ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
S. Martin
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2021

Set 2 - Vectorization and IPL

Issued: October 15, 2021
Hand in (optional): October 29, 2020 12:00am

Question 1: Manual Vectorization of Reduction Operator (23 points)

23 points total

We are interested in optimizing the performance of the following compute kernel

$$r = \sum_{i=1}^{n} a_i, \tag{1}$$

where a_i are the elements of a vector $\boldsymbol{a} \in \mathbb{R}^n$ and $r \in \mathbb{R}$ is the result of reducing $\boldsymbol{a}$ by summing up its elements. We aim at improving the performance of this kernel by exploiting the data level parallelism (DLP) of $\boldsymbol{a}$ using the SIMD capabilities available on the CPU. We will utilize the streaming SIMD extensions (SSE) to *manually* vectorize the kernel shown in Equation (1). We want to test our implementation for single precision data (32bit) and double precision data (64bit).

a) Implement vectorized code for the reduction kernel in Equation (1). A baseline version `gold_red` has already been implemented and can be used as a reference. You are asked to work on the items marked with "`TODO`" in the source file `vectorized_reduction.cpp` located in the directory with the same name inside the skeleton code directory. A guide with a possible workflow can be found in the `Readme.html` file within the source directory. You may use your browser to open the file. Furthermore, the Intel intrinsics guide[1] is a useful reference for this task.

   The solution code for the 2-way and 4-way SSE reductions can be found in the file in the solution code directory (`vectorized_reduction.cpp`). Given that they are very similar, the code for one vectorized version can easily be adjusted to the other. Version A refers to the vectorized code you have written *first* (that is, the one you spent more time on) and the other version is referred to as B. If you have implemented the 2-way vectorization first, then substitute `A=d` for the intrinsics below. Otherwise substitute `A=s`. You first need to determine the number of SIMD lanes and store it in `simd_width`. Since we declare SSE and the number of SIMD lanes in the function name already, you could also just hard code the numbers 2 and 4. Next you need to initialize a vector register `sumN` that you will use

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

as the summation container for the reduction. Here, `N` corresponds to the number of SIMD lanes for version A or B. To broadcast a scalar value to all the SIMD lanes in the register you can use the `_mm_set1_pA` or `_mm_set_pA1` intrinsics. Note that both of them work with GCC and Intel compilers, while only `_mm_set1_pA` works with Clang. Next you loop over the array elements. You must be careful to select the correct stride when doing the loop counter increment which should be `i += simd_width`. Inside the loop body, you use the `_mm_add_pA` to add up the two registers you pass as arguments. One of them must be your summation container and the other you must *load* from memory. For this you must use `_mm_load_pA` to ensure efficient loads, since we work with aligned memory. The tricky part follows after the loop where you also must reduce the SIMD lanes to arrive at the scalar result. This can be done by storing back the vector into an array with number of elements equal to the number of SIMD lanes and then reducing them manually. SSE3 and later include horizontal add instructions which allow to sum up elements of a SIMD vector directly. Alternatively, you could use a `union` that shadows an array with vector register for the summation (see the comments for `variant 1` in the code).

- **2 points for correct computation/hard coding of `simd_width`**
- **2 points for correct initialization of `sumN` register using `_mm_set1_pA` or `_mm_set_pA1`**
- **2 points for correct stride on loop index `i += simd_width`**
- **2 points for correct use of `_mm_add_pA` in the loop body**
- **2 points for correct use of `_mm_load_pA` in the loop body. You must use *this* intrinsic here! -2 points for any other loads**
- **2 points for correct reduction of the remaining vector register after the loop.**
- **2 points for the adjustments in version B**
- **2 points for adding the `Makefile` flags**

b) You can measure the performance of your vectorized code by running 'make measurement' on Euler. The program will run a parallelized version of your kernel with different amount of threads to exploit TLP (Thread Level Parallelism). The job will create two `pdf` files, one for 32bit precision and another for 64bit precision results, each with speedup plots for a small n and another with a large n.

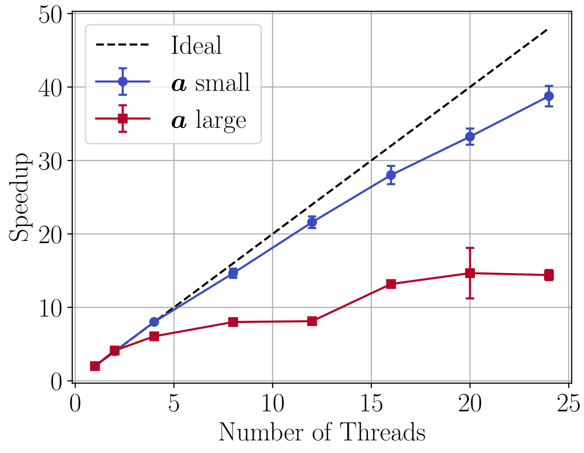
i) What is the maximum speedup you expect for 32bit precision and 64bit precision data with and without TLP?

From vectorizing the reduction kernel we can expect a speedup of $2\times$ for the 2-way vectorization and $4\times$ from the 4-way vectorization. Exploting TLP with p threads yields an expected speedup of $2p$ for the 2-way vectorization and $4p$ for the 4-way vectorization, respectively. Using one node on Euler with 24 cores would result in a speedup of $48\times$ and $96\times$ for 2-way and 4-way SIMD, respectively.

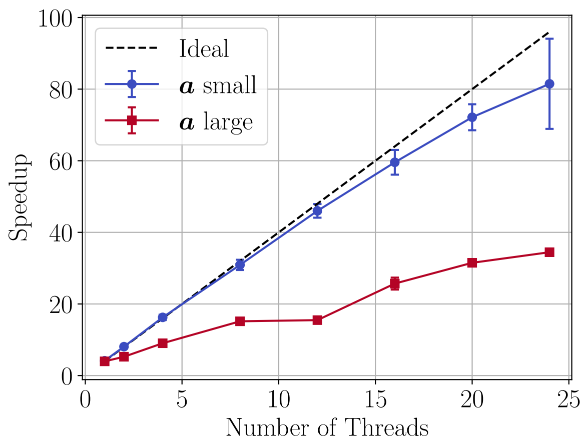
- **1 point for $2p$ speedup for 2-way SIMD**
- **1 point for $4p$ speedup for 4-way SIMD**

ii) Study the speedup plots in the generated `pdf` files and clearly explain the reason for differences you may observe for a small vector size n and a large vector size n (if there are any). Calculate the operational intensity of this kernel for single and double precision and include it in your argumentation.

Figure 1 and Figure 2 show the measured speedup for one Xeon E5-2680v3 node on Euler with 24 cores for an executable compiled with GCC 6.3.0 and Clang 4.0.1,

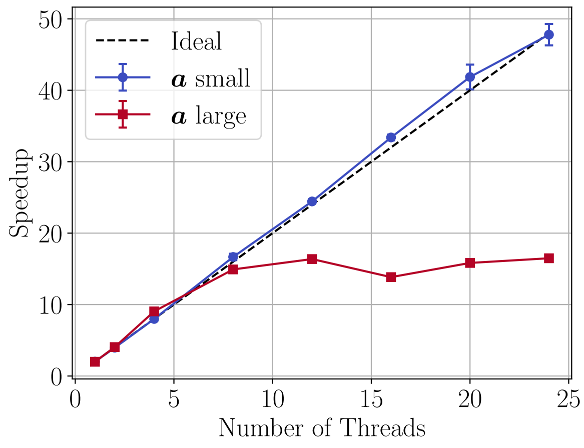


(a) 2-way SIMD

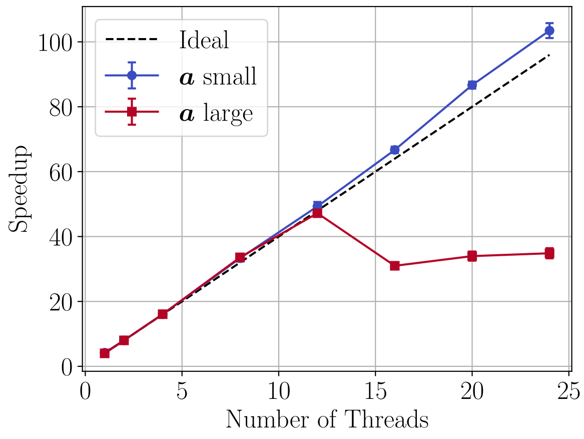


(b) 4-way SIMD

Figure 1: Xeon E5-2680v3 node on Euler (GCC 6.3.0)



(a) 2-way SIMD



(b) 4-way SIMD

Figure 2: Xeon E5-2680v3 node on Euler (Clang 4.0.1)

respectively, both with -O2 flags enabled. Small means that the per-thread workload of the array $\boldsymbol{a}$ *fits into the L2 cache*, while this is not the case for the large case. The operational intensity of this kernel is 1 flop per element (0.25 flop/byte for single precision and 0.125 flop/byte for double precision) and therefore clearly *memory bound*. We would expect the performance to be bound on what the memory bus can deliver. This is indeed what we observe for the case where $\boldsymbol{a}$ is large (the general case). We can get a decent speedup with a small number of threads because the pressure on the memory bus is still manageable. Increasing the number of threads further generates considerable amount of traffic on the memory bus and eventually causes a stall of performance because of the limitations in memory bandwidth.
The example also demonstrates nicely how much more we can gain in performance if we utilize cache memory optimally. This true for *any* kernel with a moderate to high operational intensity (which is not the case for the reduction kernel studied in this exercise). Because we perform a warm-up in our benchmark, the compulsory misses are eliminated for the small array case and most of the data accesses in the

loop are *cache hits*. In application code, a reduction kernel is usually called once at a time. In that case, the performance curve (blue line) would be similar to the one for a large array size (red line) because of compulsory cache misses. However, we learn from the blue line that if we have a kernel with higher operational intensity (that is, a sufficient amount of operations per byte loaded), if we design our data structures in a cache friendly way, we can squeeze substantially more performance out of the hardware. Also note that the Clang compiler generates superior machine code for this case than GCC. It is possible to reach performance above ideal on recent CPUs because of more advanced pipeline implementations than the classic 5-stage pipeline we have seen in the lecture. Such pipelines employ *out-of-order* execution techniques to allow execution of more than one instruction per clock cycle (IPC). This results in average cycles per instruction (CPI) below 1, which is not possible for the classic 5-stage pipeline we have studied in the lecture. The execution is called *superscalar* if the CPI is below 1.

- **2 points for identifying the cache influence for small sizes of a.**
- **2 points for correct operational intensity**
- **1 points for identifying/mentioning the memory bound property of the reduction kernel we study in this exercise.**

Question 2: Intel SPMD Program Compiler (ISPC) (42 points)

42 points total

Manual vectorization can be cumbersome and requires the use of macros to easily switch between different floating point precision. ISPC is a compiler that helps the programmer to avoid such difficulties. As a result, you will be able to write optimized code faster, often with satisfying up to excellent results. This depends on the complexity of the code you want to optimize and yourself. It is very easy to write vectorized code that performs *worse* than the baseline version (true for manual optimizations as well as with ISPC). In this exercise we will utilize the single program multiple data (SPMD) programming model used by ISPC to map program instances to the SIMD lanes on the hardware. We will study ISPC together with the general matrix multiplication kernel (GEMM)

$$C = \alpha AB + \beta C, \tag{2}$$

where $A \in \mathbb{R}^{p\times r}$, $B \in \mathbb{R}^{r\times q}$ and $C \in \mathbb{R}^{p\times q}$ are matrices. For this exercise, we consider the scalars $\alpha = 1$ and $\beta = 0$.

ISPC is used to optimize a performance critical kernel (usually defined in a function, for example) and then compile optimized machine code for that kernel only. In order to use it in our main application code, we need to *link* to the optimized code at compile time. The application code for this exercise is contained in the file `gemm.cpp` inside the `ispc_gemm` directory in the skeleton codes folder. We want to target SSE2 and AVX2 instruction sets, which are both supported by the CPUs on the euler nodes. You are asked to complete the items marked with "`TODO`" in the skeleton codes. Have a look at the `Readme.html` file inside the source directory for a suggestion of how to solve the exercise and further tips. The ISPC documentation[2] is a helpful resource for this task. Your code should compile for 32bit precision and 64bit precision data (single precision and double precision).

a) Start with a baseline implementation of the kernel in Equation (2) in the application code `gemm.cpp`. You can compile and test your code with

```
make debug=true gemm_serial
```

You may omit the debug flag if you do not need debugging symbols in your code. Your baseline GEMM implementation should return a norm of truth of 255.966 for double precision data and 256.211 for single precision data.

The baseline implementation can be found in the `gemm.cpp` application code in the ISPC solution code directory. We provide two implementations, one you should avoid and another with optimized memory access. The first implementation is the `gemm_serial` kernel. This kernel is bad because the access pattern into matrix B generates a lot of wasted bandwidth. Recall from the lecture that the hardware loads cache lines instead of single elements. Because we traverse along the row dimension in matrix B and C/C++ stores data in row-major layout, we effectively only use *one* element among the ones loaded with the cache line (note that there is a stride of q elements for every k-th element accessed). To avoid this wasteful access pattern, we can use a temporary work buffer which size should be large enough to fit into the L1 cache. It should not be too small, nor too large in order to exploit temporal locality optimally. For our optimized memory access version we use a temporary buffer size of $16\,\mathrm{kB}$ (for both, single precision and double precision data) which is half of

[2] `https://ispc.github.io/ispc.html`

the L1 cache size. A size of $32\,\mathrm{kB}$ is already too large as there is not enough space left for other temporaries. This memory optimization is implemented in the second version `gemm_serial_tile`, where the temporary storage is named `tile`. This consideration is also important for the later ISPC implementation because the vector registers can be used together with the temporary buffer efficiently. The small optimization already results in an improvement of $10\times$ compared to the naive `gemm_serial` implementation (you need to stream enough bytes to reach this number, i.e., your matrix must be large enough). When we start to vectorize, it is important to start with a baseline kernel that reflects the best implementation we can possibly achieve (or at least is somewhat close to it). If we do not, it is possible to measure unrealistic large speedups when we compare to the vectorized version of the kernel which leads to false interpretation of your performance assessment. Note that the $10\times$ difference for the two baseline kernels is a large improvement already!

- **6 points for a working GEMM implementation (norm of truth is correct)**
- **2 points if wasteful memory access pattern into matrix B is identified**
- **2 points for a memory access optimized version of the GEMM kernel (any *improvement* is fine).**

b) To get started with the ISPC compiler, you need to install it. You can install it with

```
make install_ispc_linux_x86_64
```

This works on Euler or other 64bit Linux distributions. See also the `Readme.html` file. Windows and MacOS binaries can be downloaded here[3]. All explanations given in this exercise were tested with the Linux binary of ISPC.

c) Complete the ISPC related `Makefile` flags and implement the ISPC code for the kernel of Equation (2) in the file `gemm.ispc`. We target optimized kernels for SSE2 and AVX2 instruction sets. You can compile and link to ISPC code with the following (default) target

```
make debug=true gemm
```

You may omit the debug flag if you do not need debugging symbols in your code. You can submit a job on Euler to test your code using

```
make job
```

For convenience, you may want to work with an interactive node on Euler to omit the latency associated with submitting jobs to the queue.

Before you can compile ISPC code you must specify what architecture you are going to target. For Euler, this is the 64bit x86 instruction set, `--arch=x86-64`. Because we want to compile an optimized kernel for the SSE2 and AVX2 extensions, we have to specify these flags for the individual `make` targets. For SSE2 we add `--target=sse2-i32x8` and for AVX2 `--target=avx2-i32x16`. The `i32` indicates that 32bit integers will be used for addressing and `x8` or `x16` is the *gang* size. You should experiment with different gang sizes as another configuration might run better for a different application. Note that the gang size is usually 2–$4\times$ the SIMD width of the specified target ISA. See `ispc --help` for other possible combinations.

The ISPC kernel can be found in the file `gemm.ispc` in the solution code directory. There are different ways we can attack the vectorization of the matrix-matrix product. The most

[3] https://ispc.github.io/downloads.html

important property we want to maintain for our vectorized code is to use *efficient vector loads and stores* and high utilization of the cache memory. For the former, this means that when we load the data into the registers, we want to load values that are right next to each other in memory. If this can not be ensured, ISPC must use (much) less efficient gather operations to load data at arbitrary addresses. This would be even worse for SSE2 because gathers are not available for this extension. If you already thought about improving the memory access pattern for the baseline kernel above (`gemm.cpp`) then you will have a good starting point for vectorization. Remember, the strided access into the elements of matrix B, as we did in our naive baseline implementation, would result in gather operations and completely destroy the data level parallelism (DLP) we try to exploit here. Furthermore, the `uniform` storage specifier would not work in this case. Our solution strategy is as follows:

1. We want to exploit DLP for the inner product of the matrix-matrix multiplication.
2. We utilize a temporary storage for the inner product that is large enough to fit into the L1 cache. We use the same temporary buffer layout as we did for the scalar implementation of the GEMM kernel.
3. We stream along the row dimension of matrix B, mapping SIMD lanes to column indices. Each SIMD lane will compute an individual inner product and use the `tile` buffer for temporal storage.

To implement this strategy with ISPC, we must think about where to create a SPMD execution flow in order to map the SIMD lanes to column indices in matrix B and stream along the row dimension. From the "view" of an individual SIMD lane, each of the lanes must perform the outer loops over the p and q dimensions, where the two loop indices must have a stride that corresponds to the dimension of the temporary work buffer (similar to blocking). When we process the inner dimension, we must first initialize the work buffer to zero. Each SIMD lane will use a part of this buffer so each lane must initialize its work space *individually*. Therefore, this is the first place in the code where we must tell ISPC to enter SPMD execution flow. We simply do this with a `foreach` loop construct. Now we continue by iterating over the inner dimension r, where this must be done again by all SIMD lanes together (no SPMD here). We store the current value of the A matrix in a *vector* register `Aik`, which has the *same* value for all SIMD lanes. Next we compute the inner product, where here we have to tell ISPC again that this operation must map to individual SIMD lanes (by using the `foreach` construct). Finally, we write back the result of the inner product into matrix C, again we must use SPMD execution flow here. After this operation has completed we move on to process the next batch by incrementing the `i` and `j` indices accordingly.

- **2 points for adding the `--arch=x86-64` flag to the Makefile**
- **2 points for adding a `--target` flag to the Makefile that will result in SSE2 code**
- **2 points for adding a `--target` flag to the Makefile that will result in AVX2 code**
- **4 points for completing the function signature of the ISPC kernel (you must use the `uniform` keyword here to tell ISPC that the data is coalesced in memory)**
- **6 points for an ISPC implementation that *compiles***

- 4 points for a correct ISPC implementation (the computed error is within *machine precision*)
- 4 points for an ISPC implementation that runs *faster* than the baseline kernel (any *improvement* is fine).

d) What are the speedups you expect for your SSE2 and AVX2 optimized kernels? Report two numbers for each optimization, one for single precision data and one for double precision data. If your ISPC code does not reach these expectations, please state the reason for this behavior.

We know that the SSE2 register width is 128bit. Therefore, we expect a speedup of $2\times$ for double precision data (64bit per element, 2-way SIMD) and $4\times$ for single precision data (32bit per element, 4-way SIMD). The AVX2 register is 256bit wide and twice as large as the SSE2 registers. Hence, we expect a speedup of $4\times$ for double precision data (4-way SIMD) and $8\times$ for single precision data (8-way SIMD).

We verify these numbers on a Xeon E5-2680v3 node on Euler:

```
GEMM ISPC SSE2:
Data type size:     4 byte
Number of elements: A=524288; B=1048576; C=524288
Norm of truth:      256.211
Error:              0
Speedup:            4.76702

GEMM ISPC AVX2:
Data type size:     4 byte
Number of elements: A=524288; B=1048576; C=524288
Norm of truth:      256.211
Error:              1.36433e-05
Speedup:            10.1323

GEMM ISPC SSE2:
Data type size:     8 byte
Number of elements: A=524288; B=1048576; C=524288
Norm of truth:      255.966
Error:              0
Speedup:            2.63282

GEMM ISPC AVX2:
Data type size:     8 byte
Number of elements: A=524288; B=1048576; C=524288
Norm of truth:      255.966
Error:              2.52435e-14
Speedup:            5.40656
```

If you see speedups that are much larger than these expectations, it is very likely that the baseline kernel you are comparing against is not optimized (in terms of memory accesses). Recall the $10\times$ speedup we obtained for the memory access optimization in the first part of the exercise. If we did not do this optimization, the speedup numbers shown here would have to be multiplied by a factor of 10. Note that the speedups shown above are larger than the expectation which can be attributed to the following:

- The baseline kernel we are comparing against is yet not optimal.
- Optimizations performed at -O2 level differ between the application code compiler (here GCC 6.3.0) and ISPC.
- The differences are larger for the AVX2 code than SSE2. This is because of fused-multiply-add (FMA) instructions that are available in the AVX2 instruction set but not in SSE2.

The AVX2 ISPC kernel is further benchmarked against a GEMM kernel implemented using the Eigen3[4] library. This kernel has been added in the file `gemm_eigen.cpp` in the solution code directory and can be compiled into the application code using the `eigen=true` option for the `make` command. The Eigen3 kernel is compiled with aggressive optimization and tuning parameter for the Haswell architecture on Euler. With GCC 6.3.0, the Eigen3 kernel is roughly $1.1\times$ faster (for single and double precision) than the ISPC kernel which is a satisfactory result given the amount of time we spent on optimizing the kernel.

- **1 point for correct SSE2 single precision expectation**
- **1 point for correct SSE2 double precision expectation**
- **1 point for correct AVX2 single precision expectation**
- **1 point for correct AVX2 double precision expectation**

e) If there is a non-zero error associated with your optimized kernel, explain the reason for this error.

It is possible to observe a non-zero error due to different *instruction order* for the two compiled versions in the benchmark. Recall that floating point addition and multiplication are both commutative but not necessarily associative or distributive. Therefore, variation of instruction order can cause an error which must be within machine precision.

- **2 points if any of these keywords are contained in your explanation: *instruction order, commutative, associative, distributive, machine precision***

f) Do you observe differences in errors generated by the SSE2 and AVX2 instruction sets? If so, why is that?

The SSE2 code computes an exact result because the extended instruction set does not contain fused-multiply-add (FMA) instructions. Hence, the compiler generates code with the same instruction order (by using addition and multiplication intrinsics) because our algorithm for the vectorized version is the same as our baseline (both use the same temporary buffer layout). For the AVX2 code, the algorithm is yet the same but the AVX2 extended instruction set does include FMA instructions, which are heavily utilized by the compiler. FMA instructions have a different rounding error policy, which means that a multiply followed by an addition (two single instructions) does not result in the same rounding error if compared to an FMA (one single instruction). Consequently, the result carries a different rounding error and *is not* the same as in the baseline kernel. You may force ISPC to disable FMAs by adding the option `--opt=disable-fma` or use the `nofma=true` option for the `make` command.

[4] `http://eigen.tuxfamily.org`

- **1 point for mentioning fused-multiply-add (FMA)**
- **1 point for mentioning rounding error**

Question 3: Lapack routines (22 points)

22 points total

As we have seen in the prevoius exercise it is cumbersome to always write our own subroutines for often encountered problems in numerics. A lot of common routines are therefore available in BLAS and LAPACK libraries. BLAS routines are ordered in three levels. Level one routines contain vector-vector operations, level 2 routines contain matrix-vector operations and level 3 routines contain matrix-matrix operations. LAPACK routines use BLAS routines to perform high level matrix operations such as factorizations. In this exercise we will explore different LAPACK routines for solving linear systems of equations. The library in question is the Intel MKL (Math Kernel Library).

You can load the MKL library on euler with "`module load intel/2020.0`".

Cubicsplines are a useful tool for interpolation, since the resulting function is differentiable and continous. Cubic splines are piecewise polynomials where the second derivative is assumed to be linearly interpolated between two datapoints.

$$f'' = f_i''\frac{x_{i+1}-x}{\Delta_i} + f_{i+1}''\frac{x-x_i}{\Delta_i} \qquad \Delta_i = x_{i+1}-x_i \tag{3}$$

The constants f'' are unknown parameters that have yet to be found. By integrating twice and enforcing $f(x_i) = y_i$ we obtain:

$$f = f_i''\frac{(x_{i+1}-x)^3}{6\Delta_i} + f_{i+1}''\frac{(x-x_i)^3}{6\Delta_i} + \left(\frac{y_{i+1}-y_i}{\Delta_i} - \frac{(f_{i+1}''-f_i'')\Delta_i}{6}\right)(x-x_i) + y_i - f_i''\frac{\Delta_i^2}{6} \tag{4}$$

To solve for the unknown f_i'' we enforce continous first derivatives at the datapoints. this results in the following equation:

$$f_{i-1}''\frac{\Delta_{i-1}}{6} + f_i''\frac{\Delta_{i-1}+\Delta_i}{3} + f_{i+1}''\frac{\Delta_i}{6} = \frac{y_{i+1}-y_i}{\Delta_i} - \frac{y_i-y_{i-1}}{\Delta_{i-1}} \tag{5}$$

If our dataset contains N datapoints this allows us to formulate $N-2$ equations, because the equation is not valid at the ends. Therefore we set enforce the boundary condition $f_1'' = f_N'' = 0$ to obtain natural splines.

a) Formulate the linear system of equation $Ax = b$ for N datapoints and write down what the entries of A and b are. What are the dimensions of A and b?

The matrix A is an $N-2 \times N-2$ tridiagonal matrix where the entries of the principal diagonal are $b_i = \frac{\Delta_{i-1}+\Delta_i}{3}$ (using zero based indexing for the data points). The entries on the lower and upper diagonal are $a_i = \frac{\Delta_{i-1}}{6}$ and $c_i = \frac{\Delta_i}{6}$ respectively. Notice that $a_i = c_{i-1}$ which makes this matrix symmetric. The vector b has dimensions $1 \times N-2$ and its entries are given by $d_i = \frac{y_{i+1}-y_i}{\Delta_{i+1}} - \frac{y_i-y_{i-1}}{\Delta_i}$.

$$A = \begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & \ddots & \\ & & \ddots & \ddots & c_{N-3} \\ 0 & & & a_{N-2} & b_{N-2} \end{bmatrix} \qquad b = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{N-2} \end{bmatrix} \tag{6}$$

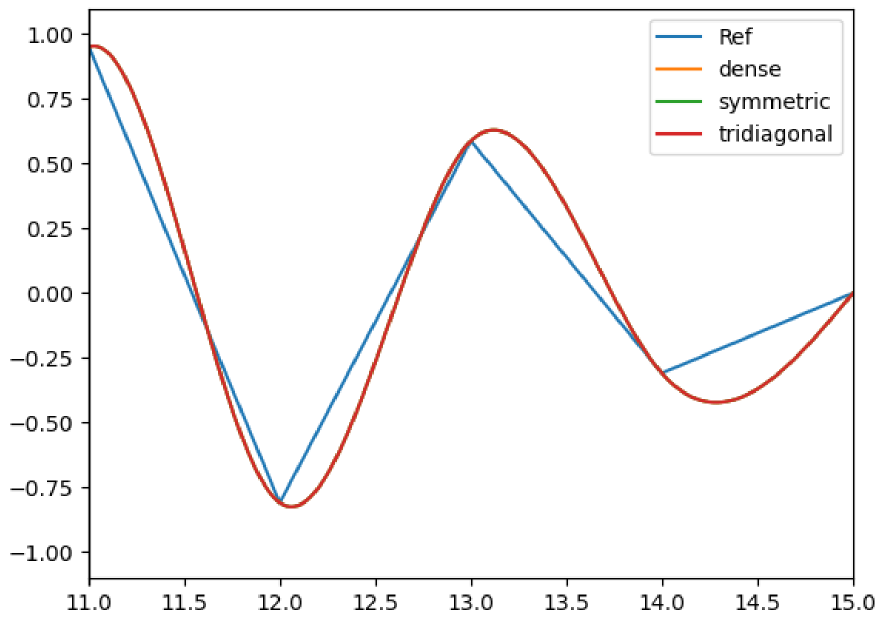


Figure 3: Result of the interpolated dataset.

- 2 points for correct dimensions of A and b
- 2 points for correct A matrix
- 1 point for correct b vector

b) In the file `main.cpp` there is an skeleton code for calculating the parameters of a cubicspline interpolation for the provided dataset with $N = 1000$ datapoints. We want to compare the performance of three solving algorithms for dense (general), symmetric and tridiagonal matrices from the Intel MKL LAPACK library.

i) Name the function calls to the three routines for the double datatype.
Hint: The Intel LAPACK function advisor might be useful.
The three functions are called `LAPACKE_dgesv`, `LAPACKE_dsysv` and `LAPACKE_dgtsv`.

- 1 point per correct funtion

ii) Initialize the arrays with the values of subquestion a) according to the documentation for the three routines.

- 2 points for correct initialization of matrix A for dense and symmetric routines
- 1 point for correct initialization of principal diagonal array for tridiagonal routines
- 1 point for correct initialization of off-diagonal of A
- 1 point for right hand vector b

iii) Use the three routines to calculate the parameters for the cubicsplines. The program will save the results of the solution vectors in three files which can be plotted with the provided python script to check for correctness. The python script will take care of enforcing the boundary condition. The result vector should therefore only contain the unknown f_i'' that are found by solving the system of equations. Report the measured time of the three routines.

For the interested reader: In this exercise we want to show you that just using the best implementation of an algorithm can be suboptimal if a more efficient algorithm can be used. The dense solver will perform a LU decomposition to solve the system of equations which has time complexity of $\mathcal{O}(\frac{2}{3}n^3)$ the symmetric routine exploits the symmetry and performs a Cholesky decomposition with a time complexity of around $\mathcal{O}(\frac{1}{3}n^3)$. In contrast the tridiagonal solver has a time complexity of $\mathcal{O}(n)$ Which will always be a lot faster than the other two implementations. To better illustrate the performance the same problem has been run on Euler with 35'000 datapoints. The reported timings were:

- Dense routine needed: 1416.6s
- Symmetrical routine needed: 865.428s
- Tridiagonal routine needed: 0.0041838s

This more or less matches the performance that we expect given the time complexities of the different solvers.

- **2 points for each correct function call**
- **1 point for correct plot**
- **2 points for reporting times**