Vector code generation and high level programming for finite element simulation

Angel Hippolyte

Summer Internship Project
Report

Vector code generation and high level programming for finite element simulation

Submitted by

Angel Hippolyte
Computer Science Master 1
Université de Bordeaux

Under the guidance of

Olivier Aumage
Team leader





Project Team STORM
INRIA
200 Av. de la Vieille Tour, 33405 Talence

Summer Internship 2023

Project Team STORM

INRIA

Olivier Aumage
(Project Guide)

Date: 15/05/2023 - 12/08/2023

Contents

My internship took place from May 15, 2023 to August 12, 2023 at the research center Inria Bordeaux Sud-Ouest within the STORM team and was under the supervision of Mr. AUMAGE Olivier (STORM team leader). I also worked with Mr. POPOV Mihail, Mrs. GUERMOUCHE Amina and Mr. LIMA Laercio from Inria and Mr. TROTTER James from SIMULA in Norway.

Chapter 1

Introduction

In the world of science, simulations are important as it enables to virtually predict the behavior of complex objects whose real-life experimentations would be hard to put into practice if not impossible. The FEniCSx computing platform enables to perform such simulations without requiring low level programming skills. As the simulations are compute intensive, we can improve the performance by using the SIMD architecture provided into modern processors and then reduce execution time. How can we let FEniCSx simulations benefit from SIMD instruction sets ? Before we get down to the heart of the matter, we will introduce a few important concepts for the rest of the study. Then let us define the method developed to achieve our goals. From the development of a microbenchmark program enabling to gather various data, to the analysis of these data in order to retrieve some relevant information. Finally, we are going to evaluate the results obtained using various graphs.

Chapter 2

Background

2.1 FEniCSx Computing Platform

FEniCSx is a widely-used open-source computing platform that tackles partial differential equations (PDEs). This platform enables users to efficiently translate scientific models into finite element code. Whether you are new to it or an experienced programmer, FEniCSx offers a user freindly interface for diverse programers through its high-level Python and C++ interfaces. [1] The FEniCSx computing platform comprises multiple components:

The Unified Form Language (UFL) is a specialized language used to declare finite element discretizations of variational forms and functionals. Its primary purpose is to provide a versatile interface for defining finite element spaces and expressions for weak forms, presented in a notation that closely resembles mathematical notation. UFL can be written using Python and C++.[2]

FFCx serves as a compiler specifically designed for finite element variational forms. By taking a high-level description of the form in the Unified Form Language (UFL) written in Python as input, FFCx is capable of generating efficient low-level C code. This code can then be utilized to assemble the corresponding discrete operator with optimized performance.[3]

DOLFINx offers a comprehensive problem-solving environment tailored for models that rely on partial differential equations. It encompasses essential elements of FEniCS functionality, incorporating data structures and algorithms necessary for computational meshes and finite element assembly. In essence, DOLFINx provides a versatile framework for effectively tackling partial differential equation-based models.[4] `DOLFINx` provides the portability of `FEniCSx` as it links up the system and the additionnal softwares: `MPI` for communications, `PETSc` for linear algebra etc.

In a nutshell: To define a simulation experience, a scientist would write some high level code in UFL which is very close from the mathematical notation. Then, this code is going to be compiled with FFCx to generate a C code that can be used to assemble the

corresponding discrete operator. Finally, DOLFINx is used to define and solve the global problem at top-level using the Basix library in C++. During this study, we are going to focus on the efficiency of the C file generated by FFCx as it is going to be compiled to a binary executable using GCC, the GNU compiler.

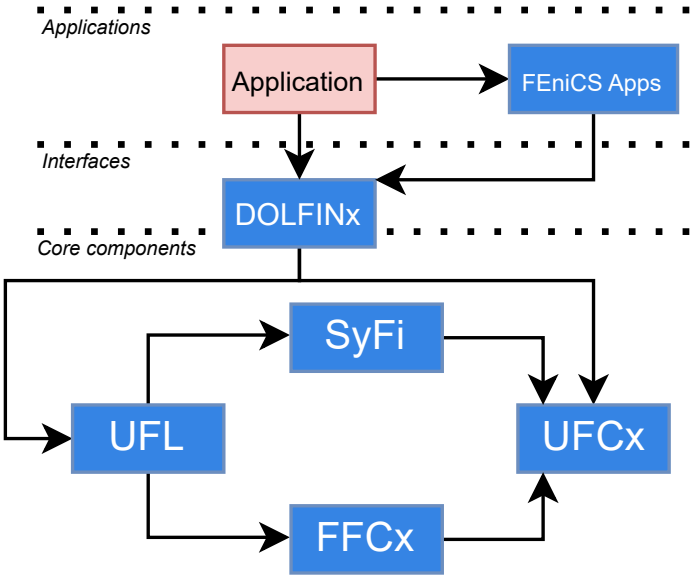


Figure 2.1: DOLFIN functions as the main user interface of FEniCS and handles the communication between the various components of FEniCS and external software. Solid lines indicate dependencies and data flow.[4]

As FEniCSx is extendable for the use of SIMD, let us define the architecture.

2.2 SIMD Architecture

Single Instruction, Multiple Data (SIMD) is a type of parallel processing in Flynn's taxonomy. SIMD can be internal (part of the hardware design) and it can be directly accessible through an instruction set architecture. SIMD describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously.[5]

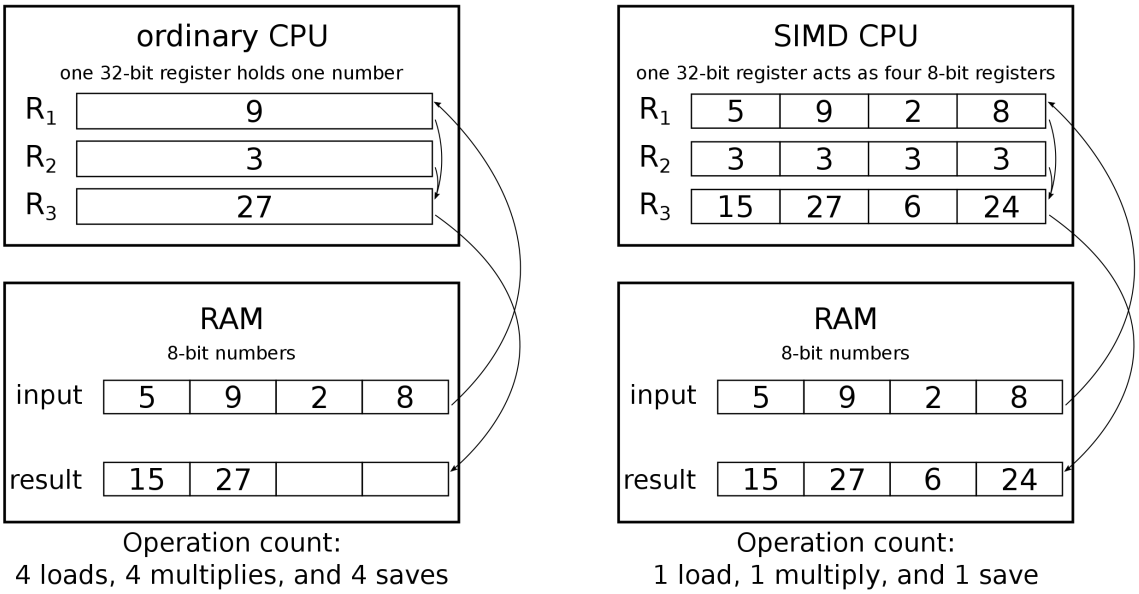


Figure 2.2: The ordinary CPU has to load, multiply and save for each 8-bit number (12 operations) whereas the SIMD CPU would load every 8-bit number inside a vector (3 operations).

In the example of figure 2.2, the SIMD CPU is computing a single instruction on a 32-bit register but modern SIMD architectures can use up to 512-bit registers.

```
__m256d vect1 = _mm256_setr_pd(1.0, 2.0, 3.0, 4.0);
__m256d vect2 = _mm256_set1_pd(2.0); // := [2, 2, 2, 2]
__m256d mul = _mm256_mul(vect1, vect2);
```

Figure 2.3: Example of the use of AVX2 instructions from the intel intrinsics library (256-bit registers). Two vectors are loaded and multiplied together.

The `__m256d` type declares a 256-bit register containing 4 double-precision floats (64 bits each). The result of `mul` is `[2.0, 4.0, 6.0, 8.0]`.

Takeaway: The use of SIMD architecture represents a good opportunity to improve the performance but it is not always easy to take advantage of it. Indeed, different hardwares have different simd instructions and the compiler is not able to easily select them.

Now that we're familiar with SIMD, it's interesting to focus on few compilation flags that can help us use it.

2.3 Compilation Flags

At compilation time, there are some flags which enable the compilator to enhance the assembly code generated and thus improve the execution performance. Among these, we are particularly interested in 4 flags:

- `-fassociative-math` enables re-association of operands in series of floating-point operations.
- `-ffast-math` Enables a range of optimizations that provide faster, though sometimes less precise, mathematical operations.
- Global levels of optimization (often used for debugging)
  - `-O0`: no optimization at all
  - `-O1`: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
  - `-O2`: performs nearly all supported optimizations that do not involve a space-speed tradeoff
  - `-O3`: turns on all optimizations specified by `-O2` and also performs many loop improvements including **auto-vectorization** (cf. 2.2). `-O3` is an agressive set of optimizations that may result in poor performance in some scenarios.

Takeaway: Compilation flags give a lot of control to the developer but also plenty situations to explore.

Now we know what SIMD and compilation flags are, we need to define the context we are going to study it.

2.4 Simulation Kernels

2.4.1 Definition

A simulation kernel can be defined (roughly) by the attributes below:

- A shape of a given dimension (2D or 3D) to define the object to realize the simulation on (see Figure 2.4)
- A mesh splitting the shape into several cells (see Figure 2.5)
- Some degrees of freedom defining the amount of quadrature points on each cell (see Figure 2.6)
- A linear partial differential equation which describe the behavior of each quadrature point

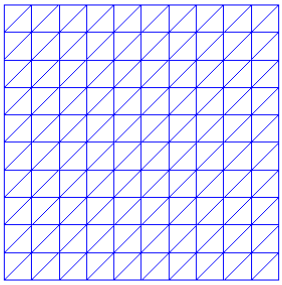
2.4.2 Assembly

When the C file is generated from the UFL code, the most computation intensive part is located into the integral functions which are designed to compute the result of the simulation at the level of one cell from the coordinates of its quadrature points (**local assembly**). The purpose of these functions is to approximate the solutions of a linear partial differential system. On top of that, the program iterates over each cell, gathering local assemblies together in order to form one matrix containing the overall simulation result (**global assembly**).

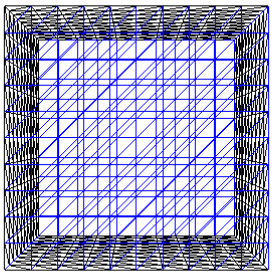
2.4.3 Compute Intensity of Integrals

Figure 3.1 shows a basic example of UFL code which can be used to solve the poisson equation with 1 degree of freedom using triangle cells. Using FFCx to compile this code would generate a C code containing several features, namely 3 functions similar to the ones from Figure 3.2. The higher the degrees of freedom and the dimension of the shape, the higher the number of iterations and the vector sizes. So the complexity of the global problem is reflected in the local problem.

Figure 2.4: Simulations can be ran on rectangle 2D surfaces or even 3D shapes such as the cube. The choice of the simulation shape will involve more or less computation complexity.

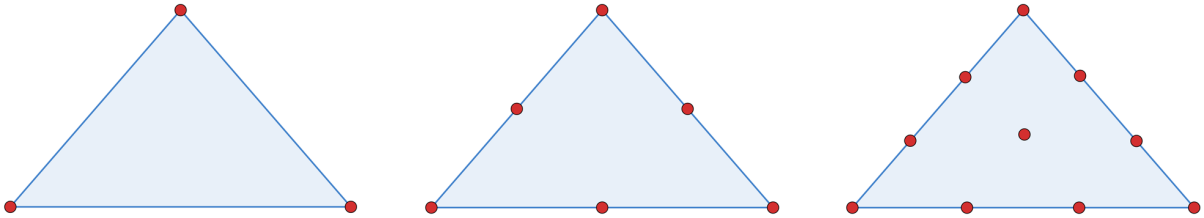


(a) A square can be meshed with triangle cells, that is to say divided into several squares containing 2 triangles. $L \times H \times 2$ cells.



(b) A cube can be meshed with tetrahedron cells, that is to say divided into several squares containing 2 tetrahedrons. $L \times l \times H \times 2$ cells.

Figure 2.5: 2D and 3D mesh examples [6]



(a) 1 degree of freedom (b) 2 degrees of freedom (c) 3 degrees of freedom

Figure 2.6: Different degrees of freedom for a 2D triangle cell

Chapter 3

Method

Now that we have assimilated the necessary knowledges regarding the FEniCSx computing platform, the architectural possibilities and the different tweakable compilation flags, we can get to the heart of the matter. It begins with the method used to measure and analyse the performance and the energy consumption of the different kernels.

```
    element = FiniteElement("Lagrange", triangle, 1)

    coord_element = VectorElement("Lagrange", triangle, 1)
    mesh = Mesh(coord_element)

    V = FunctionSpace(mesh, element)
    u = TrialFunction(V)
    v = TestFunction(V)
    f = Coefficient(V)
    g = Coefficient(V)
    kappa = Constant(mesh)
    a = kappa * inner(grad(u), grad(v)) * dx
    L = inner(f, v) * dx + inner(g, v) * ds
```

Figure 3.1: UFL code of the poisson kernel for a 2D simulation with 1 degree of freedom. Both highlighted lines are defining the degrees of freedom and the cell type of the mesh.

Given the preceding code, we cannot predict which combination of compilation flags will provide the best performance and the least energy consumption. Also, some flags could improve the auto-vectorization capacities of the compilator which can be interesting.

```
void tabulate_tensor_integral_cca08(
    double*  restrict A, const double* restrict w,
    const double* restrict c,
    const double* restrict coordinate_dofs,
    const int* restrict entity_local_index,
    const uint8_t* restrict quadrature_permutation
){
    ...
    for (int ic = 0; ic < 3; ++ic)
    {
        J_c0 += coordinate_dofs[ic * 3] * FE4_C0_D10_Q48e[...][ic];
        J_c3 += coordinate_dofs[ic * 3 + 1] * FE4_C1_D01_Q48e[...][ic];
        J_c1 += coordinate_dofs[ic * 3] * FE4_C1_D01_Q48e[...][ic];
        J_c2 += coordinate_dofs[ic * 3 + 1] * FE4_C0_D10_Q48e[...][ic];
    }
    ...
    for (int iq = 0; iq < 3; ++iq)
    {
        const double fw0 = sp_48e[22] * weights_48e[iq];
        const double fw1 = sp_48e[21] * weights_48e[iq];
        const double fw2 = sp_48e[20] * weights_48e[iq];
        double t0[6];
        double t1[6];
        for (int i = 0; i < 6; ++i)
        {
        t0[i] = fw0 * FE8_C0_D10_Q48e[0][0][iq][i]
        + fw1 * FE8_C0_D01_Q48e[0][0][iq][i];
        t1[i] = fw1 * FE8_C0_D10_Q48e[0][0][iq][i]
        + fw2 * FE8_C0_D01_Q48e[0][0][iq][i];
        }
        for (int i = 0; i < 6; ++i)
            for (int j = 0; j < 6; ++j)
            A[6 * i + j] += FE8_C0_D10_Q48e[0][0][iq][j] * t0[i]
            + FE8_C0_D01_Q48e[0][0][iq][j] * t1[i];
    }
}
```

Figure 3.2: Poisson integral function for 6 quadrature points on a 3D shape. We mainly note the several loops that increase the arithmetic intensity.

3.1 Local and global assemblies

As the computations operated in the loops from the integral functions are not subject to complex dependencies (no conditions, simple table calculations) 3.2, we can expect the compilator to operate some optimizations to improve execution time, such as unrolling

the loops and vectorizing them by generating SIMD instructions. Then we target to improve the efficiency of the integral functions. As a consequence, the global kernels efficiency should also be improved. Therefore, each study will be carried out both on the independent integrals and on the global kernels. Thus, it is going to enable us to evaluate how much improving the local assembly improves the global one.
As a reference, we are going to use 8 different kernels of poisson with 1 to 4 degrees of freedom for both 2D and 3D shape (rectangle and cube). The 8 global kernels are going to be used as a representation of the main result but we are also going to investigate in each integral function (3 for each kernel) as it could give some additional information.

3.2 Experimentation Protocol

To make simulations, we are going to need a program that enables us to choose whether we want to execute the global kernels or only the local integral functions independently, a specific number of meta-repetitions, and a specific metric to measure (time or energy).

3.2.1 Micro-benchmark Program

Main C++ Program

The main C++ program is designed to run different types of simulation on demand. The execution performance is measured in seconds, cache + core and DRAM energy consumption in Joule (J) and arithmetic intensity in FLOPS/s (floating-point operations per second). The program is designed to run each operator several times within the same process (meta-repetitions), providing a warmup round before taking final measurements. The warmup section is necessary to avoid execution from "cold cache" which would not be representative of a real context execution. Each simulation experience result is written on the standard output (see Figure 3.3) and logged into a `csv` file (see Figure 3.4). The energy consumption measurement is a bit more complex and is going to be detailed into the next section.

```
./benchmark <mode> <metrics> <reps>
```

- `<mode>` can be `--global` or `--local` and specifies whether we prefer to run the simulation on global kernels or on integral functions independently.
- `<metrics>` enbales to choose between a time execution performance measurement with `--time` and energy consumption or arthmetic intensity measurement with `--likwid`
- `<reps>` is set to 1 by default but you can specify it to make your experiences run several times.

```
[ahippoly@devel02 _build]$ ./benchmark --global --time 2
--------------------------------------------------------------------------
WARNING: There was an error initializing an OpenFabrics device.

  Local host:   devel02
  Local device: hfi1_0
--------------------------------------------------------------------------
Starting simulation of operator 'poisson - triangle - 1'
   - Warmup... Done.
   - terminated in 0.3090646030 sec
   - terminated in 0.3096402080 sec
Starting simulation of operator 'poisson - triangle - 2'
   - Warmup... Done.
   - terminated in 0.5043167120 sec
   - terminated in 0.5178613090 sec
Starting simulation of operator 'poisson - triangle - 3'
   - Warmup... Done.
   - terminated in 1.1230217170 sec
   - terminated in 1.1700759180 sec
Simulation ended successfully.
```

Figure 3.3: C++ benchmark program execution instance measuring the performance of the global kernels with 2 repetitions.

```
operator;vecto;opt;fastmath;fassociativemath;time
```

Figure 3.4: CSV simulation entry model informing the kernel name (operator), the compilation flags combination and the execution time given in seconds.

Python Simulation Program

The python part is the top-level program used to run the simulations. It enables automated serial compilation and execution of the C++ benchmark program. The main function of this program is to iterate over every flag combination. Then for each iteration, the C++ benchmark program is compiled with the corresponding flags and then executed.

```
options = {
    "vecto": ["", "auto", "avx2"],
    "opt": ["", "O1", "O2", "O3"],
    "fastmath": [True, False],
    "fassociativemath": [True, False],
}

for vecto in options['vecto']:
    for opt in options["opt"]:
        for fastmath in options["fastmath"]:
            for fassociativemath in options["fassociativemath"]:
                flags = {
                    "vecto":vecto,
                    "opt":opt,
                    "fastmath":fastmath,
                    "fassociativemath":fassociativemath
                }
                print("- Compilation... ")
                compilation = Compilation(flags, metrics, vecto)
                compilation.start()
                print("- Execution of benchmark program... ")
                execution = Execution(mode, metrics, reps, flags)
                execution.start()
```

Figure 3.5: Python top-level simulation program main function that compile and execute the program with each flag combination.

This micro-benchmark program is going to be used throughout the entire study to gather data, analyse it and retrieve relevant informations.

3.2.2 Energy

To profile the energy consumed by the RAM, cache, and cores of the machine when executing different kernels, we are going to call the executable `likwid-perfctr` from the benchmark program to generate a report containing various informations (see Figure 3.7). To target some specific instructions, we can use the API marker provided by the likwid library (see Figure 3.6).

```
likwid_markerInit();
likwid_markerThreadInit();

// void measurement for warmup
likwid_markerStartRegion("void");
likwid_markerStopRegion("void");

likwid_markerStartRegion(char_arr);
// region to measure
likwid_markerStopRegion(char_arr);

likwid_markerClose();
```

Figure 3.6: Use of API markers to measure a specific region within the code. The void region is intended to warmup the caches and avoid an execution from cold caches.

```
TABLE,Region poisson_tri_1_integral_cell_0,Group 1 Metric,ENERGY,11
Metric,Core 1,
Runtime (RDTSC) [s],0.0023,
Runtime unhalted [s],0.0022,
Clock [MHz],2593.9815,
CPI,0.6593,
Temperature [C],201,
Energy [J],0.0936,
Power [W],41.3219,
Energy PP0 [J],0,
Power PP0 [W],0,
Energy DRAM [J],0.0193,
Power DRAM [W],8.5082,
STRUCT,Info,3
CPU name:,Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz,
```

Figure 3.7: LIKWID report example of the operator poisson_tri_1_integral_cell_0.

In the generated report, we are only interested in the amount of energy consumed (in Joule) by the `DRAM` and `PP0` which correspond to the total consumption of the cache and the core components (the isolated core consumption is unavailable on the cluster used). To bring these informations together in a simple line such as Figure 3.4, the generated report is going to be parsed between each execution within the python program.

3.3 Analysis

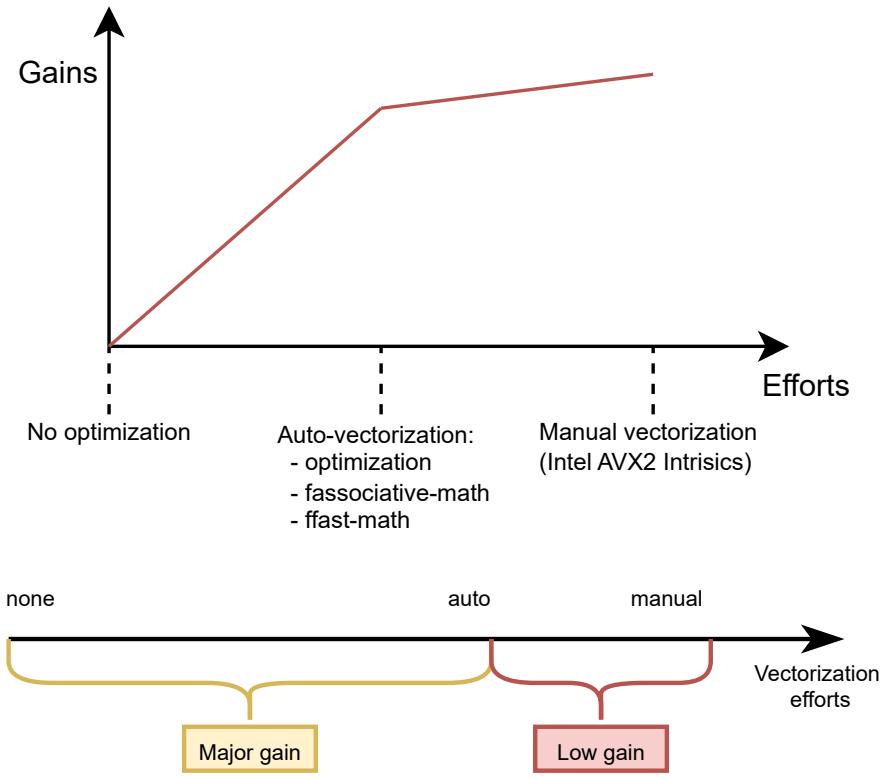


Figure 3.8: Expected gains compared to efforts. The figure shows that the expected main performance improvements are going to be provided by the auto-vectorization. Indeed, the manually vectorized kernels should also provide a substantial improvement. These two versions should show way better performances than the version without optimization at all.

Now we have an efficient tool for running various simulations, it is time to define what kind of data are we going to gather, whose analysis will bring us to the evaluation from chapter 4. The figure above shows the preliminary expectations.

3.3.1 Auto-vectorization

In the first place, we tried to use the tool `MAQAO` which enables to profile the whole execution, giving relevant informations regarding the auto-vectorization process during compilation. Unfortunately, the software is kind of whimsical concerning the context of execution of the kernels. Thus, we had to make our experimentations in a different way, that is to say analysing the performance and energy consumption difference between auto-vectorized and the version of the kernel fully using loops.

Our first aim is to try different flag combination which could ease auto-vectorization :

- `-fassociative-math` : `True`/`False`
- `-ffast-math` : `True`/`False`
- `optimization` : `None`/`O1`/`O2`/`O3`

Each kernel is executed with 128 cells: the 2D shape is a 8×8 square and the 3D shape is a $4 \times 4 \times 4$ cube. Each square is divided into two triangles and each cube divided into two tetrahedrons. $4 \times 4 \times 4 \times 2 = 8 \times 8 \times 2 = 128$ cells. By tweaking these compilation flags, we are going to figure out which of them have the greatest impact on performance and energy consumption. Afterwards, it will be interesting to introduce a new compilation flag to the experiences: `-fno-tree-vectorize` which forces the compilator not to vectorize. This is going enable us to quantify the auto-vectorization by computing the difference of performance and energy consumption between the possibly vectorized kernel and the definitely not vectorized one.

3.3.2 Manual vectorization

Because the manual vectorization is fastidious, we decided not to vectorize every kernel but only 3 of them: the poisson kernel with 1, 2 and three degrees of freedom but it should be enough to realize the benefits from manual vectorization. When it comes to choosing the type of SIMD instructions (size of registers), AVX2 makes a good candidate as it is available on the `bora` nodes from PlaFRIM (see 4.1), also AVX2 enables to compute the operation of 4 double precision floats at the time which is enough on the weaker kernels whereas AVX512 would encourage a lot of non-used vector space. Within the corresponding C files generated by FFCx, we add the vectorized part separated by a macro definition `#define AVX2` which permit us to enable/disable manual vectorization at compilation using the flag `-D AVX2`.

```
#ifdef AVX2
__m256d coordinate_dofs_vect0 = _mm256_setr_pd(...);
__m256d coordinate_dofs_vect1 = _mm256_setr_pd(...);

__m256d mul_J_c0 = _mm256_mul_pd(...);
__m256d mul_J_c1 = _mm256_mul_pd(...);
__m256d mul_J_c2 = _mm256_mul_pd(...);
__m256d mul_J_c3 = _mm256_mul_pd(...);

double J_c0 = mul_J_c0[0] + mul_J_c0[1] + mul_J_c0[2];
double J_c1 = mul_J_c1[0] + mul_J_c1[1] + mul_J_c1[2];
double J_c2 = mul_J_c2[0] + mul_J_c2[1] + mul_J_c2[2];
double J_c3 = mul_J_c3[0] + mul_J_c3[1] + mul_J_c3[2];

__m256d w_vect_0 = _mm256_loadu_pd(w);
__m256d w_vect_1 = _mm256_loadu_pd(w+3);

__m256d weights_39d_vect_0 = _mm256_loadu_pd(weights_39d);
__m256d weights_39d_vect_1 = _mm256_loadu_pd(weights_39d+3);

for (int iq = 0; iq < 6; ++iq){
    double w0 = 0.0;
    __m256d w0_0, w0_1 = _mm256_set1_pd(w0);
    __m256d FE6_C0_Q39d_vect_0 = _mm256_loadu_pd(...);
    __m256d FE6_C0_Q39d_vect_1 = _mm256_loadu_pd(...);
    __m256d mul = _mm256_add_pd(_mm256_mul_pd(...),
    _mm256_mul_pd(...));

    w0 += mul[0] + mul[1] + mul[2];

    double sv_39d[1];
    sv_39d[0] = sp_39d[3] * w0;
    __m256d fw0 = _mm256_set1_pd(sv_39d[0] * weights_39d[iq]);
    __m256d A_vect_0 = _mm256_mul_pd(fw0, FE6_C0_Q39d_vect_0);
    __m256d A_vect_1 = _mm256_mul_pd(fw0, FE6_C0_Q39d_vect_1);
    for (int i = 0; i < 3; ++i)
        A[i] += A_vect_0[i];
    for (int i = 3; i < 6; ++i)
        A[i] += A_vect_1[i%3];
}
#else
// original code
#endif
```

Figure 3.9: Manually vectorized version of the integral function from 3.2.

Chapter 4

Evaluation

In the next section, we are going to present the environment where the programs are going to be executed.

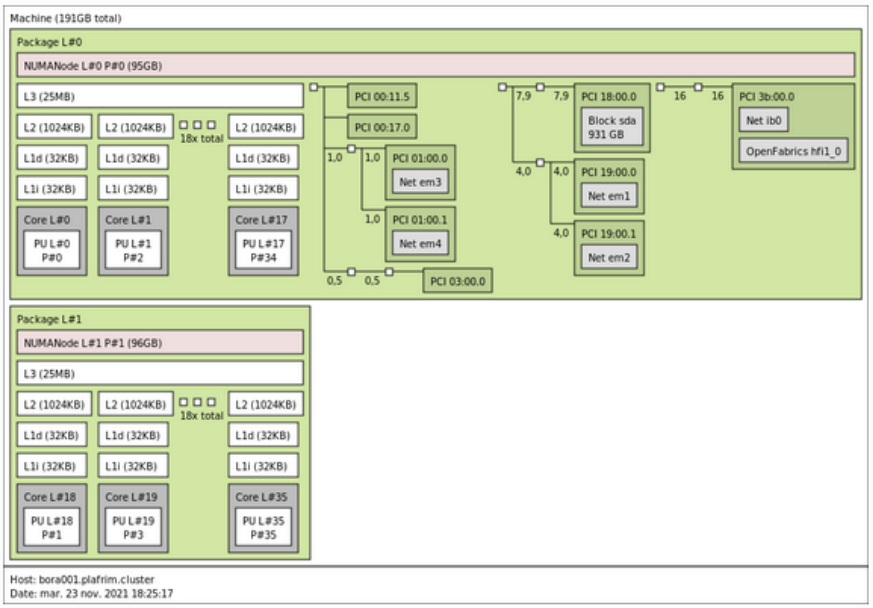
4.1 PlaFRIM

PlaFRIM (Plateforme Fédérative pour la Recherche en Informatique et Mathématiques) is a computation cluster located in Bordeaux (France) that provides users with a large HPC software environment containing widespread compilers, parallel libraries, runtimes, communication libraries, etc [7]. PlaFRIM uses Slurm as a cluster management and job scheduling system which is going to be very useful during the study. We are going to use the nodes called "bora" on PlaFRIM as the microprocessor Xeon Gold 6140 provide **AVX2** and **AVX512** (See Figure 4.1).

CPU

2x 18-core Cascade Lake Intel Xeon Skylake Gold 6240 @ 2.6 GHz (CPU specs).

By default, Turbo-Boost and Hyperthreading are disabled to ensure the reproducibility of the experiments carried out on the nodes.



Memory

192 GB (5.3 GB/core) @ 2933 MT/s.

Network

OmniPath 100 Gbit/s.

10 Gbit/s Ethernet.

Storage

Local disk (/tmp) of 1 To (SATA Seagate ST1000NX0443 @ 7.2krpm).

BeeGFS over 100G OmniPath.

Figure 4.1: Bora nodes architecture

Slurm enables to submit a job defining different simulation settings such as the type and the number of nodes to allocate, the number of tasks to run on each node, the maximum allocation time for a node, the output file etc. (See Figure 4.2). Then a simple run of `sbatch jobsScript.sh` will submit the corresponding job.

```
#!/bin/sh

#SBATCH --job-name=Time
#SBATCH --nodes 1
#SBATCH --ntasks-per-node=1
#SBATCH --time=1-00:00:00
#SBATCH --constraint bora
#SBATCH --exclusive
#SBATCH --output myout.out
#SBATCH --error myout.err

# *shell command*
```

Figure 4.2: Example of job submission on a bora node

4.2 Auto-vectorization

4.2.1 Performance

The following results are produced with the median result of 100 executions for each flag combination.

Local Assembly

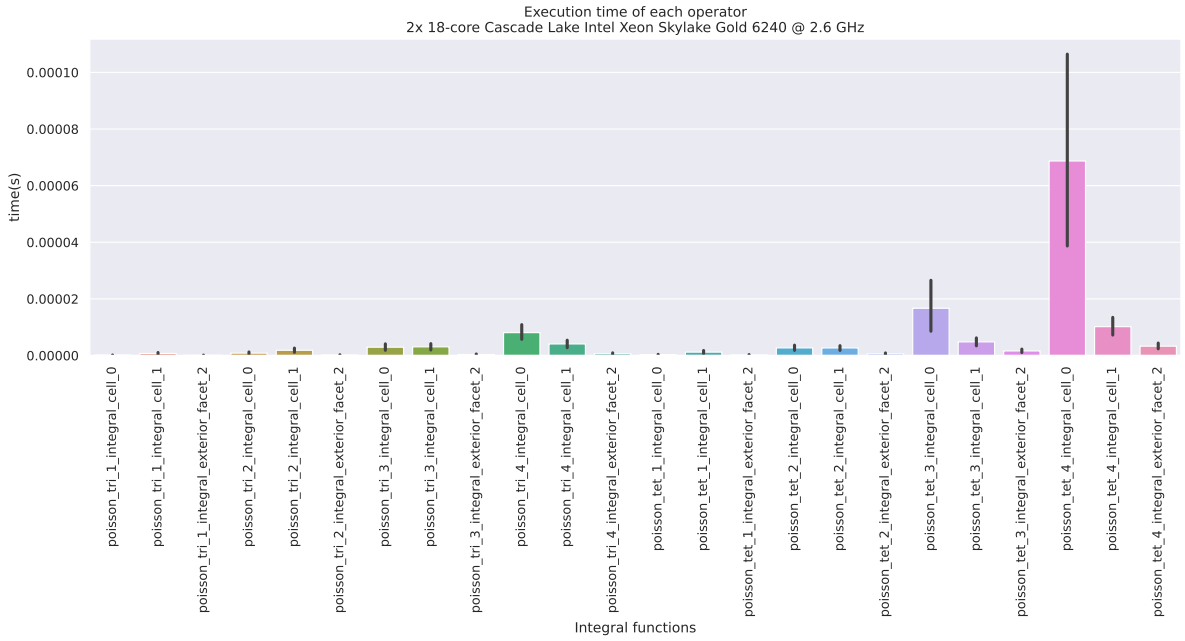


Figure 4.3: Impact of compilation flags on performance. The vertical lines centered on the bars represent the variation of performance with the different flags.

As a first graph, the bar chart 4.3 shows how important it is to select the right combination of compilation flags. The performance difference on the integral functions can be up to ±50%. Now let us take a look at each combination to determine the most important flags.

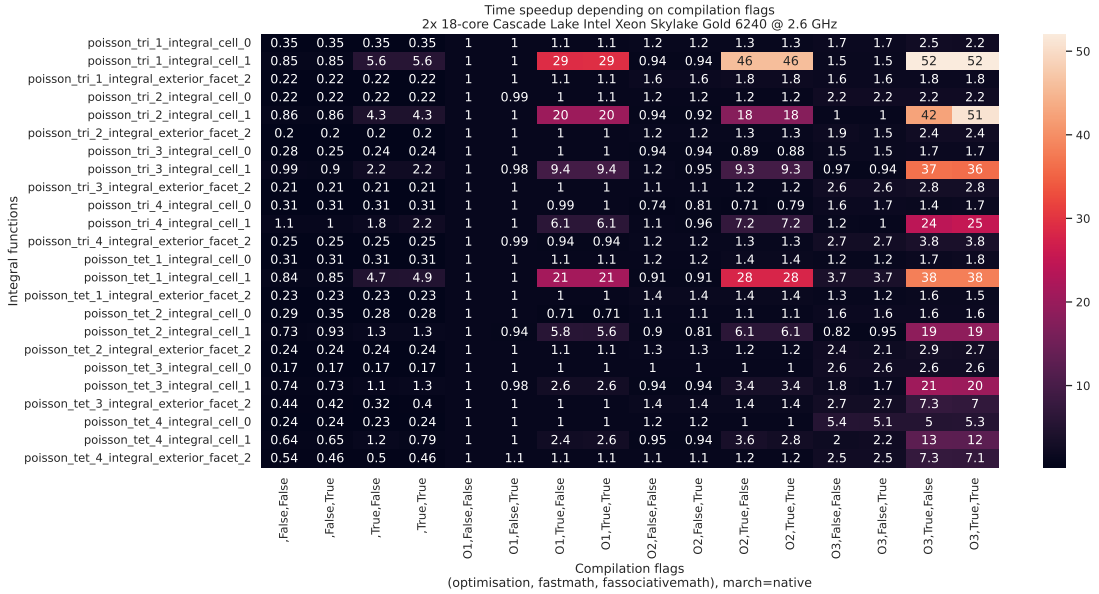


Figure 4.4: Speedup depending on the combination of compilation flags. The combination of reference is the O1 optimization without fastmath or fassociativemath. Each value is computed the following way: $\frac{TimeOfReference}{TimeInSeconds}$

Here we can see that the performance is globally better as we move from the left to the right side of the heatmap. This means that for a given integral function, the execution efficiency is improved as we increase the optimization level from `O1` to `O3`. It is also worth noting that the use of `-ffast-math` has a huge impact on the speedup unlike `-fassociative-math` which doesn't make much of a difference.

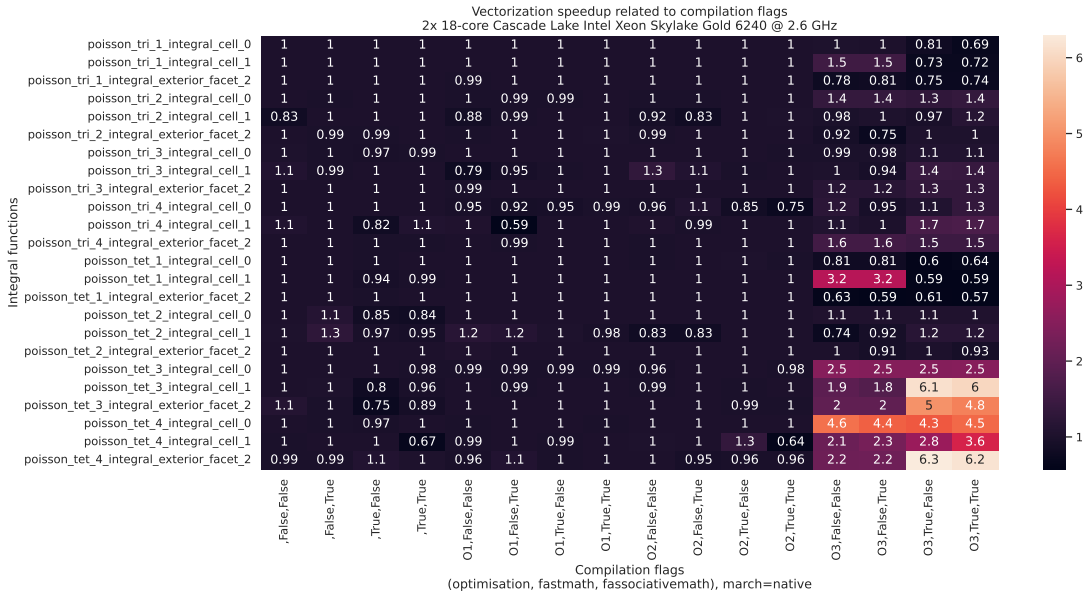


Figure 4.5: Speedup related to auto-vectorization depending on compilation flags. Each value is computed using $\frac{timeWithoutVectorization}{timeWithAutoVectorization}$ so the greater the number, the more efficient the vectorization.

Figure 4.5 shows that the difference of performance is way higher with the level of optimization `O3`, it suggests that the optimization level favors auto-vectorization. Also, the activation of `-ffast-math` shows a slight increase of performance, which can be interpreted as a better ease for the compilator to automatically vectorize when the mathematical operations can be simplified and reordered.

Global Assembly

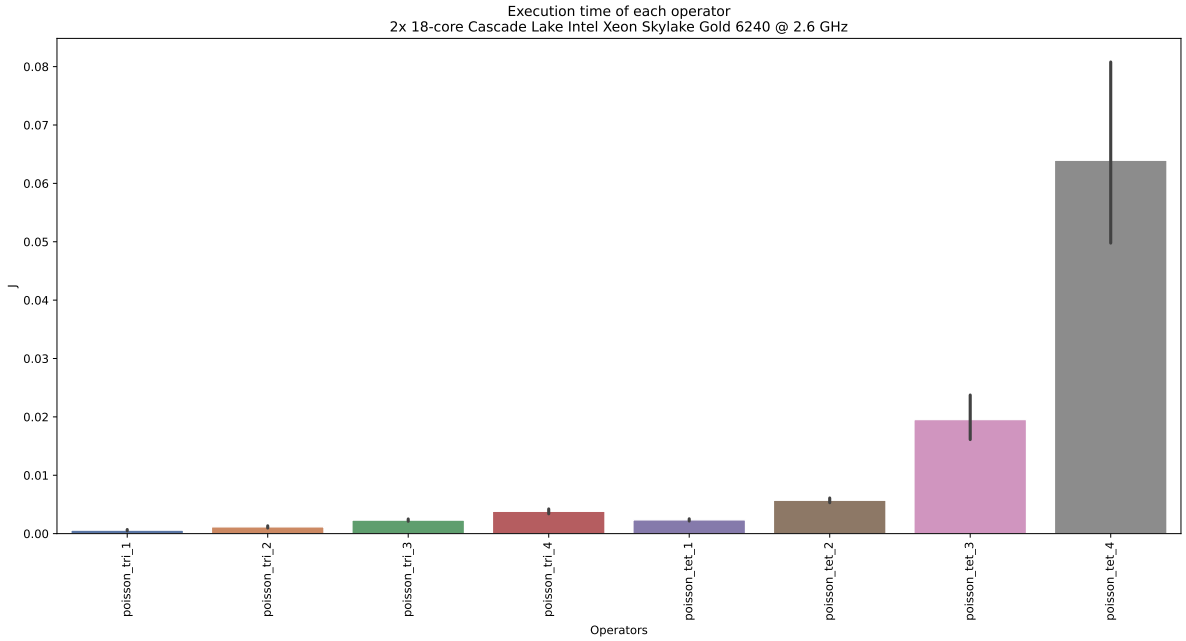


Figure 4.6: Impact of compilation flags on performance. The vertical lines centered on the bars represent the variation of performance with the different flags.

As we can see on the chart 4.6, the compilation flags have less impact on the global assembly performance compared to the local one. As a matter of fact, the simulation complexity also comes with the assembly complexity wich is not directly related with the computation of integrals. The right combination can increase the efficiency by $\sim$ 20% at most. On the other hand, we note that the importance of compilation flags increases with the intensity of the problem which is related to the dimension of the shape, the cell type, and the degrees of freedom (1, 2, 3, or 4).

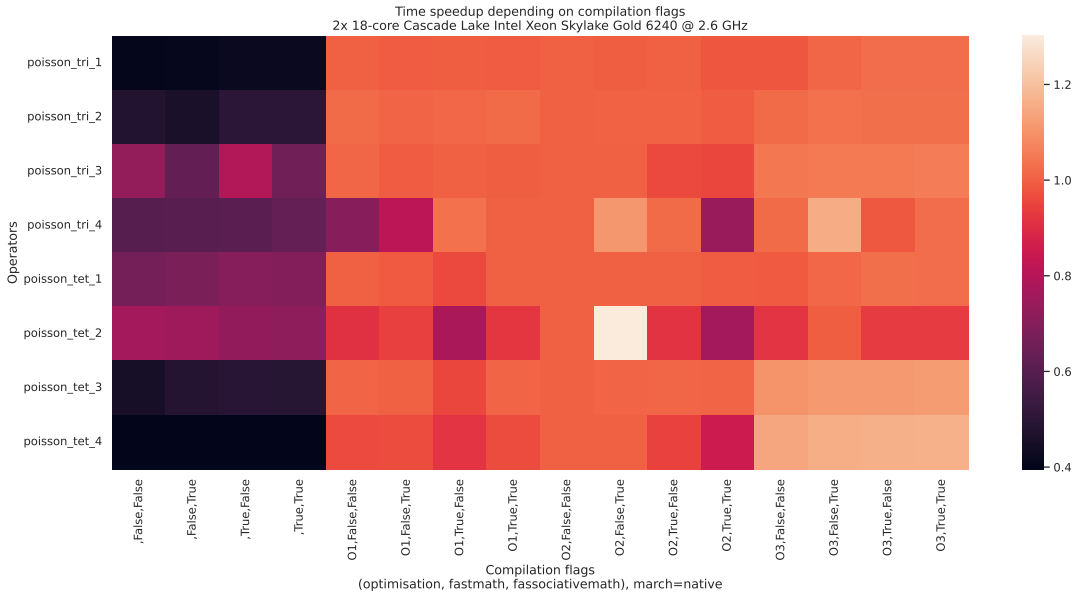


Figure 4.7: Speedup depending on compilation flags combination. The combination of reference is the O1 optimization without fastmath or fassociativemath. Each value is computed the following way: $\frac{TimeOfReference}{TimeInSeconds}$

The heatmap from Figure 4.7 shows that without any optimization, we are loosing between 20% and 60% depending on the kernel and whether we use `-fassociative-math` or `-ffast-math`. On the other hand, the level of optimization `O3` enables to reach up to 20% of performance improvement for the operator `poisson_tet_4`.

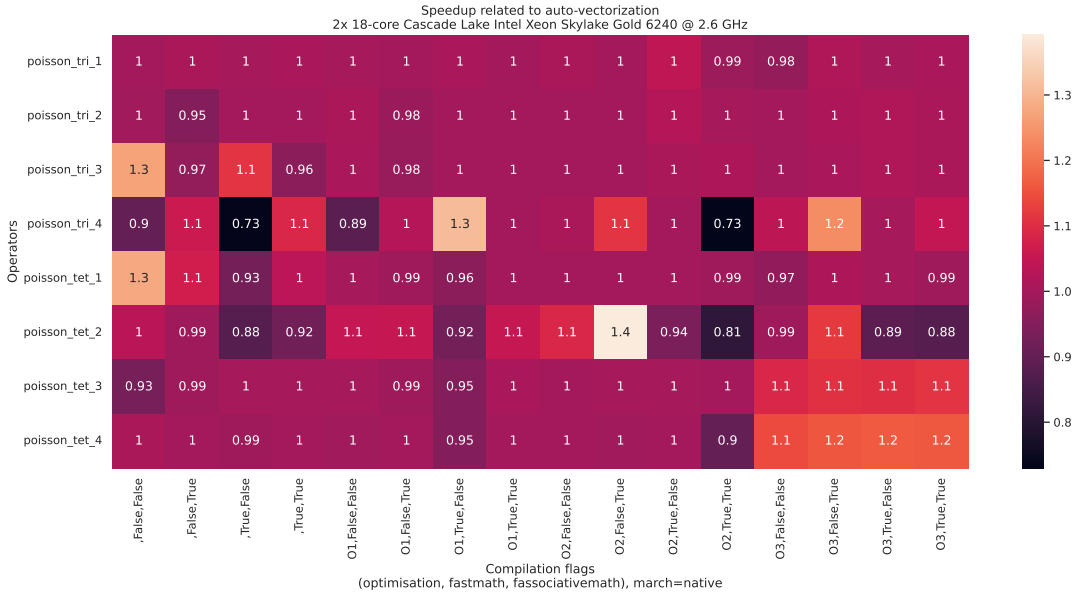


Figure 4.8: Speedup related to auto-vectorization depending on compilation flags. Each value is computed using $\frac{timeWithoutVectorization}{timeWithAutoVectorization}$ so the greater the number, the more auto-vectorization is involved.

Similarly to the local assembly, on figure 4.8 we can see that the best performance

is reached with `O3` (20% improvement). In particular, at this level of optimization, the auto vectorization is enabled so we can expect that this improvement is related to some SIMD instruction generation. We will take the study a step further in the section 4.3 to compare with manually verctorized kernels.

4.2.2 Energy Consumption

Local Assembly

For accuracy reasons, the energy consumption of the integral functions is measured with 100000 executions.

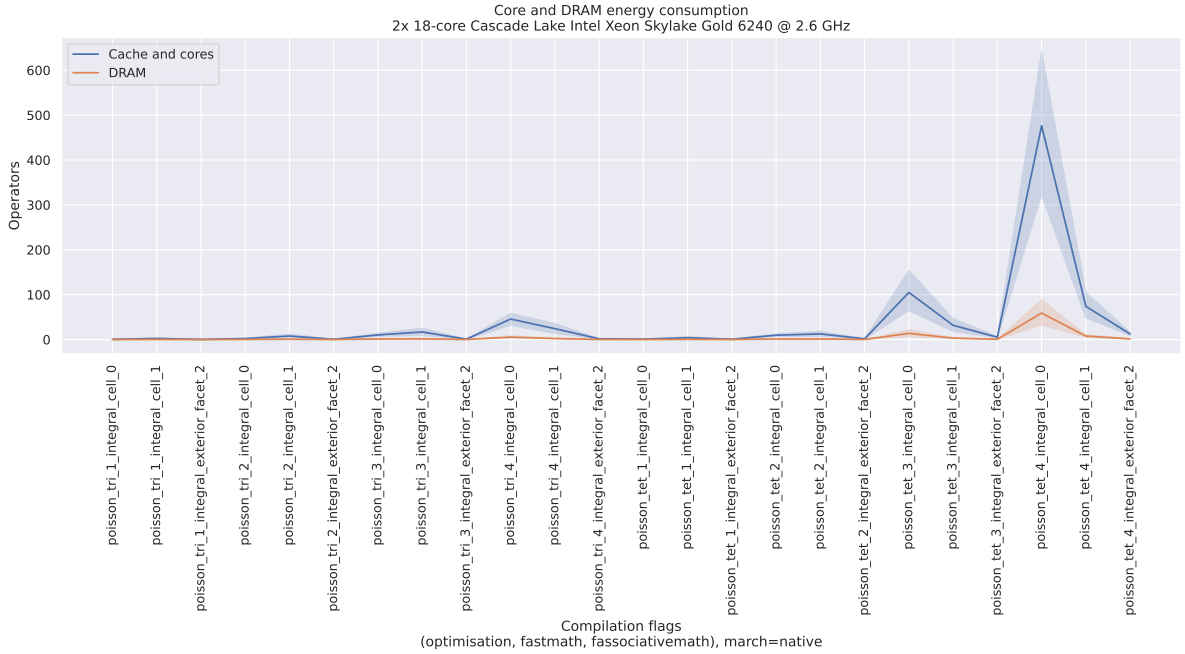


Figure 4.9: Energy consumption on both the DRAM of the machine and the caches and cores of the processor. The values displayed on the left have to be multiplied by 10^{-5} to represent the consumption of a single execution. The shadow part of the lines shows the variation of energy consumption with the different flags.

The figure 4.9 shows that the main source of consumption comes from the caches and the cores, proportionnally followed by the DRAM consumption. We can see that the cache and core consumption can be lowered down to 50% with the right flag combination.

Let's get down to the details to see the consumption associated with each flag combination.

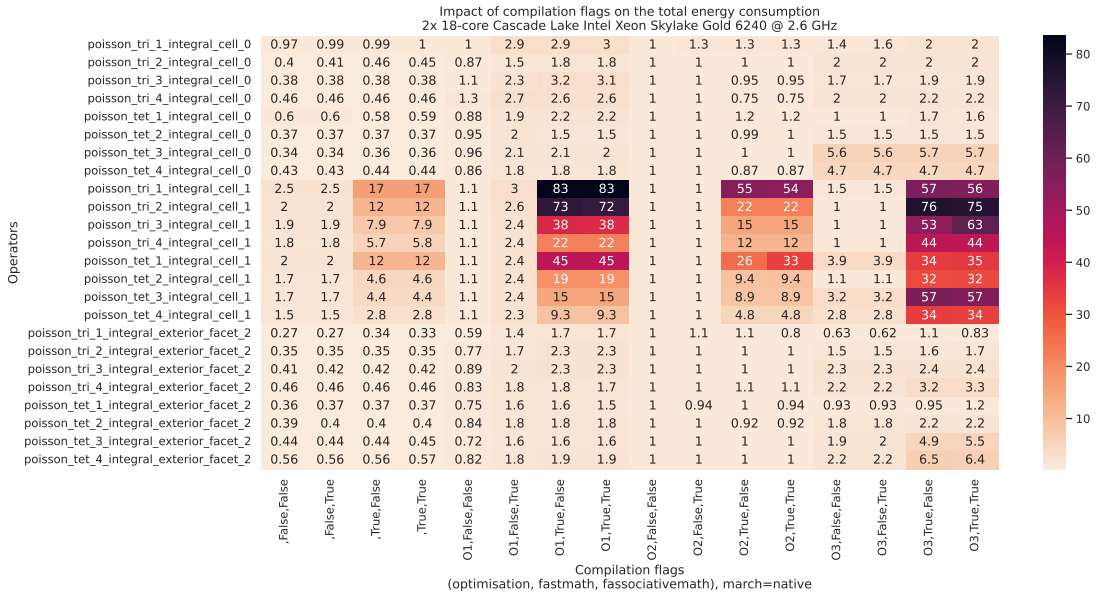


Figure 4.10: Total energy loss factor depending on compilation flags combination (Cache and core + DRAM). The combination of reference is the O2 optimization without fastmath or fassociativemath.

As we could expect, figure 4.10 shows that globally, the energy consumption depends on the optimization level. Afterall, the flag `-fassociative-math` doesn't really affect consumption whereas `-ffast-math` clearly does. Surprisely, consumption doesn't raise linearly as we increase the level of optimization. As a matter of fact, both most energy-consuming levels are `O1` and `O3`.

Now, let us move on the effect of auto-vectorization on energy consumption.

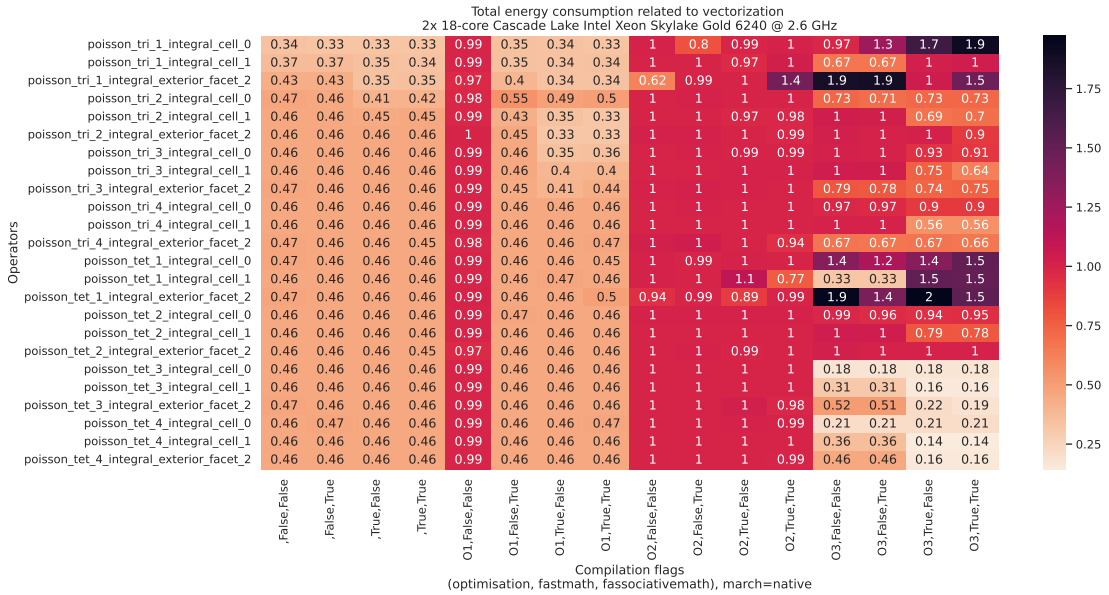


Figure 4.11: Energy loss factor related to auto-vectorization depending on compilation flags. The combination of reference is the O2 optimization without fastmath or fassociativemath. Each value is computed using $\frac{energyWithAutoVectorization}{energyWithoutVectorization}$ so the greater the number, the more auto-vectorization is involved.

Figure 4.11 shows that for most integrals, there is an absolute decrease of energy consumption when the `O3` optimization level is selected. Figure 4.5 precedently showed that auto-vectorization was particularly involved with these six integral functions. It means that the execution of SIMD instructions lowers the consumption.

Now we know the effect of energy consumption on the integrals, let us carry the same kind of study on the global kernels.

Global Assembly

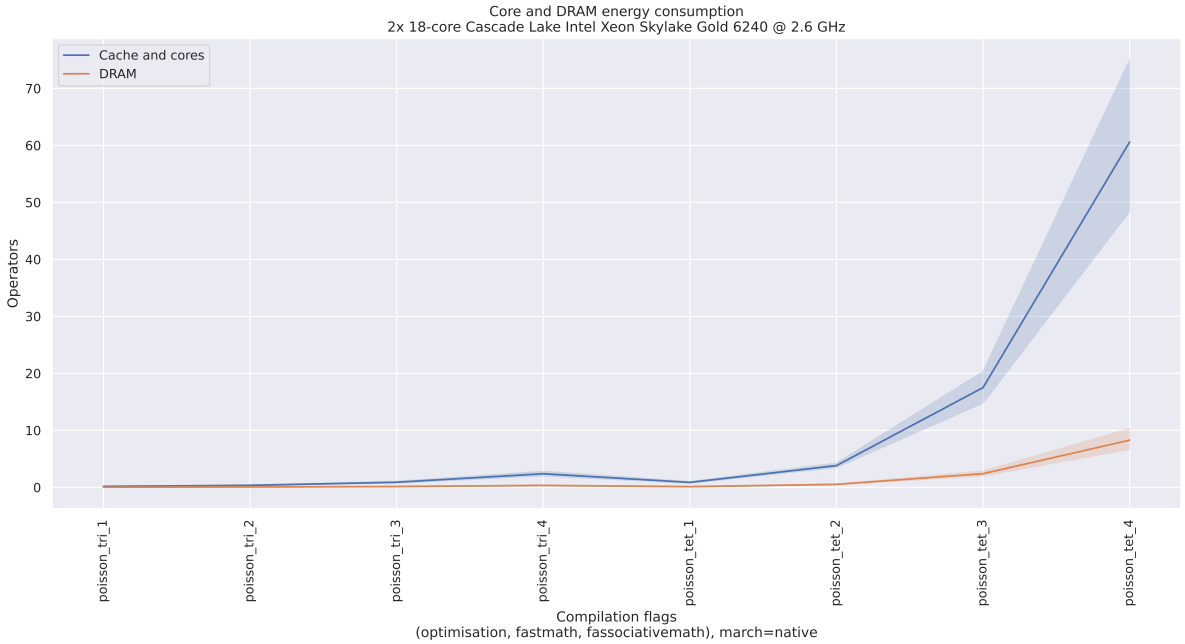


Figure 4.12: Energy consumption on both the DRAM of the machine and the cache and core components of the processor. The shadow part of the lines shows the variation of energy consumption with the different flags.

Figure 4.12 shows that both DRAM and cache and cores energy consumption depends on the kernel's complexity: the higher the execution time of the kernel, the more energy is consumed. Also, compilation flags have a considerable effect on energy consumption of kernel execution. For instance, with the right flag combination, the cache and cores consumption of the operator `poisson_tet_4` can be lowered by about 30%.

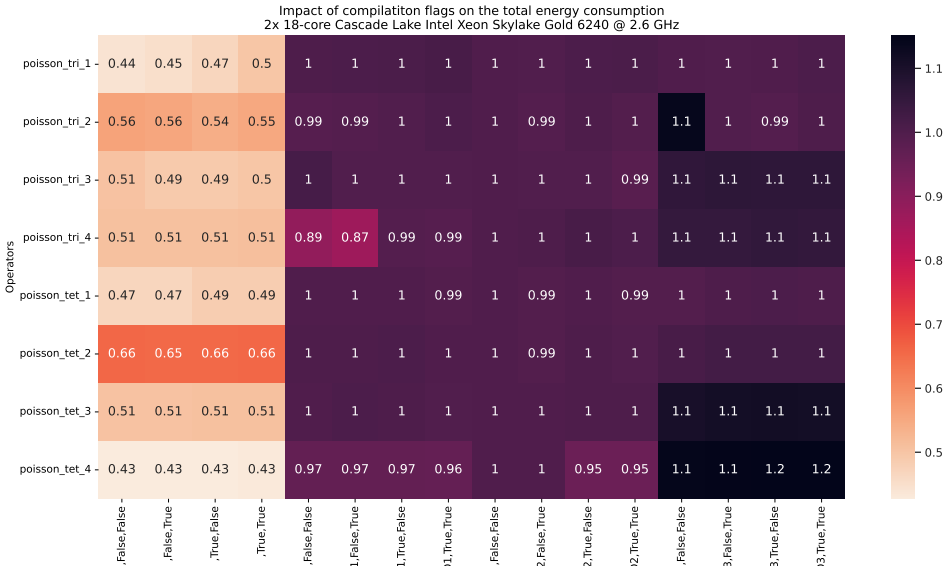


Figure 4.13: Total energy loss factor depending on compilation flags (Cache and core + DRAM). The combination of reference is the O2 optimization without fastmath or fassociativemath.

As we could expect, Figure 4.13 shows that the energy consumption clearly depends on the level of optimization selected at compilation time. We can see that the consumption can be raised by 10% to 20% with `O3` for some kernels.

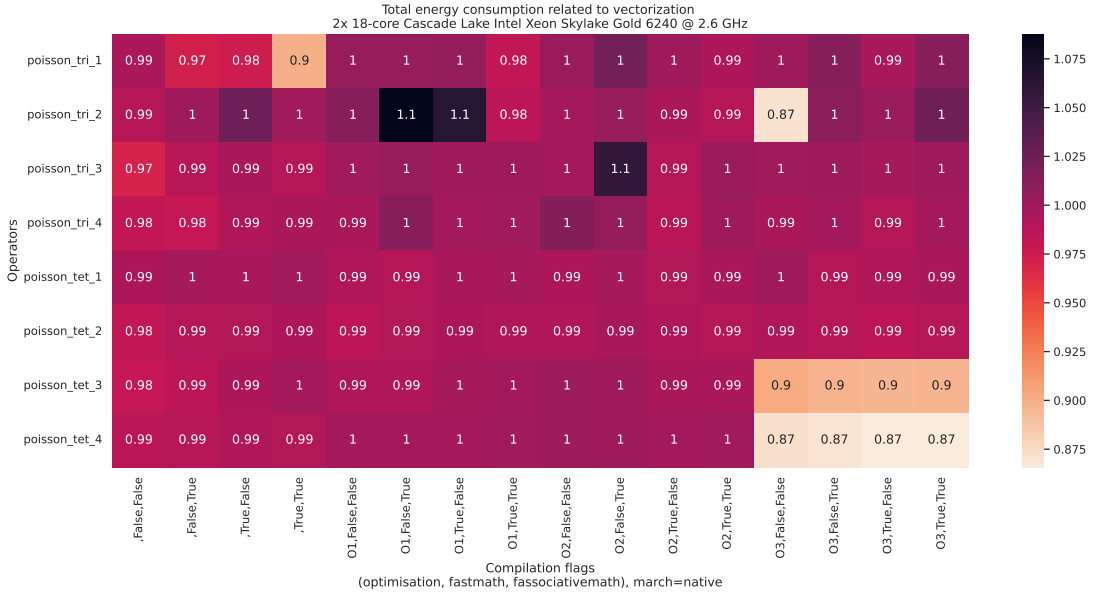


Figure 4.14: Energy loss factor related to auto-vectorization depending on compilation flags

One more time, Figure 4.14 shows that energy consumption is lowered by the use of SIMD instructions.

To put it in a nutshell, SIMD instructions should provide better performance but also less energy consumption.

4.3 Manual Vectorization

As a final evaluation, this section is dedicated to the manual vectorization of 3 sets of integral functions respectively used during the computation of `poisson_tri_1`, `poisson_tri_2`, and `poisson_tri_3`. In order to check the correction of the computations with the AVX2 intel intrinsics, the following testing system has been set up.

4.3.1 Tests

```
double* poisson_tri_1_integral_cell_0_ref(void){
    // Define the reference results
    double table[9] = {1, -0.5, -0.5, -0.5, 0.5, 0, -0.5, 0, 0.5};
    // Allocate space for a matrix
    double* A_ref = (double*)calloc(9, sizeof(double));
    // Copy elements into the matrix
    memcpy(A_ref, table, 9*sizeof(double));
    return A_ref;
}
```

Figure 4.15: Function returning the reference result for the execution of an integral with trivial inputs.

```
#define EPSILON 0.000001
bool poisson_tri_1_integral_cell_0(void)
{
    // Input declaration
    double* A = (double*)calloc(9, sizeof(double));
    ...
    // Computes the result in A
    integral_207b4bdc.tabulate_tensor_float64(A, ...);
    // Define A_ref here
    double* A_ref = poisson_tet_1_integral_cell_0_ref();
    for(int i = 0; i < 9; i++){
        if(fabs(A[i] - A_ref[i]) >= EPSILON)
            return false;
    }
    return true;
}
```

Figure 4.16: Code for testing the correction of the modified integral functions with SIMD instructions.

The function from Figure 4.15 is a way to store the resulting values from the execution of an integral function with some given inputs. The method presented in Figure 4.16 provides a way to test whether the values returned by the function are correct or not. It is about comparing the result of the current execution (integral function modified with SIMD instructions in our case) with the reference result previously stored. Because we are manipulating floating numbers of double precision, we cannot expect the exact same result after each execution so we need to introduce some uncertainty tolerance which is the `EPSILON` value equal to 10^{-6}.

To make things more convenient, I have been modifying `FFCx` to be able to automatically generate the testing code from Figure 4.16 permitting to test each integral function from a kernel. This feature has been added by the implementation of a new argument, enabling to execute `FFCx` as follows to generate the tests:

```
ffcx kernel.py --test
```

4.3.2 Performance

Local Assembly

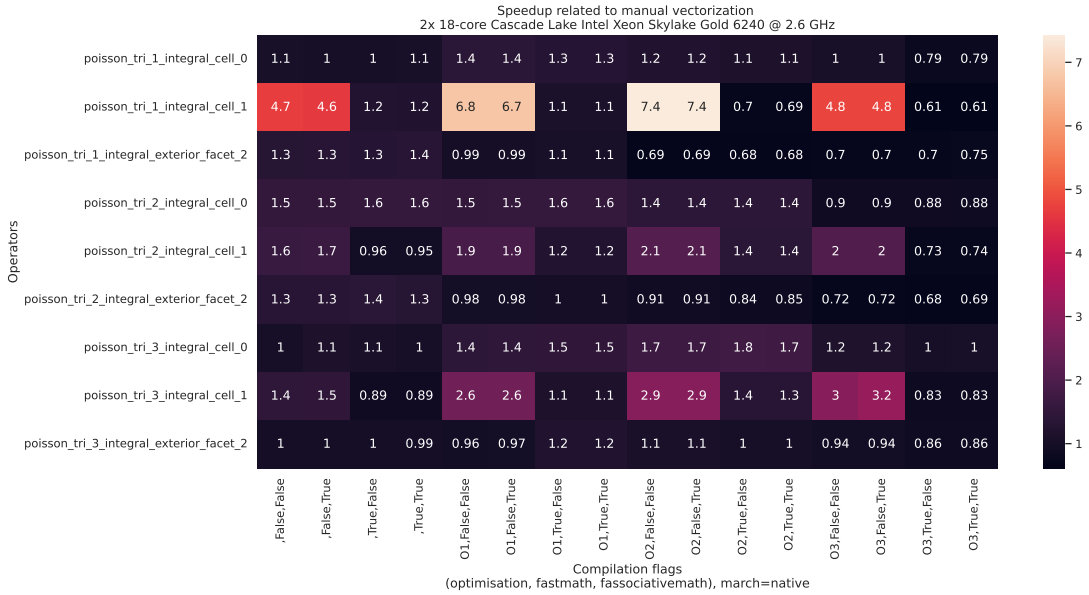


Figure 4.17: Speedup related to AVX2 manual vectorization depending on the combination of compilation flags. Each value is computed the following way: $\frac{timeWithAutoVectorization}{timeWithManualVectorization}$. The greater the number, the more performance improvement is related to manual vectorization.

Figure 4.17 shows that the difference is much smaller when `-ffast-math` is activated. Indeed, as seen in the section 4.2 `-ffast-math` favors auto-vectorization and then performances are getting closer from the ones with manual vectorization. On the other hand, when this flag is disabled, we can see that the difference is much more significant (up to 7.4 times better) because we are comparing a fully vectorized code with a totally

sequential one, and this reminds us the importance of vector code. Also for unknown reasons, it seems that some integrals have better performance with the auto-vectorized version, especially when the `O3` optimization is activated with `-ffast-math` so we can assume that `O3` enables to make better decisions on the size of vectors whereas the manual vectorization code is arbitrarily using 256 bits vectors.

Let us now explore the impact of manual vectorization on the global kernels.

Global Assembly

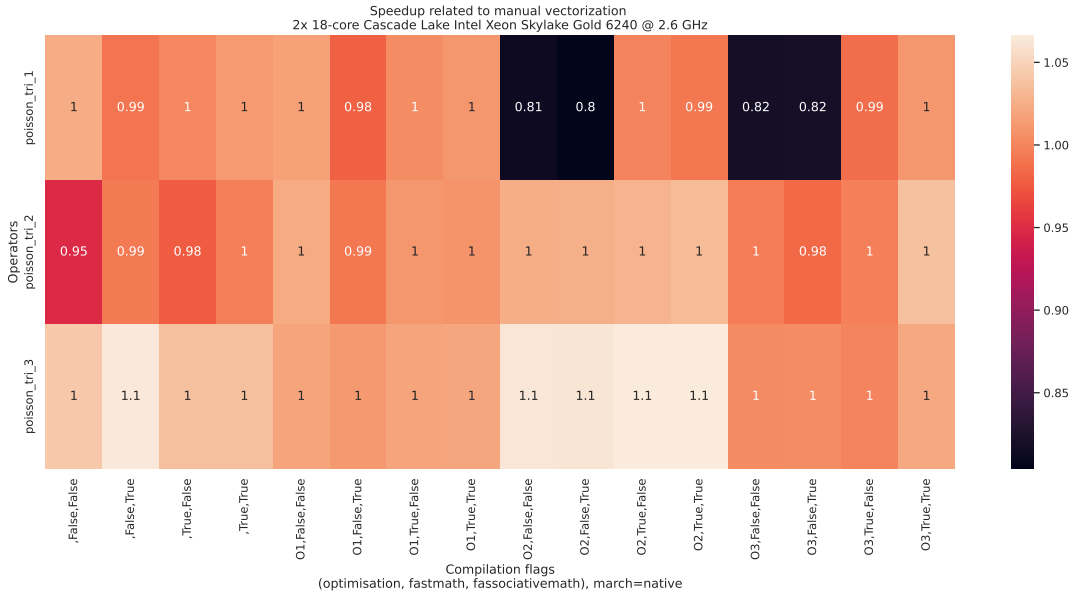


Figure 4.18: Speedup related to AVX2 manual vectorization depending on the combination of compilation flags. Each value is computed the following way: $\frac{timeWithAutoVectorization}{timeWithManualVectorization}$. The greater the number, the more performance improvement is related to manual vectorization.

Finally, Figure 4.18 shows that manual vectorization does not really improve the performance on both first kernels because their complexity is very low. However, `poisson_tri_3` shows a performance improvement of 10% with `O2` which is very promising for more complex kernels with higher compute intensity such as `poisson_tri_4` or even 3D ones.

Chapter 5

Conclusion

In a nutshell, this internship has been an excellent and rewarding experience. During the discovery of the computing platform `FEniCSx`, I had to find a way to make each components work together to get the ability to edit, compile and execute the code. It was an important step, during which I learned a lot about installation tools and `UNIX` environments. Then I got interested in some existing scientific paper such as the paper on vectorisation in Firedrake [8] and also the ACM Transactions on Mathematical Software [9], an article to which James TROTTER contributed. Both papers were relevant in the context of my internship an then inspired me to carry out my own study.

Firstly, I developed my own microbenchmark program enabling me to make my own measurements of the performance and the energy consumption of some kernels of reference and their integral functions individually.

Afterwards, I could evaluate the importance of compilation flags, and their contribution to the generation of vector code compared to the manual use of the Intel intrinsics `SIMD` instructions. Indeed, catching the right flag combination, sometimes featuring vector code inside the integral functions, can lead to significant performance improvements on sufficiently complex kernels. Moreover, the energy consumption doesn't always increase with the performance which is a good point.

Finally, the technical aspects of the work I've done are not flawless and could be improved provided enough time. One of the opening possibilities would be to take the `FFCx` compiler modification a step further, featuring the automatic vetor code generation using the library `MIPP` [10], which is a portable and open-source wrapper for vector intrinsic functions (SIMD). It enables to use provided functions to automatically generate the right intrisic calls depending on the client specific architecture.

References

[1] FEniCSx Project. https://fenicsproject.org/. [Accessed: 2023-07-24].

[2] Martin Sandve Alnæs. *UFL: a finite element form language*, pages 303–338. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[3] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. *FFC: the FEniCS form compiler*, pages 227–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] Anders Logg, Garth N. Wells, and Johan Hake. *DOLFIN: a C++/Python finite element library*, pages 173–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[5] Wikipedia contributors. Single instruction, multiple data — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Single_instruction,_multiple_data&oldid=1166388689, 2023. [Accessed 24-07-2023].

[6] FEniCSx Project, Built-in meshes. https://fenicsproject.org/olddocs/dolfin/1.4.0/python/demo/documented/built-in_meshes/python/documentation.html. [Accessed: 2023-07-25].

[7] PlaFRIM. https://www.plafrim.fr. [Accessed: 2023-08-01].

[8] Tianjiao Sun, Lawrence Mitchell, Kaushik Kulkarni, Andreas Klöckner, David A Ham, and Paul HJ Kelly. A study of vectorization for matrix-free finite element methods. *The International Journal of High Performance Computing Applications*, 34(6):629–644, 2020.

[9] James D. Trotter, Xing Cai, and Simon W. Funke. On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core cpus. *ACM Trans. Math. Softw.*, 48(2), may 2022.

[10] MIPP. https://github.com/aff3ct/MIPP. [Accessed: 2023-08-19].