess exercise requires to use different forms of the **gather** and **scatter** functions. For that an input array of **float** is provided and randomly filled. Also an array of random indices is provided. It is required to (1) gather data from the **input** array to the tmp **float_v** variable according to the **index** array; (2) similarly, gather the data, but only when it satisfies a given condition; (3) put the data from the tmp **float_v** variable to the **output** array, when it satisfies a given condition.

Part of the source code of RandomAccess.cpp

```
float input[N];
float output[N];

int main() {

  unsigned int index[float_v::Size];

  // fill input array with random numbers from 0 to 1
  for( int i = 0; i < N; i++ ) {
    input[i] = float(rand())/float(RAND_MAX);
  }

  // fill output array with 0
  for( int i = 0; i < N; i++ ) {
    output[i] = 0;
  }

  // fill indices with random numbers from 0 to N-1
  for( int i = 0; i < float_v::Size; i++) {
    index[i] = static_cast<unsigned int>(float(rand())/float(RAND_MAX)*N);
  }

  cout << "Indices: ";
  for(int i=0; i<float_v::Size; i++)
    cout << index[i] << " ";
  cout << endl;

    /// gather without masking
  float_v tmp;
  //TODO gather data with indices "index" from the array "input" into float_v tmp
  // Use void  gather (const float *array, const uint_v &indexes)
…
    /// gather with masking
  float_v tmp2;
  //TODO gather data with indices "index" from the array "input" into float_v tmp2, if the value of "input" large then 0.5
  // Use void  gather (const float *array, const uint_v &indexes, const float_m &mask)
…
  //TODO create mask for values for an obtained tmp values, which are large than 0.5 and
  //TODO put all values smaller than 0.5 from tmp to the array "output" at the places given by indices "index"
  // Use void scatter (float *array, const uint_v &indexes, const float_m &mask) const
```

Typical output

```
Indices: 25 96 76 42
Gather without masking: results are WRONG.
-4.67253e+33 is not equal to 0.988475
4.59163e-41 is not equal to 0.912037
0 is not equal to 0.80679
Gather with masking: results are WRONG.
Scatter with masking: results are correct.
```

3_Vc/3_RandomAccess: solution

(1) To gather data from the given places in an array, one needs to create a **uint_v** vector with corresponding indexes, then the **gather** function can be applied directly.

Part of the source code of RandomAccess_solution.cpp

```
  uint_v ind(index);
  tmp.gather(input,ind);
```

(2) To gather data under the given condition, one needs to create a mask with a corresponding type. Since the data is **float**, the **float_m** mask must be created, then the **gather** function with the mask parameter is applied directly. To ensure that the entries, which were masked out, have meaningful values, we need to initialise the output variable before gathering.

Part of the source code of RandomAccess_solution.cpp

```
  float_m mask = tmp > 0.5f;
  float_v tmp2(Vc::Zero);
  tmp2.gather(input, ind, mask);
```

(3) To scatter data under the given condition one needs to create a mask with a corresponding type. Since the data is **float**, the same mask can be used as in (2), then the **scatter** function with the mask parameter is applied directly.

Part of the source code of RandomAccess_solution.cpp

```
  mask = tmp < 0.5f;
  tmp.scatter(output, ind, mask);
```

Typical output after solution

```
Indices: 98 40 24 14
Gather without masking: results are correct.
Gather with masking: results are correct.
Scatter with masking: results are correct.
```