M. Troyer, P. Koumoutsakos
ETH Zürich, HIT G 31.8
CH-8093 Zürich

Set 2 - SIMD and BLAS

Issued: February 20, 2015
Hand in: March 2, 2015, 12:00pm

Question 1: SIMD force calculation

```
1  /// compute the Lennard-Jones force acting on one particle at position x0
2  scalar_type compute_force(std::vector<scalar_type> const& positions,
3                            scalar_type x0, scalar_type rc)
4  {
5      scalar_type rm2 = rm * rm;
6      scalar_type force = 0.;
7      for (size_type i=0; i<N; ++i) {
8          if (std::abs(r) < rc) {
9              scalar_type r2 = r * r;
10             scalar_type s2 = rm2 / r2;  // (rm/r)^2
11             scalar_type s6 = s2*s2*s2;  // (rm/r)^6
12             force += 12*eps * (s6*s6 - s6) / r;
13         }
14     }
15     return force;
16 }
```

Listing 1: `calculate_force` function included in `force1d.cpp`

The code above is a simple force calculation for a one-dimensional nbody problem. In this exercise you should have to speedup the `calculate_force` with manual SIMD.

a) Implement the loop with manual SSE intrinsics (or AVX if you have a suitable machine). As you cannot branch depending on the value of `r` when processing several values with one instruction, you need to avoid any `if` statements.

Hint: The comparison intrinsics like `_mm_cmplt_ps` create a mask depending on the outcome of the comparison. In a logic operation like `_mm_and_ps` this mask can be used to set all vector elements to zero for which the comparison yielded `false`.

We have to compute the absolute value of the distance `x0 - x[j]`, then we have to set the force to zero if this is greater than the cut-off. For the first part we set up a bit mask allowing us to unset the sign bits of the four `floats` packed into an SSE vector with a bitwise and operation. (An alternative would be to compute and compare the squares.) For the second part we compute the inverse distance as

```
__m128 rinv  = _mm_and_ps( _mm_cmplt_ps(absr,rc), _mm_rcp_ps(r) );
```

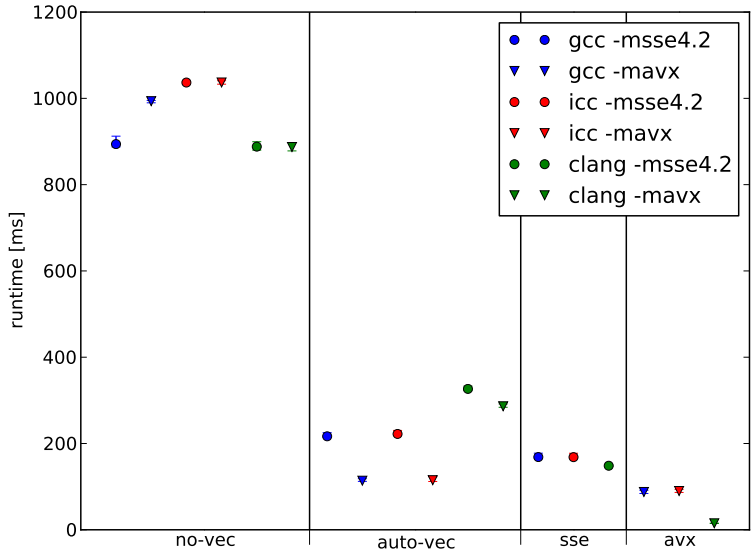

Figure 1: Runtime of the 1D force calculation compiled with different compilers. The left two panes are timings of the `force1d_vec` code without and with automatic vectorization. The right two panes are the manually vectorized codes with SSE and AVX intrinsics, respectively. Circles target SSE4.2 features only, triangles use AVX.

which will be zero if the comparison is not satisfied.

The rest of the loop consists of simple algebra operations only, vectorization is straightforward. Simple SSE and AVX implementations are provided with this solution. We don't assume the input vector length to be a multiple of the vector length and compute the remaining elements serially. Fig. 1 presents benchmark results with three different compilers: GCC 4.7.1, ICC 13.0.0 and Clang 3.2. For the former two vectorization is the default and needs to be disabled for the comparison. For Clang, the llvm vectorizer is still experimental and needs to be enabled explicitly. With the former two compilers vectorization gives speedups up to $5\times$ (SSE) and $10\times$ (AVX), exceeding the expected SIMD effects – most probably due to the fast `rcp` instruction. The experimental LLVM vectorizer only manages a factor of three. Our manual SSE and AVX codes beat the automatic vectorizers by 20 – 30% – except for Clang/LLVM surprisingly generating $6\times$ faster code from our `force1d_avx` implementation.

b) Discuss how the force calculation of the attached two-dimensional Molecular Dynamics code can be vectorized. Is a different data layout for the particle positions more suited to SIMD vectorization? Sketch how the periodic boundaries can be implemented with SIMD intrinsics, i.e. without branching.

Before, we had stored the particle positions as one vector of 2D positions. This allowed for a convenient iteration over particles, but now we would like to load several `x` coordinates in one SIMD vector alongside a second vector of corresponding `y` coordinates. Therefore we'd like to store the `x` coordinates of all particles in one continuous vector and the `y` coordinates in another one.

Our previous code calculated the distance considering periodic boundary conditions like this:

```
scalar_type r = x-y;
if     ( r < -extent/2 ) r += extent;
else if( r >  extent/2 ) r -= extent;
```

As with the cut-off radius we can get rid of the branches by and'ing the shifts with the outcome of the comparison:

```
// scalar_type r = x-y;
__m128 r = _mm_sub_ps(x,y);
// if     ( r < -l/2 ) r += l;
__m128 mask = _mm_cmplt_ps(r,minr);
__m128 shift = _mm_and_ps(mask,l);
r = _mm_add_ps(r,shift);
// else if( r >  l/2 ) r -= l;
mask = _mm_cmpgt_ps(r,maxr);
shift = _mm_and_ps(mask,l);
r = _mm_sub_ps(r,shift);
```

Question 2: BLAS

In this exercise you should familiarize yourself with general matrix multiplications (GEMM) using a BLAS library. We provide you with a skeleton code where you should replace the matrix multiplication with a call to GEMM. Finish the skeleton code and install a BLAS library on your personal system.

a) Report on what version of BLAS you are using, how you can link to it, if it is single-threaded or multi-threaded, and how can you specify the number of threads in the multi-threaded version. How much faster is your code with BLAS compared to the trivial implementation of the matrix multiplication in the skeleton code?

One can find many BLAS libraries on the market, basically every hardware vendor has its own implementation which is tuned for its own hardware. For example the vendor could optimize the block size to the available cache, or even use special proprietary instructions to improve performance.

Here are some common BLAS packages:

OpenBLAS `http://www.openblas.net`

Usual linking with: `c++ -lopenblas -lpthread main.cpp`.

Number of threads can be set with the `OPENBLAS_NUM_THREADS` environment variable.

Intel MKL `https://software.intel.com/en-us/intel-mkl`

For linking see the online guide `https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor`. For every installation one has different linking arguments.

Number of threads can be set with the `MKL_NUM_THREADS` environment variable.

Apple VecLib Available only on Apple devices, it can be included in your program with the `Accelerate` framework: `c++ -framework Accelerate main.cpp`

Number of threads can be set with the `VECLIB_MAXIMUM_THREADS` environment variable.

ATLAS `http://math-atlas.sourceforge.net`
Number of threads is fixed at compile time.

b) Same as a) except that the code has to run on EULER.

Summary

Summarize your answers, results and plots into a PDF document. Furthermore, elucidate the main structure of the code and report possible code details that are relevant in terms of accuracy or performance. Send the PDF document and source code to your assigned teaching assistant.