ETH zürich

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)

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.

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?

  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.

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

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

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.

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
```

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

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.

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.

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

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

Question 3: Lapack routines (22 points)

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?

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.

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

   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.

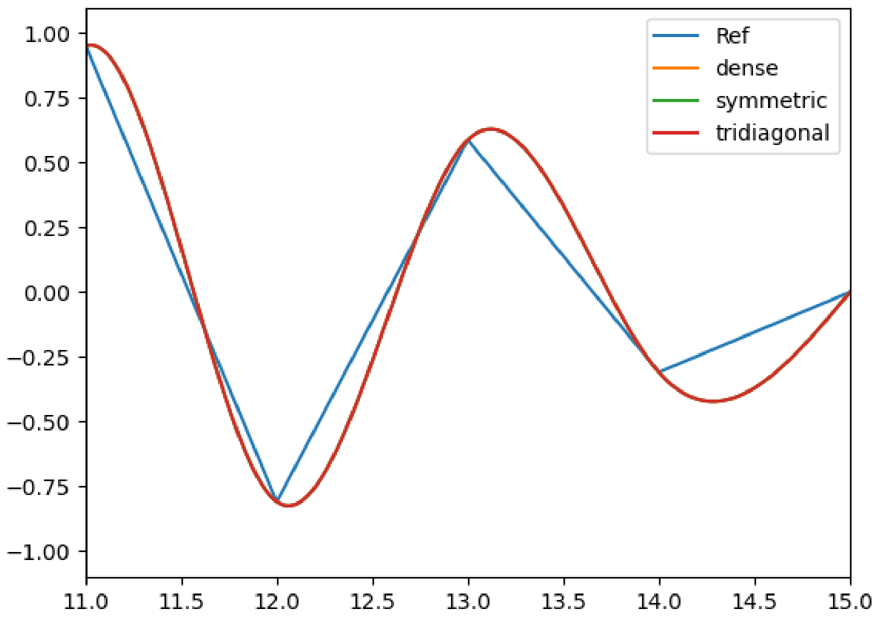

Figure 1: Result of the interpolated dataset.