EUPEX
European Pilot for Exascale

EUPEX deliverable D3.2
Applications optimised for SVE and HBM

Document properties

Contract Number:	101033975
Contractual Deadline:	M24 (December 31, 2023)
Dissemination level:	Public
Deliverable Nature:	Report
Edited by:	Andrew Beggs, Olivier Marsden, ECMWF
Keywords:	EUPEX, HPC, Optimisation, Applications, Benchmarks, SVE, HBM
Authors:	Daniele Cesarini, Fabio Pitari, Filippo Barbari, Federico Ficarelli, Piero Lanucara: CINECA, Gabriele Cavallaro: FZJ, Emanuele Casarotti: HPC4NDR, Igor Piljić: UNIZG, Andrew Beggs, Olivier Marsden, Ioan Hadade: ECMWF, Luca Tornatore: INAF, Vasilis Flouris, Fotis Nikolaidis, Angelos Bilas: FORTH, Erwan Raffin, Antoine Morvan: Atos, Ondrej Meca, Riha Lubomir: VSB-TUO, Theophile Lohier: Cybeletech, Valérie Brenner, Luigi Genovese, Maxime Delorme: CEA
Reviewers:	Hans-Christian Hoppe, ParTec
Submission Date:	29.12.2023
Status:	Final

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101033975. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Germany, Italy, Greece, United Kingdom, Czech Republic, Croatia.

History of changes

Version	Date	Status	Contributors
0.1	01.08.2023	Document setup	O. Marsden start
0.2	03.10.2023	Shared overleaf	O. Meca & O. Marsden
0.5	31.10.2023	Preliminary content deadline	All
0.7	29.11.2023	Second draft deadline	All
0.9	10.12.2023	Internal review	Hans-Christian Hoppe
0.95	15.12.2023	Internal review comments addressed	All
1.0	29.12.2023	Final deliverable prepared	ECMWF

Contents

History of changes 1

List of Figures 3

List of Tables 5

Executive Summary 6

1 Introduction 7
- 1.1 Irene - Fujitsu A64FX . . . 7
- 1.2 SVE . . . 8
- 1.3 HBM . . . 9

2 Tailoring EUPEX Benchmark Applications to SVE and HBM 10
- 2.1 IFS Weather Forecasting Suite . . . 10
- 2.2 Cybeletech - Agriculture . . . 20
- 2.3 Artificial Intelligence for Earth Observation - AI4EO . . . 28
- 2.4 SPECFEM3D . . . 38
- 2.5 ESPRESO FEM . . . 41
- 2.6 LiGen . . . 52
- 2.7 BIGDFT . . . 59
- 2.8 OpenGADGET . . . 65

3 Tailoring EUPEX Mini-applications To SVE and HBM 84
- 3.1 Bolt65 . . . 84
- 3.2 Dyablo - Whole Sun . . . 98

4 Summary 112

List of Acronyms and Abbreviations 114

Bibliography 117

5 Appendix 121
- 5.1 Appendix: IFS . . . 121

List of Figures

1 **IFS-CloudSC** : effect of auto-vectorisation . . . 11
2 **IFS-CloudSC** : performance of different implementations of CloudSC kernel . . . 12
3 **IFS-CloudSC** : cost of handwritten SVE intrinsic implementation . . . 13
4 **IFS-CloudSC** : cost of compiler-vectorised implementation . . . 13
5 **IFS-CloudSC** : effect of HBM on CloudSC . . . 15
6 **IFS-CloudSC** : roofline plot on A64FX . . . 16
7 **IFS** : effect of HBM on IFS performance . . . 17
8 **IFS** : effect of HBM on four IFS components . . . 18
9 **CySim**: parametric estimation workflow . . . 20
10 **CySim**: workflow RAM consumption on ARM . . . 24
11 **CySim**: workflow RAM consumption on X86 . . . 24
12 **CySim**: runtime of workflow contributions - 4 nodes . . . 24
13 **CySim**: runtime of workflow contributions - 10 nodes . . . 24
14 **CySim**: runtime of workflow contributions - 24 nodes . . . 25
15 **CySim**: speed ratio correlations table . . . 25
16 **CySim**: speed ratio correlations plot . . . 26
18 **AI4EO** : Schematic overview of HPDBSCAN . . . 29
19 **AI4EO** : memory layout . . . 31
20 **AI4EO** : single node performance . . . 35
21 **AI4EO** : MPI scalability . . . 35
22 **AI4EO** : hybrid scalability . . . 35
23 **AI4EO** : MPI scalability . . . 35
24 **AI4EO** : A64FX single-core performance . . . 36
25 **AI4EO** : A64FX single-node performance . . . 36
26 **AI4EO** : energy consumption after optimisation . . . 36
27 **SPECFEM3D** : HBM usage . . . 39
28 **ESPRESO FEM** : 2D kernel assembly time . . . 48
29 **ESPRESO FEM** : 3D linear element kernel assembly time . . . 48
30 **ESPRESO FEM** : 3D quadratic element kernel assembly time . . . 49
31 **LiGen** : flamegraph plot . . . 53
32 **LiGen** : roofline graph . . . 56
33 **LiGen** : application throughput (MFLOPs) . . . 57
34 **LiGen** : application throughput (ligands/s) . . . 57
35 **LiGen** : optimisation of vectorisation . . . 57
36 **LiGen** : performance improvements from optimisation efforts . . . 57
37 **LiGen** : effects of optimisation on memory bandwidth . . . 57
38 **LiGen** : before-after 1 . . . 58
39 **LiGen** : before-after 2 . . . 58
40 **BigDFT** : workflow overview . . . 60
41 **BigDFT** : example kernel in CUDA and SyCL . . . 61
42 **BigDFT** : Fock mini-app with free boundary conditions . . . 62
43 **BigDFT** : Fock mini-app compute time with periodic boundary conditions . . . 63
44 **OpenGADGET** : roofline graphs . . . 69

45 **OpenGADGET** : application phases . . . 71
46 **OpenGADGET** : vectorisation ratio . . . 78
47 **OpenGADGET** : cache usage . . . 79
48 **OpenGADGET** : IPC . . . 80
49 **OpenGADGET** : effect of compiler . . . 82

50 **Bolt65** : transcoder schematic . . . 86
51 **Bolt65** : performance comparison . . . 91
52 **Bolt65** : effect of compiler . . . 93
53 **Bolt65** : instruction count . . . 94
54 **Bolt65** : vectorisation ration . . . 94
55 **Bolt65** : profiling results . . . 95
56 **Bolt65** : profiling results memory . . . 96
57 **Dyablo** : numerical accuracy . . . 103
58 **Dyablo** : hardware binding . . . 107
59 **Dyablo** : Intel Xeon Max topology . . . 110

List of Tables

1 Computational power (MFLOPS) and Instructions Per Cycle (IPC) comparisons between OPT and DEFAULT version on 4 nodes each . . . 39
2 List of tested elements with their number of nodes, Gauss Points (GPs), and abbreviations 47
3 Comparison of implicit and explicit dual operator F for 2D examples . . . 50
4 Comparison of implicit and explicit dual operator F for 3D examples . . . 50
5 Comparison of Intel MKL and SuiteSparse on Karolina . . . 51
6 Fock mini-app L3-cache and HBM bandwidths . . . 64
7 Stall ratios . . . 70
8 The definition of adopted metrics as functions of architectural events on ARM64FX . . 73
9 The definition of adopted metrics as functions of architectural events on ARM64FX [continues from the previous table] . . . 74
10 The definition of adopted metrics as functions of architectural events on ARM64FX [continues from the previous table] . . . 74
11 The architectural events on ARM64FX relevant to determine the vectorization . . . 76
12 The options used to tune the compilation. . . . 77

13 Test video sequences . . . 89
14 Implementation parameters . . . 90
15 Speedups with armclang compiler . . . 92
16 Speedups with FCC compiler . . . 92
17 Overall memory volume [GBytes] . . . 97
18 Overall memory bandwidth [MBytes/s] . . . 98
19 Dyablo runtime for the shorter test case using the different configurations . . . 102
20 Dyablo runtime for the full test case using the different configurations . . . 103
21 Dyablo runtime for the shorter test case . . . 105
22 Dyablo runtime for the shorter test case, with different build flags on the different configurations . . . 106
23 Dyablo runtime for the shorter test case, depending on the distribution scheme. The arrow points to the worse execution time, against which the performance gain is calculated. . . . 109
24 Dyablo runtime for the shorter test case, depending on the memory allocation scheme, on the Intel Xeon Max 9480 system. . . . 111

Executive Summary

This document reports activities accomplished in WP3 during the second year of the project. The work done in this document has been carried out over the period from M12 to M24, and it is focused on the optimisation of the applications that constitute the EUPEX benchmark suite.

In particular, the work has tailored applications to make good use of the two main hardware specificities which the EUPEX pilot platform should expose to end developers: the Scalable Vector Extension (SVE) instruction set, and High Bandwidth Memory (HBM). Both of these hardware features are still unusual for CPUs, and the Fujitsu A64FX platform is to date the only platform to combine both of these characteristics. Accordingly, the A64FX partition of the Irène supercomputer hosted by the CEA in France has been used for a large fraction of this optimisation work.

As described in the DoA part B, the objectives of this deliverable are the optimisation of EUPEX use cases for SVE and HBM, with a particular focus on portability, scalability, and accuracy. Porting strategies for targeting the novel hardware features are described for each application use case. Benchmark results highlighting the effect of SVE and HBM usage are also presented and discussed.

Lessons learned from this exercise by each partner have been distilled in the conclusions, and will provide useful insights for the community of applications that will be hoping to target the future JUPITER Exascale-class European supercomputer, which will feature EPI hardware.

1 Introduction

The EUPEX project is breaking ground in demonstrating a number of new technologies deployed together in an HPC environment. It is therefore important that project applications demonstrate these technologies to be usable by, and useful to, a range of scientific applications representative of workloads that might be encountered in future European HPC centres.

1.1 Irene - Fujitsu A64FX

Based on Fujitsu PRIMEHPC FX700 technology, the A64FX partition of the Irene system, hosted by TGCC-CEA, is used as a proxy system in the EUPEX project. It is made up of 10 chassis, each hosting 8 air-cooled computational nodes for a total of 80 nodes. These are integrated into GENCI's Joliot-Curie supercomputer, which is a machine dedicated to European academic and industrial open research.

The A64FX partition gives scientists the opportunity to port their applications, and prepare for the future European processor by leveraging the unique, HPC focused features of the A64FX processor. These features include the SVE vector instruction set and the use of HBM2 fast access memory. Each node contains [1][2]:

- 1x 48-core Fujitsu A64FX ARM processor clocked at 1.8 GHz
- 2.7648 DP TFlops / 5.5296 SP TFlops @ 1.8 GHz
- 32 GB HBM2 memory (1024 GB/s) out of which 24 GB is dedicated to the user
- 1x 512GB NVMe disk
- NVIDIA HDR100 Infiniband (12 GB/s)

The Irene system provides a large variety of compiler stacks exposed via the module system such as Fujitsu, ARM, GNU, and NVIDIA as well as OpenMPI with UCX for MPI. Some examples of complete toolchains that are available on Irene are:

- fujitsu/1.2.0 + openmpi/4.0.5 + SSL
- arm-compiler/21.0.0 + openmpi/4.0.5 + armpl (ARM performance libraries)
- gcc/11.1.0 + openmpi/4.0.5

For application profiling, a number of tools are made available such as:

- Fujitsu FIPP/FAPP [3]
- ARM Forge MAP (now called Linaro MAP) [4]
- Linux perf with ability to set paranoid levels to capture CPU events
- In the context of the EUPEX project, ScoreP has been ported to the platform [5]

1.2 SVE

SVE, or Scalable Vector Extension[6], is a SIMD ISA developed by Arm, in part, to allow for vector code to scale with increasing hardware capabilities. This is done by making SIMD instructions vector length agnostic, and instead determining this at runtime. The hardware vector length can vary from 128 bits up to 2048 bits, in 128-bit increments. This allows developers to write code specialised for SVE while being assured that the same code will run on future machines - allowing for more time to be invested into optimising future proofed code.

This idea is a first in terms of SIMD ISAs, where, for example, code optimised and compiled for the Intel AVX2 instruction set would need to be re-optimized and recompiled in order to make use of the AVX512 instruction set available on more recent platforms.

To date, this technology has been adopted by Fujitsu's A64FX, AWS' Graviton3 processors, and most recently, the NVIDIA Grace processor. Interestingly, the A64FX processor has a single 512 bit vector length, Graviton3 has 2x256bit vectors, and Grace has 4x128bit vectors.

The three most straightforward ways for software to target SVE are as follows :

- Use an SVE-enabled library provided by the ARM compiler environment or by the CPU designer
- Make use of auto-vectorization capabilities of a compiler, be it via compiler-specific flags and directives, or OpenMP SIMD directives
- Employ SVE intrinsics or assembler instructions to mandate explicit SVE usage

In applications that make use of algorithmic elements present in mathematical libraries, making use of SVE-enabled libraries is the first thing to be done. Examples of elements that can be found in such libraries include FFTs, dense and sparse linear algebra operations, sorting operations, and many more. This might be as easy as targeting the right optimised library at build time, for software already relying on a mathematical library to provide algorithmic building blocks, or modifying the code to rely on a library call instead of a bespoke in-code implementation of the algorithmic element. In codes where such building blocks make up a significant fraction of execution time, a large performance uplift can be achieved.

The second approach leverages the compiler to do the heavy lifting of targeting the SVE instruction set. All major compilers support some form of auto-vectorisation, in which loop constructs applying identical instructions to multiple addresses are replaced by a single vector instruction. Simple loops free of predicates and data dependencies are easily vectorised by compilers. However, more complex loops might require some code reorganisation and/or annotation with compiler directives to achieve vectorisation. The use of compiler-based automatic vectorisation is critical in achieving good performance in code that does not have algorithmic hotspots that can be handled by library calls and must be handled on an individual basis. Areas of code that do not vectorise well can often be reorganised or annotated with compiler directives, in order to improve auto-vectorisation by giving the compiler information that can only be guaranteed by the programmer. Compiler directives related to vectorisation can be split into vendor-specific directives (see the Fujitsu compiler guides [7][8] for Fujitsu compiler examples) and more general directives such as the OpenMP SIMD directives, which are more portable.

Finally, SVE intrinsic functions may be inserted directly into code. For non-library-acceleratable code, this approach offers maximum performance potential, but is also the most invasive and time consuming to implement. Platform intrinsics such as SVE are C-style functions, and cannot be used as is without a custom interface from Fortran code. Intrinsics-based code tends also to be more time-consuming to read, due to its assembly like appearance, and is less platform-portable, since vector intrinsics are platform-specific.

It is also possible to resort to SVE assembly, but this is not typically an approach end-user developers might wish to follow.

1.3 HBM

In the world of HPC, HBM, or High-Bandwidth Memory, is an interface for SDRAM that saw a first hardware implementation in 2013 (HBM1 [9]). It was competing at the time with an alternative interface named HMC (Hybrid Memory Cube), but has since then entirely supplanted it for high-bandwidth applications.

The HBM interface aims to allow higher bandwidth than (G)DDR of the same generation, while reducing power consumption. These advantages do come with a strong downside, since HBM memory is phsically attached to the device at manufacturing time, meaning memory size is not a configuration option but rather a chip skew. This last point is an issue for CPUs more than for GPUs, since GDDR memory used on GPUs is also soldered in place at assembly time. HBM memory is also more expensive per byte than DDR memory. The interface has gone through 3 major revisions, with bandwidth more than doubling at each revision, and maximum capacity also increasing thanks to an increase in the number of stacked layers.

Thanks to its industry-leading bandwidth, HBM has been used on top-of-the-range GPUs for most of the last decade. Despite its bandwidth advantage, HBM's use for CPUs has not been widespread, to date. Notable exceptions are the the NEC Aurora vector CPU family, and the Fujitsu A64FX. Most recently Intel has released the Xeon MAX versions of the "Sapphire Rapids" CPU generation featuring a combination of HBM and DDR memory.

From an application's standpoint, the main aspect to consider is the limited size of HBM. Here a critical difference appears between the EUPEX software development platform, the A64FX-based Irene machine hosted at the CEA, and the future EUPEX prototype machine. The A64FX-based system has a limited amount of HBM2 memory per node – 32 GB, of which 24GB is usable by an application – and no node-level DDR-based memory. The EUPEX prototype, on the other hand, will have a much larger amount of HBM3 memory per socket, and will also have DDR memory on the motherboard. On Irene, the total application resident memory must fit into the HBM. This is restrictive on application design and problem size. With the presence of DDR-based memory on the EUPEX prototype, the role of the HBM can be different (e.g., a very large user-managed cache), as it is not required that the entire application resident memory reside in HBM. The absence of a system-managed cache mode for HBM handling on the Rhea-based EUPEX prototype make it different from the only other CPU platform which offers a combination of HBM- and DDR- based memory. The practical implications of this difference will only become fully clear once application work on the Rhea-based EUPEX platform hardware commences.

2 Tailoring EUPEX Benchmark Applications to SVE and HBM

In this section, we detail the work done on the application workflows constituting the EUPEX benchmark suite in order to make use of the Scalable Vector Extension (SVE) instruction set and the High Bandwidth Memory (HBM) features that will be present on the EUPEX prototype.

2.1 IFS Weather Forecasting Suite

The Integrated Forecasting System (IFS) is ECMWF's flagship weather prediction software suite. It is a Numerical Weather Prediction (NWP) program that is used to create a variety of forecast products for Member and Co-operating States. A detailed description of the IFS can be found in EUPEX deliverable D3.1[10].

The IFS is an extremely varied codebase due to the truly global scale of the problem space. In light of this we have chosen a suitable subprogram, internally called a dwarf, named CloudSC to evaluate SVE and HBM. CloudSC is a cloud microphysics parameterisation scheme, which can be easily scaled to stress even the most advanced technologies.

2.1.1 CloudSC Dwarf Results

Using C SVE intrinsics in Fortran

SVE intrinsics allow for explicit and hand optimised vectorisation. This allows for potentially higher performance than what the compiler would be capable of, if extra knowledge about the code is necessary for vectorisation. It also ensures vectorisation is used - even if the compiler can't auto-vectorise code.

The IFS, and by extension CloudSC, is written almost entirely in Fortran, and it is not feasible to translate the whole codebase due to its size and complexity. Unfortunately SVE intrinsics are only available in C, and this is not something that can be easily changed.

The solution is to identify hot sections of the codebase and hand write SVE kernels for these in C. These kernels can then be called from Fortran, as it supports the C ABI.

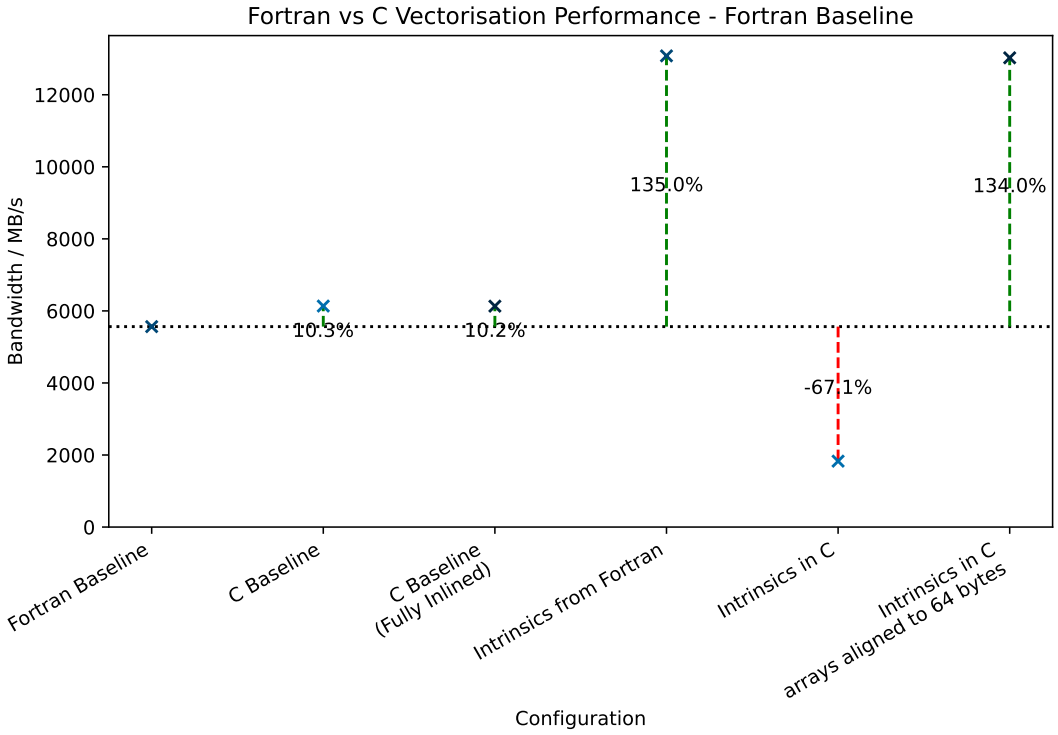


Figure 1: Performance of various kernels against an auto-vectorised Fortran kernel baseline

Figure 1 shows the results of an experiment designed to measure any impacts of cross language compilation. This experiment used three kernels, all identical in function, one written in Fortran, one in C, and another in SVE intrinsics. These kernels were then called from either a Fortran or C driver, which contained the loop headers - much like what would be implemented in CloudSC.

In these simple examples, it was possible to beat the auto-vectorised code. This can be seen with the two intrinsics kernels beating the Fortran baseline by 135% and 134% for Fortran and aligned C drivers respectively. However, initial result for the C kernel showed a significant 67.1% slowdown compared to the Fortran baseline. Further analysis revealed that this was due to memory alignment issues. The Fujitsu Fortran compiler automatically aligns to 64 byte boundaries by default, but the Fujitsu C compiler instead chooses 16 byte boundaries. This was identified by Fujitsu's FAPP tool [3] showing a higher "stream mode prefetch rate" for the Fortran handler configuration. This was easily fixed using `aligned_alloc`.

Experiments were also done comparing inlined and non-inlined code, to asses the impact of function calls between driver and kernel, but are not represented in Figure 1 as the difference was negligible.

In conclusion, not only is it possible for SVE intrinsics to be used in Fortran via C, but it can also out-perform the compiler - although care must be taken to accomodate subtle compiler differences.

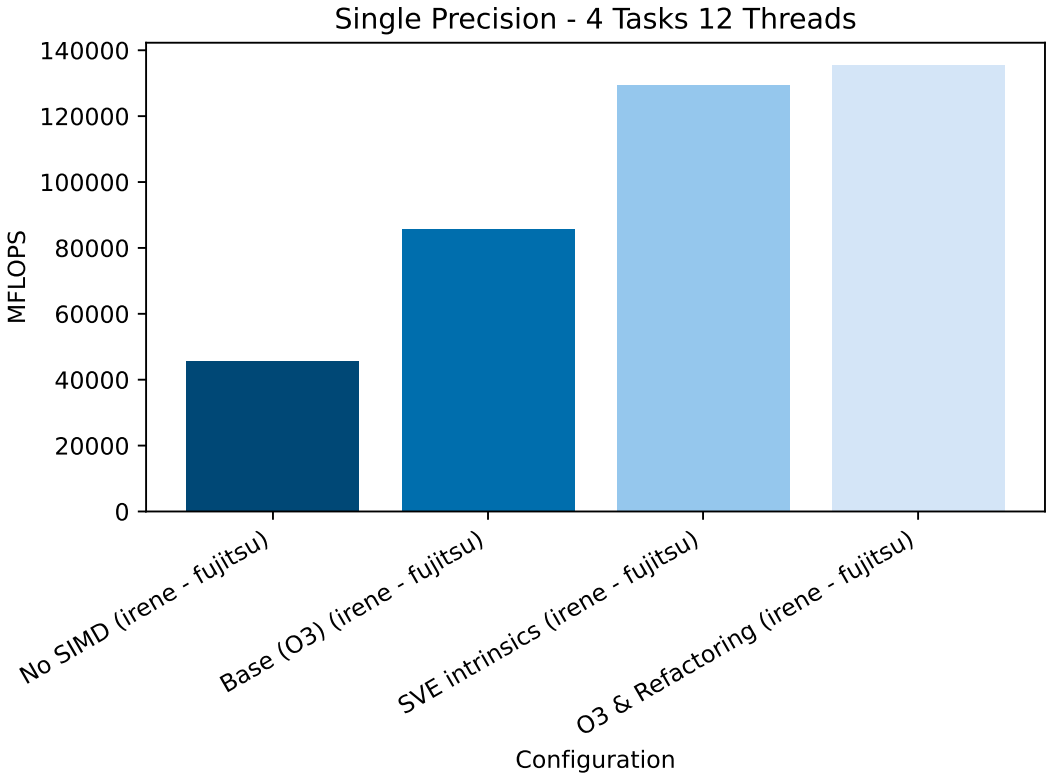


Figure 2: Performance of CloudSC without vectorisation, with auto-vectorisation, a handwritten SVE kernel, & auto-vectorised refactored kernel

Effects of Vectorisation on Performance

By just using the compiler with sensible flags, *e.g.* `-KA64FX, -KSVE, -KARMV8_3_A, -Ksimd=2`, *etc.*, auto-vectorisation is able to increase performance by 87.6% from 45.7 GFLOPS to 85.8 GFLOPS - an expected and easy result. However, the A64FX processor is capable of much higher performance, which cannot be reached through compiler flags alone.

As can be seen in Figure 2, refactoring the hottest loop within CloudSC yields an increase in throughput of 57.9% over base performance. The change was minimal, less than 10 lines, and involved changing a gather/scatter operation to contiguous load/store accesses. Gather/scatters are particularly expensive on the A64FX due to both its pipeline design and the high latency of HBM.

95.6% of this performance of the auto-vectorised code can be achieved by a hand written SVE intrinsics kernel, which can be seen in Appendix 5.1.1. Unlike the auto-vectorisation done by the compiler, the SVE intrinsics are completely portable across SVE-enabled platforms and do not rely on the compiler's ability to perform optimisations. This allows us to immediately achieve what we know is good performance, on unknown SVE-enabled platforms, which might have compilers lacking maturity.

This approach also allows us to incrementally vectorise our application, starting with the hottest loops for maximum impact.

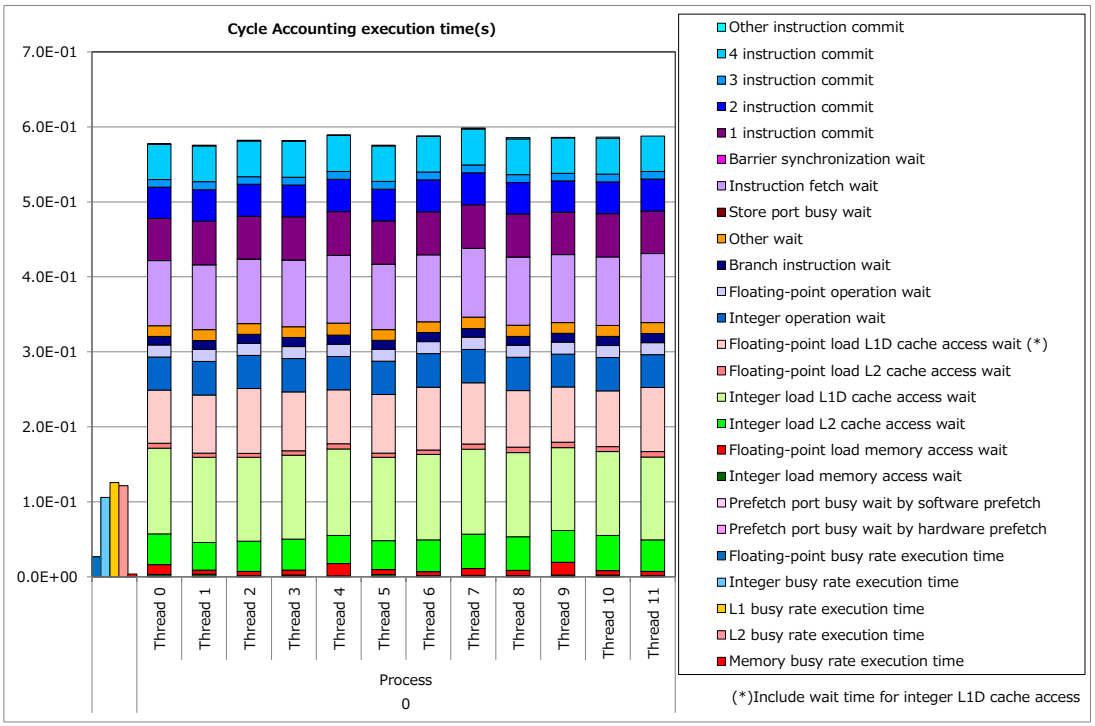


Figure 3: Cycle accounting by cores in CMG 0 from FAPP, for handwritten SVE intrinsics

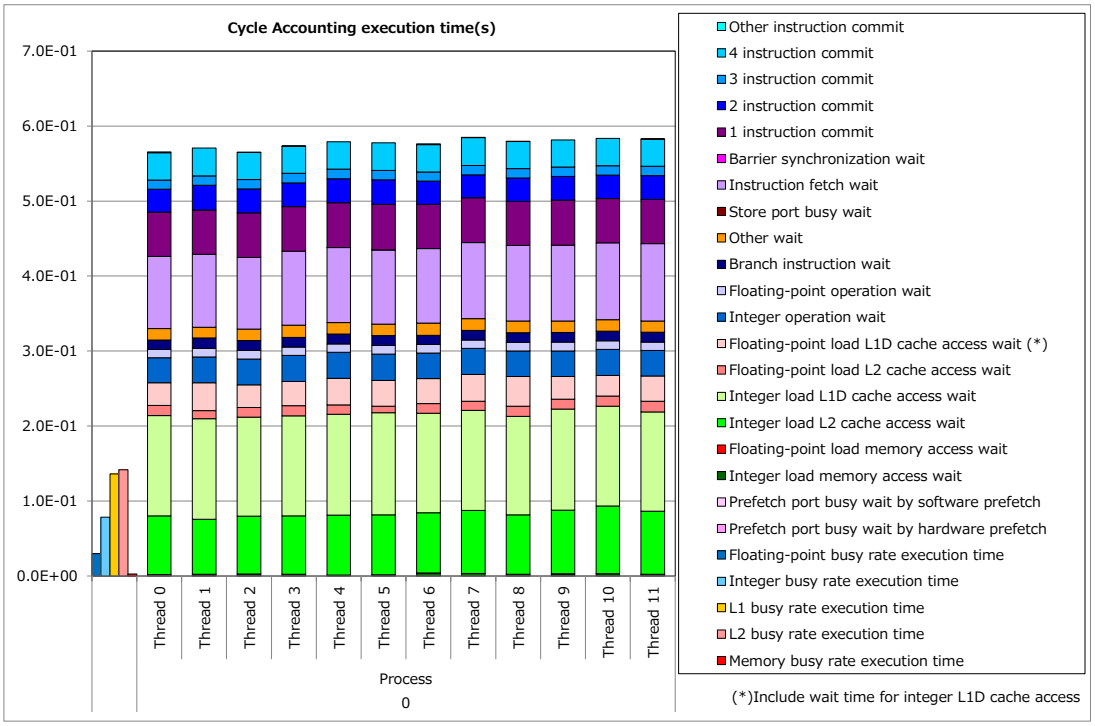


Figure 4: Cycle accounting by cores in CMG 0 from FAPP, for compiler vectorised code

As seen in Figure 2, a hand written SVE intrinsics kernel written in C and called from Fortran, achieves 95.6% of performance of an auto-vectorised loop in CloudSC. This kernel also does not use any memory prefetching optimisations, which could easily account for this missing 5% of performance. This is reinforced by Figures 3 and 4, as Figure 3 (handwritten SVE intrinsics kernel) has more cycles spent on FP L1D cache waits and FP memory access waits, than Figure 4 representing auto-vectorised code.

There are, however, some drawbacks to hand-written SVE code. The first is that by starting with the hottest loop and working to the coldest, returns will be diminishing. Although, this does allow developers to decide when to stop manually vectorising the code, as it is obvious when subsequent vectorisation efforts will not provide a worthwhile performance increase.

Another drawback is that the code is, by design, vectorised in a piecemeal manner. This means that the code isn't optimised across loops, which could result in unrealised potential performance. A solution to this could be to perform these inter-loop optimisations once a satisfactory amount of vectorisation has been achieved, and therefore when it is unlikely for the vectorised code to change much.

It's also worth noting that when explicit intrinsics have been used, the compiler is constrained in the optimisations it can apply to the intrinsics. While it is still allowed to interpret the intrinsics as it sees best, by being more explicit, we have deprived the compiler of its own interpretation of the original code. This has the drawback of preventing future, possibly superior, compilers from optimising the original code. However, this is easily mitigated by benchmarking new compiler versions with and without the explicit intrinsics enabled, which we recommend should be contained within an `#ifdef` statement.

Impact of Memory Bandwidth on CloudSC

CloudSC is known to be compute intensive and often not bound by memory bandwidth, as shown in Figure 6. However, a more empirical metric of this would be desirable.

The following procedure provides this. Performance is modelled with a naïve equation, which ignores I/O and communication performance:

$$T_1 = T_0\alpha\left(\frac{f_0}{f_1}\right) + T_0\beta\left(\frac{BW_0}{BW_1}\right)$$

Where:

- T_0 & T_1: Are some performance metric from two separate runs
- α: Compute dependency (α = 1, purely compute bound)
- f_0 & f_1: The CPU frequencies used on the two runs
- β: Memory bandwidth dependency (β = 1, purely memory bound)
- BW_0 & BW_1: The memory bandwidths used on the two runs

This equation is similar to Amdahl's law, but instead of splitting the application into sequential and parallel sections, it splits it into compute bound and memory bound. As we cannot change the memory bandwidth, we assume the ratio between runs to be 1. However, we can change the CPU frequency, and so runs are taken at various frequencies. We then plot the following rearranged equation:

$$\frac{T_1}{T_0} = \alpha \left(\frac{f_0}{f_1} \right) + \beta$$

α and β can then be easily calculated with a linear regression. These were α = 0.651 & β = 0.343 for CloudSC, indicating that it is heavily compute bound.

However, a portion of CloudSC is still memory bound, so while its performance increases might be less significant than those from SVE, HBM does provide some benefit. Unfortunately we cannot quantify this on Irene as there is no way to toggle the HBM on and off.

However, using an Intel Sapphire Rapids Max based machine, Figure 5 could be obtained. While there certainly are architectural differences to ARM based systems, it illustrates well that CloudSC can see some benefits from HBM - even if it's mostly compute bound.

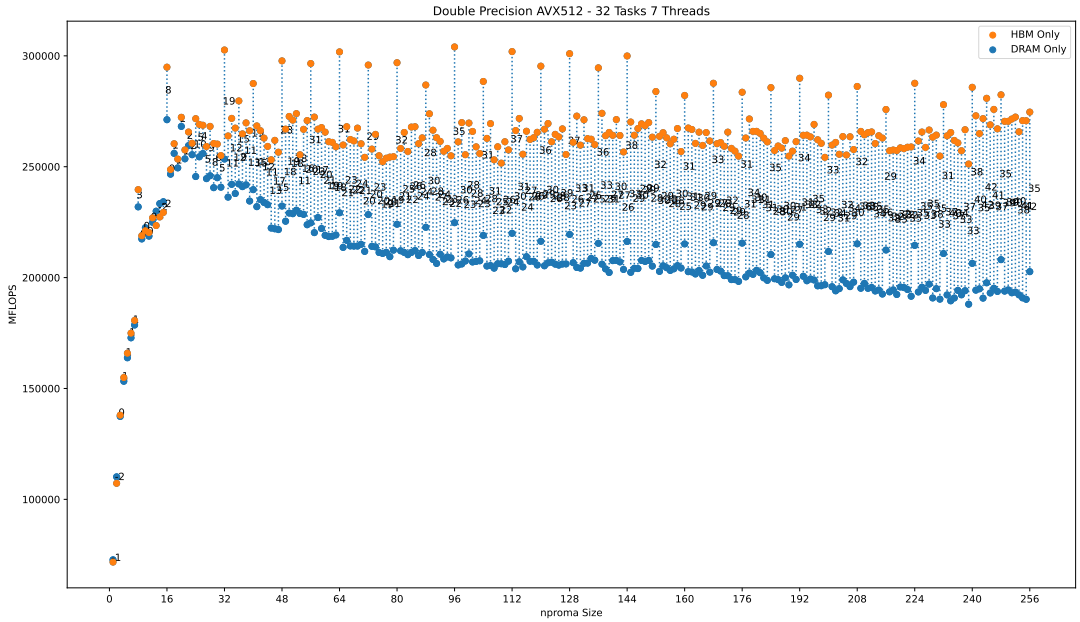


Figure 5: Performance of CloudSC with increasing nproma size (analogous to blocking size for the compute loop), HBM only vs DRAM only. The text is percentage performance gained with HBM

Effects of HBM Memory on Full IFS

As described previously, the CloudSC mini-app is heavily CPU-frequency bound, and as a consequence benefits comparatively little from the very large memory bandwidth that HBM provides.

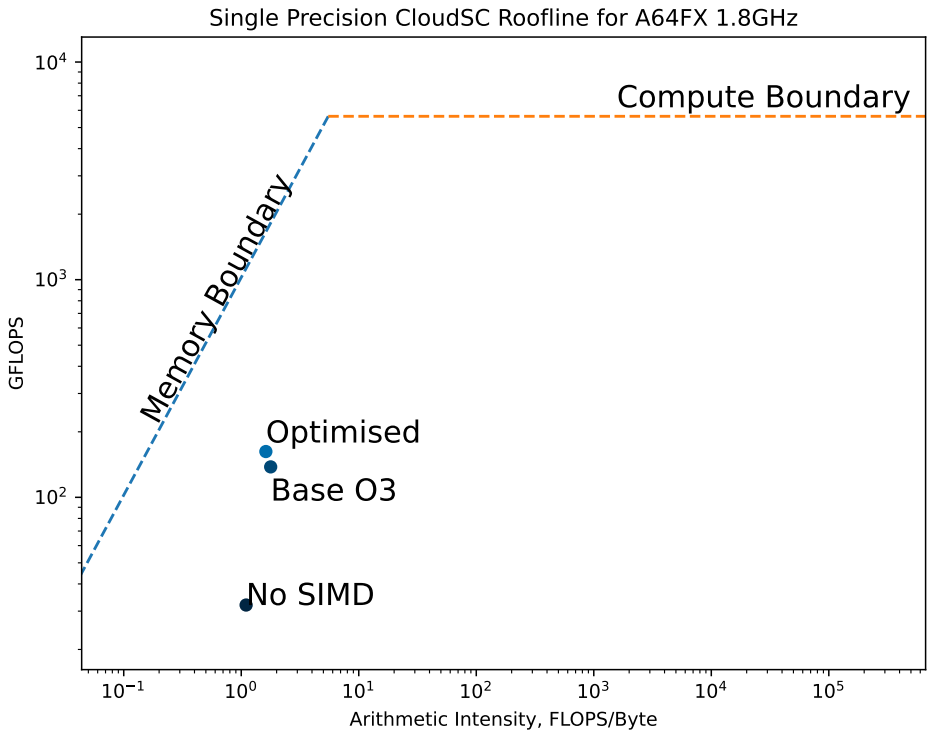


Figure 6: Single precision roofline graph for the A64FX processor of CloudSC with various optimisations applied. Theoretical bounds used

However, the IFS contains many components that have very different computational patterns and memory traffic from the CloudSC code. The effect of HBM on IFS performance has therefore been analysed. In order to be able to do a fair comparison against a baseline in which memory type is the only difference, a platform exposing both HBM and DDR must be used. To this end, a dual-socket Intel Xeon Max (Sapphire Rapids) node was used for this work. The CPU was a Xeon Max 9580, with 56 cores clocked at a base frequency of 1.9 GHz with a maximum turbo frequency of 3.5 GHz, and 64GB of HBM. On top of the HBM, each socket had 256GB of DDR5 RAM with 4400 MT/s.

The IFS was run across both CPU sockets, on a TcO79 137-level grid, in a configuration of 16 (MPI) tasks x 7 (OpenMP) threads. It was run first in DDR-only mode, with HBM disabled, and then with HBM being used preferentially, via the use of the numactl utility, *i.e.*

```
mpirun <any additional flags> numactl --preferred-many=8-15 ./a.out
```

The `--preferred-many=8-15` here requests that NUMA regions 8-15 of the platform, namely the HBM regions of the NUMA platform, be used preferentially.

Figure 7 shows the difference in performance between the reference run using RAM, and the HBM-enabled run. Performance is measured in Forecast Days Per Day (FDPD), and plotted as a function of NPROMA, the cache-blocking innermost dimension of data structures in the IFS. HBM is seen to have a large beneficial effect on IFS performance, with more than 25% improvement to performance. Optimal cache-blocking length is also substantially higher, due to the very high bus width (1024b). To

put the FDPD numbers shown in Figure 7 into perspective, ECMWF's operational TcO1279 15 day forecasts must complete in under one hour, corresponding to a minimal operational requirement of 360 FDPD.

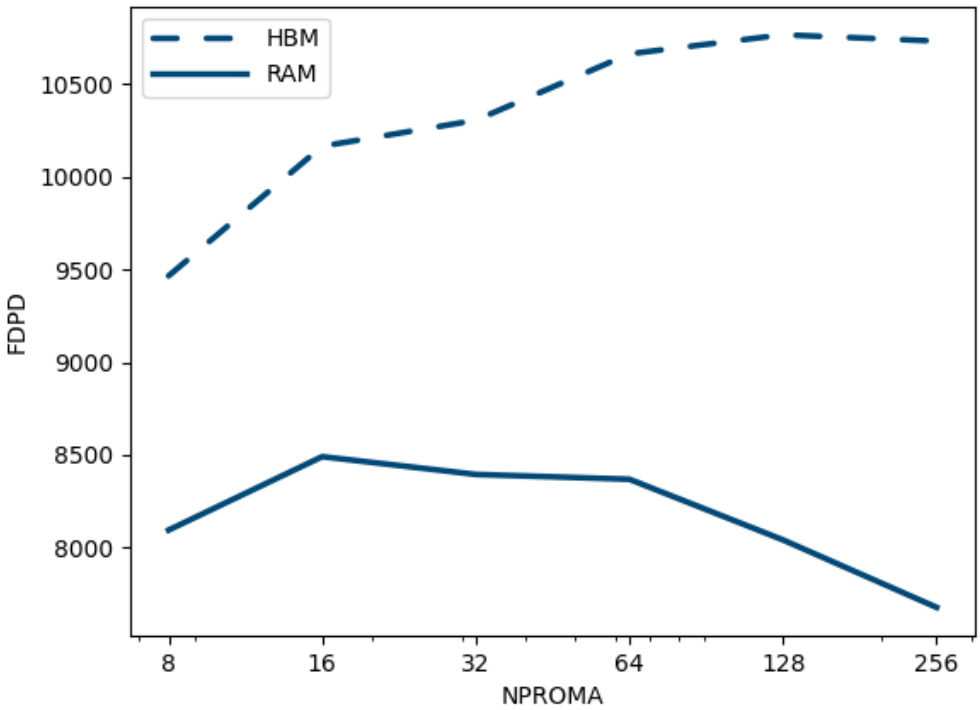


Figure 7: Forecast days per day of execution, with and without HBM. Higher is better. Grid resolution Tco79

The IFS is a large codebase, consisting of a number of algorithmic components that exhibit different coding styles and different computational patterns. It is therefore interesting to to look at the effect of HBM memory in more detail, at a finer level of granularity. Figure 8 illustrates the differences in effect of HBM on different components in an IFS timestep. It plots the time spent in 4 components (physical parameterisations, grid-point dynamics, radiation, direct Fourier transform), as a function of the NPROMA innermost cache-blocking array dimension, normalised for each component by the time spent with an NPROMA value of eight. Dashed lines correspond to the reference runs without HBM, while solid lines are obtained from runs using HBM. Different components are observed to benefit from HBM to very different degrees. One extreme is the direct Fourier transform which sees almost no improvement in wall-time from HBM, while the opposite extreme is the grid-point dynamics, which sees a 45% speed-up vs baseline for an NPROMA value of 128. This HBM-related performance benefit in the grid-point dynamics increases strongly with the NPROMA value, unlike the performance of the same component without HBM, which is seen to degrade significantly with increasing NPROMA value. Indeed for an NPROMA of 128, grid-point dynamics in the HBM run takes less than half the time taken in the run without HBM. This can be contrasted with the radiation, where performance with

HBM is about 25% better than the baseline, but independent of NPROMA, while performance without HBM degrades slightly with increasing NPROMA.

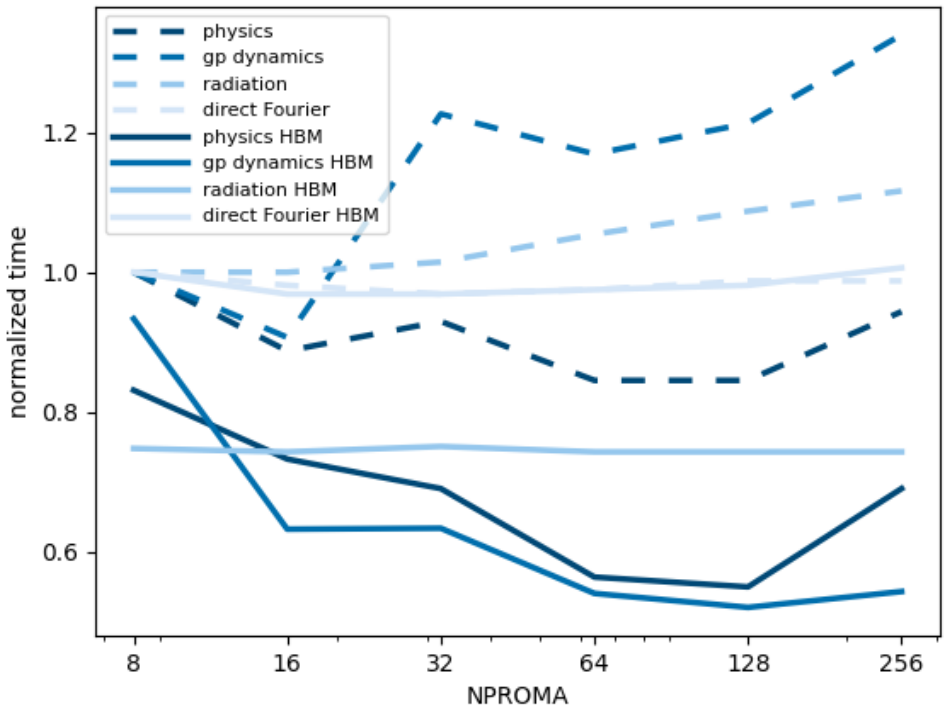


Figure 8: Effect of HBM on cost of different atmospheric components. Lower is better. Grid resolution Tco79

2.1.2 Conclusion

From the previously described work, we have learnt the following about both HBM and SVE:

- The CloudSC mini-app is largely unaffected by High Bandwidth Memory as it is heavily compute bound
  - This can be determined with a very simple test performed on any platform
  - Effort should be put into optimising with vectorisation
- The IFS as a whole does benefit substantially from High Bandwidth Memory
  - This is due to it being made from many different components
- There are a lot of gains to be made with vectorisation
  - Even with only a few lines changed
- Compiler auto-vectorisation can only vectorise what it's given – not refactor entire algorithms

- Code bases written in “legacy languages” can still leverage cutting-edge language-specific features without performance penalty
  - Care must be taken to accomodate hidden differences between language specifications
- Source to source translation could be an ideal way to quickly increase performance while compilers achieve maturity

2.2 Cybeletech - Agriculture

2.2.1 Workflow Description

The CySim software suite is a closed-source library enabling simulation of plant growth on a wide range of species and agricultural contexts (open field vs greenhouse, annual vs perennial crop, cereals vs oleaginous vs vegetables, etc.).

The CySim core library is built in C/C++ and simulates the growth of plants. The mathematical formalization, relying on state-space dynamical systems theory, describes the cropping system as:

$$\begin{cases} X(t+1) = F(X(t), U(t), \theta) \\ X(0) = X_0 \end{cases} \tag{2.1}$$

The three inputs requested are: the environment $U(t)$ is expressed in temporal series of climate data at an hourly or daily timestep, θ are the system parameters which characterize a genotype, X_0 is the initial state (grain biomass, sowing density, etc.). The state function F describing the biophysical processes is controlled by external variables $U(t)$ at time t, which include the climate condition and cultural practices. The output consists of the timeseries of state variables $X(1:t)$, e.g. organs biomass, yield, stresses, etc.

The EUPEX workflow focuses on parametric estimation, i.e. estimating relevant θ parameters for a species, based on observations of plant growth (X partially observed in known U timeseries). The workflow is depicted in Figure 9.

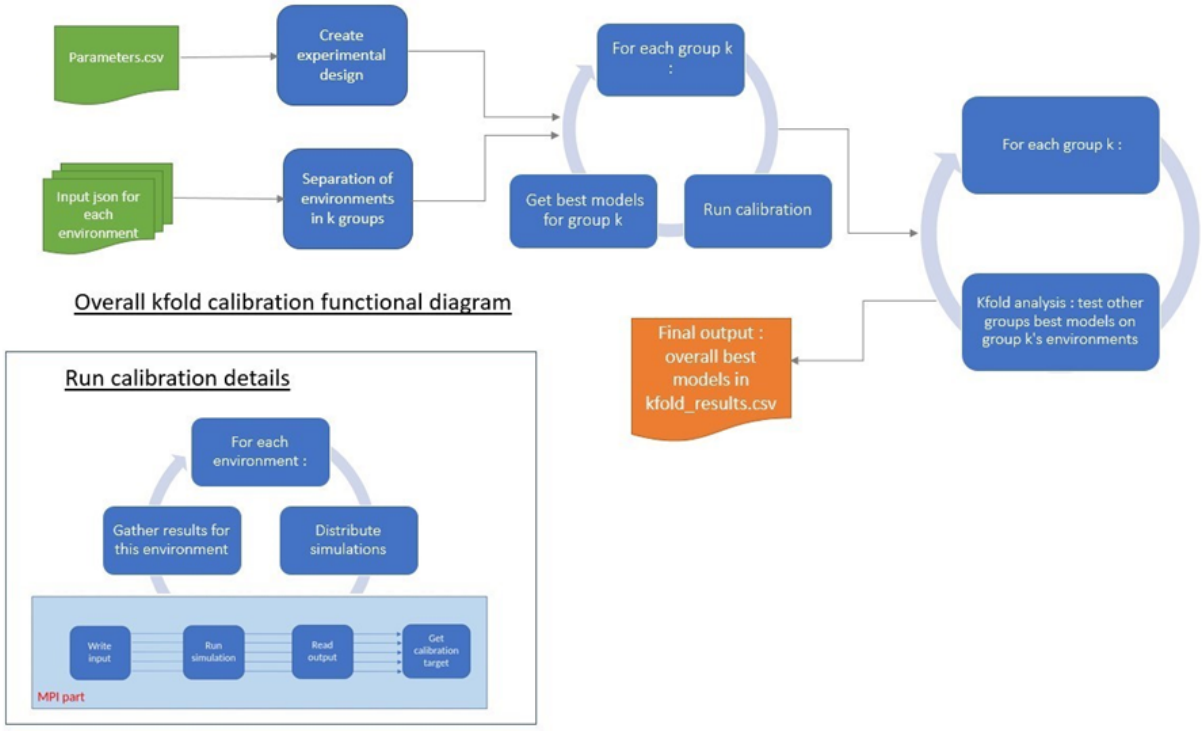


Figure 9: Parametric estimation workflow using the kfold algorithm

The data from the Cybeletech database on crop species properties is used to define the range of acceptable values for each parameter of θ. An experimental design is built based on these extrema and user specifications such as the target number of simulations and the sampling method. The simulation results are gathered and accuracy metrics, such as Root Mean Square Error or Dynamic Time Warping, are computed to evaluate each set of parameters across the different environmental conditions. These results are analysed to identify the parameters set or group of parameters sets describing most accurately the plant response to its environment. If there are enough individuals, they can be separated into subgroups, and calibrations will be performed separately on each of them. The best models of each group will then be tested on other groups in order to identify the statistically most valid among them. If there are not enough individuals, then a single calibration is performed, and the best models are selected.

This therefore needs large simulations, and is a relevant training for the next use case: yield prediction from several climate time series from IFS/ECMWF.

Accuracy estimation and model selection is implemented in Python and has standard package dependencies. The experimental design generation workflow is implemented in Python using SciPY. The experimental design is split according to the number of CPUs available and the simulation is then distributed using MPI. The split and parallelisation are implemented in Python using the library mpi4py [11] while the plant growth model is implemented in C++. The runs of the models are triggered by the Python script using the precompiled .exe file. The C++ library for plant growth simulation relies on standard C++ libraries.

The workflow is usually deployed and run using a Docker image, which contains the precompiled C++ libraries and executable files for plant growth simulation, and Python scripts of the workflow.

When packaged into a container, an application can run inside an isolated environment, embedding its own dependencies into the container. The container can then be run on almost any computing infrastructure. By abstracting the software infrastructure, the container allows us to make the application truly portable and easily deploy on an on-premises server, a virtual machine in the cloud, or even a developer's laptop. It eases guaranteeing reproducibility and compatibility between various application maturity stages: research, development, test, and production. For those reasons, containerization has become a requirement for a lot of companies to deliver software development and is the base of most Cloud services platforms. We believe empowerment by containerization (e.g. abstraction of complex C/C++ or Fortran dependencies) is very relevant for the adoption of new computational resources and supercomputers by scientific software engineers and scientists, as it has been for acceptance of Cloud solutions. For example, containerisation allows us to bring our own release versions to a machine with the guarantee of exact behavior compared to the developer laptop: it allows the control of not just the dependencies, but also the environment, directly managed by the user, with autonomy from machine administration services.

2.2.2 Optimisations and Results

Deployment of Containerized Code

A Docker image for ARM has been built on Cybeletech servers thanks to Docker CLI plugin `docker buildx`. The image is then exported to the Irene cluster to run the evaluation. The data

needed on crop species properties are usually accessed over the internet into postgresql databases. As the Irene cluster does not allow connections with external servers, the data needed on crop species properties are extracted from our databases into json and csv files and copied to the cluster. Therefore, the I/O of the data cannot be evaluated here.

Runs of the workflow have been performed and monitored on Irene. Two container methods have been investigated: Apptainer and Irene Pcocc-rs.

Apptainer [12] is an open source project created to run complex applications on HPC clusters in a simple, portable, and reproducible way. First developed at Lawrence Berkeley National Laboratory, it quickly became popular at other HPC sites, academic sites, and beyond. It uses Singularity Image Format (SIF) images. Apptainer needs to be installed by the user on Irene, since it is not pre-installed by administrators. The launch is then made through a Slurm submission script:

```
#!/bin/bash
#MSUB -r box_128_32r                # Request name
#MSUB -n 2                         # Total number of tasks to use
#MSUB -c 2
#MSUB -T 86400                      # Elapsed time limit in seconds
#MSUB -o output_%I                  # Standard output. %I is the job id
#MSUB -e error_%I                   # Error output. %I is the job id
#MSUB -q a64fx                      # Choosing nodes
#MSUB -A epxt310

./apptainer/bin/apptainer shell --compat -B calibrationV3:/calibration
--fakeroot ~/cysim_20221201_arm.sif
cd /calibration/
export USER=cybele && export HOME=/home/cybele/
source .env
mpirun  --allow-run-as-root -n 2 python3 kfold_calibration.py 2
```

Irene provides a tool for container launch, `pcocc` [2], allowing to deploy both virtual machines (VMs) and containers on compute resources. Container images can be launched interactively on the login node or within Slurm jobs. As per the TGCC manual [2], clusters of containers are instantiated on the fly with a single command, which allocates the necessary resources to host the virtual machines, including private Ethernet and/or InfiniBand networks, and starts VMs with ephemeral disks created from the common image. High performance networks and GPUs can be used from within the virtual machines. However, `pcocc` was implemented for x86 architectures only. A new tool, `pcocc-rs`, has been released in beta at the beginning of 2023 to enable ARM access to containers and VMs.

The launch in Slurm is made through a batch script:

```
#!/bin/bash
#MSUB -r box_128_32r                # Request name
#MSUB -n 2                         # Total number of tasks to use
#MSUB -c 2
#MSUB -T 86400                      # Elapsed time limit in seconds
```

```
#MSUB -o output_%I                # Standard output. %I is the job id
#MSUB -e error_%I                 # Error output. %I is the job id
#MSUB -q a64fx                    # Choosing nodes
#MSUB -A epxt310

pcocc-rs run cysim_20221201_from_irene155
--mount src=~/calibrationV3/,dst=/calibration
-- bash -c "cd /calibration/ && export USER=cybele && export HOME=/home/cybele/
&& source .env && mpirun --oversubscribe -n 2 python3 kfold_calibration.py 2"
```

Runs of the workflow have been performed and monitored on Irene using apptainer in early 2023. We then made the final runs on pcocc-rs when it was released. The results presented are generated using pcocc-rs, but results were identical to those obtained from the primary runs on Apptainer in terms of resource use and computational time.

Memory

Low memory usage is a key on Irene due to the limited High Bandwidth Memory. Instead of keeping all state variables stored in memory, a paradigm has been added to the CySim framework, to reduce "Observables". I.e. the state variables that will be kept in memory on each passed $X(n)$ timestep during the simulation. This way we are more independent from simulation timeline and in our case, RAM usage is reduced by 2/3 [13] for a timeline of 200 days.

For a batch of simulations, the RAM usage has been optimized to remain as stable as possible while the number of simulations increases: as explained in the workflow description, most ressources used for a simulation are destroyed at the end of it, only some data is kept in memory in the python script: a partial extract of the state variables and the accuracy metrics. That data is then written to a file to be accessed in the metrics analysis phase.

The RAM usage is monitored by the `tracemalloc` python library to generate statistics on allocated memory blocks per line. We focus on block allocation during the entire workflow. Given that the launch of a batch of simulations is nearly state-less, the mean RAM usage per node is very stable around 1700MB, independently of the number of simulations, as seen in Figure 10. The RAM usage on x86 architectures Intel® Core™ i7-9850H CPU 2.59 GHz and AMD EPYC 7000 series (AMD EPYC 7571) 2.5 GHz where computed for 100 to 1000 simulations and 4 to 10 nodes. As presented in Figure 11, the RAM usage is slighly superior for ARM than x86, variation between the x86 and ARM infrastructure is < 1% for equivalent number of simulations and nodes.

Computational Time

CPU consumption is tracked through the `CProfile` python library. `CProfile` provides deterministic profiling of python instructions, enabling tracking of each instruction's contribution to the total runtime.

Relative contribution of workflow main steps to computing time are shown in Figures 12, 13, 14.

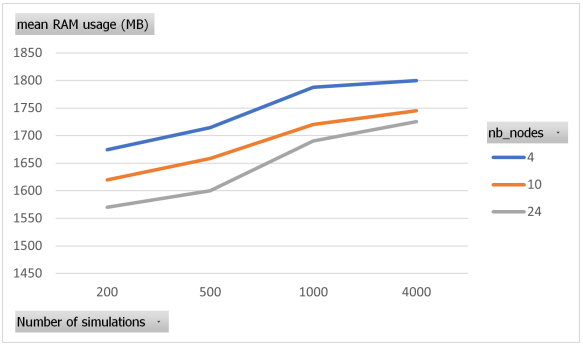


Figure 10: RAM mean usage for the entire workflow on ARM.

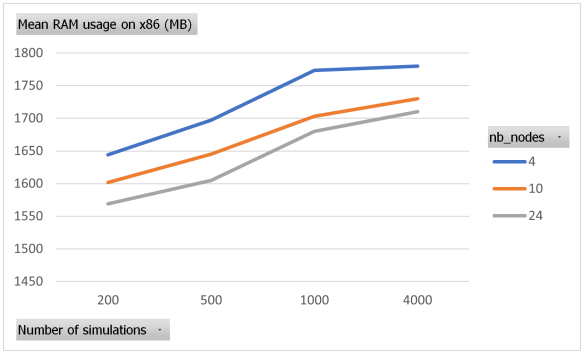


Figure 11: RAM mean usage for the entire workflow on x86 Intel i7-9850H.

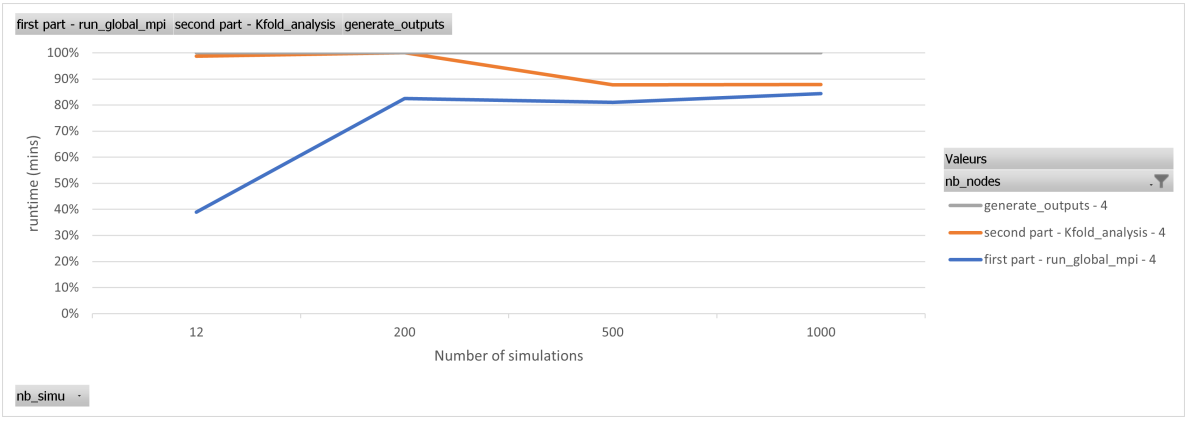


Figure 12: Relative runtime contribution of main workflow functions on 4 nodes.

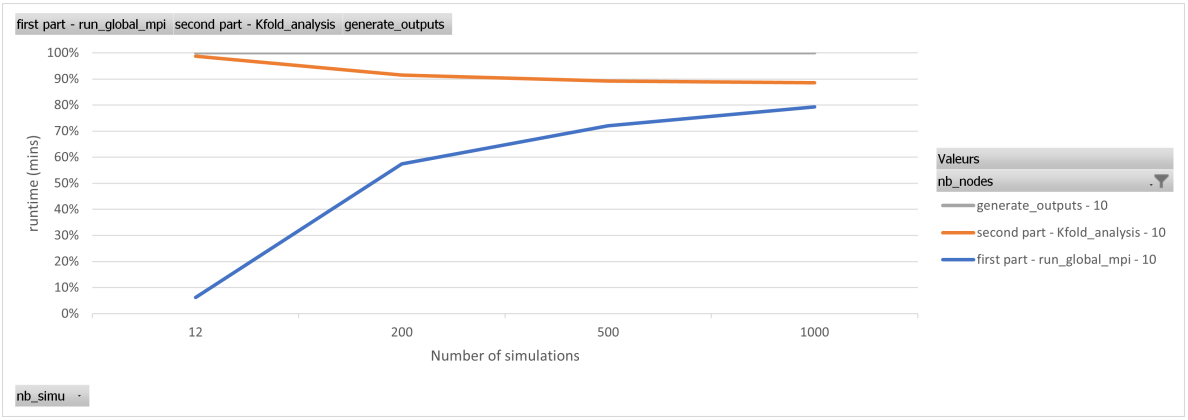


Figure 13: Relative runtime contribution of main workflow functions on 10 nodes.

The construction of the model inputs and experimental design generation are straightforward and do not require high computational power or consume a lot of memory. More over, it is independent of the number of simulations, it does not appear on the Figures 12 to 14 as its contribution is too small to be visualized properly.

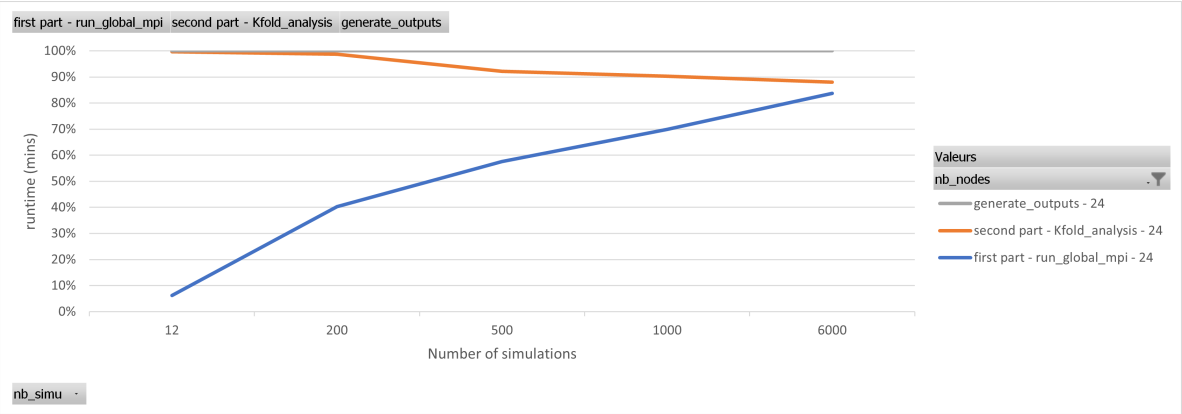


Figure 14: Relative runtime contribution of main workflow functions on 24 nodes.

`generate_outputs` concatenates output creation in the first part (`run_calibration`) and the second part (`kfold_analysis`). The `kfold_analysis` computing time is independent of the number of simulations, as expected, therefore its relative contribution decreases strongly with the number of calibration simulations. Simulations for the calibration phase (`run_calibration`) is the step of the workflow having the highest computational demand and runtime: it surpass 60% of the total runtime at only 200 simulations (see figures 12, 13) and its contribution keeps increasing with a logarithmic shape while simulations number increases. The output generation of the calibration phase is the second most time consuming function in the workflow. Its relative contribution increases with the number of simulations, but remains low compaired to the simulation time itself (14.3% of the `run_calibration` runtime for 1000 simulations).

Therefore, we investigated the speedup ratio of the `run_calibration` phase. The speedup ratio $S(p)$ is defined here as the ratio of the time required by the sequential algorithm $T(1)$, to the time required by parallel algorithm using p processors to solve the same problem, $T(p)$:

$$S(p) = \frac{T(1)}{T(p)} \tag{2.2}$$

With MPI, the master node does not contribute to the parrallelized calculations, thefore p refers is this study to the number of slave nodes, eg. the number of nodes mobilized through SLURM minus one. The speedup ratio is computed based on the data provided by CProfile. Time data is reported in seconds.

The speedup ratio is closed to the ideal value (number of slave nodes), as seen in Figure 15 and figure 16. The speedup ratio is independant of the number of simulations. This scalability profile is very similar to the one obtained on x86 and presented in the previous deliverable.

nb simulations\nb slaves	3	9	23
200	3.00	8.78	20.31
500	2.97	8.95	21.28
1000	3.00	8.94	20.80

Figure 15: Table of the speed ratio correlation to simulations number and slave nodes numbers.

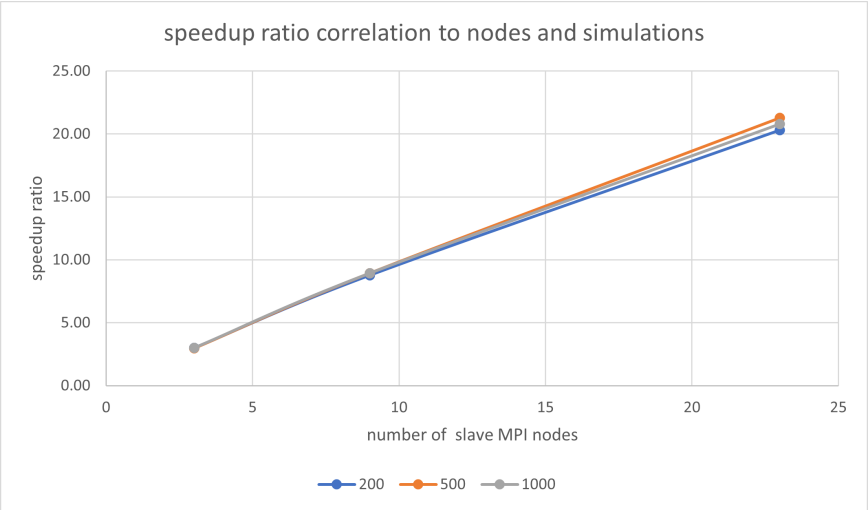


Figure 16: Speed ratio correlation to simulations number and slave nodes numbers.

Simulations have been performed on a x86 architecture Intel® Core™ i7-9850H CPU 2.59 GHz to compare the runtime of the workflow on with IRENE ARM processors. Runtime increases significantly on ARM from x86, as expected. The total runtime increases by 4.1 to 5.2, with no visible correlation to number of simulations. The increase on the `run_calibration` phase is slightly more important on low number of nodes, see figure 17.

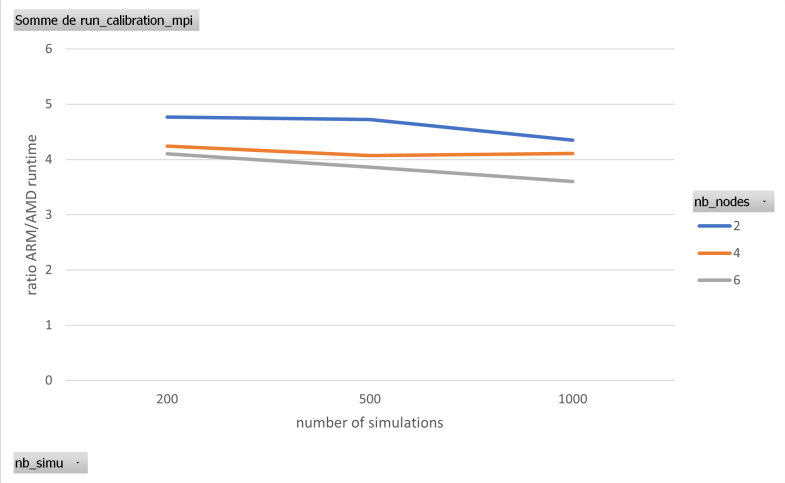


Figure 17: CPU time increase on ARM processors compared to x86 on MPI paralleled part of the workflow.

2.2.3 Conclusion

Simulation is the time consuming part of the workflow, which was scaled through embarrassing parallelism. The speedup ratio on the simulation phase is almost equal to the number of slave nodes used, therefore the scalability is ideal on this part.

The workflow also has embarrassing parallelism on the most RAM consuming part, which is the simulation. Some optimisations were made on the quantity of data to keep in memory after each simulation to aggregate and exploit the results. As a result, we observe a low RAM need increase while the simulation number increases. We can anticipate a maximal request for our workflow of 38GB for 1million simulations, which is very reasonable for EUPEX purpose.

Therefore, we conclude that the workflow performance profile is ready for interfacing with IFS workflow and suited to large simulations at Europe Scale on EUPEX ARM nodes.

Containerization technologies offers great opportunity to port compiled and interpretable codes from x86 to ARM architecture. It guarantees compatibility of x86 applications for a deployment on ARM, without code adaptation and all the associated costs (developers training, maintenance of a double codebase...). This turnkey solution could be a major asset to ease adoption of EUPEX solution.

RAM usage remains the same compared to x86 processors, which is essential regarding High Bandwidth Memory limitations on ARM architectures. However, we observed an impact on computing performance. Future work on EUPEX architecture should investigate how this computing time increase and the ARM energy consumption saving offset each other in the total energy consumption balance.

2.3 Artificial Intelligence for Earth Observation - AI4EO

Earth observation (EO) is one of the main applications of the Space industry as it enables us to monitor land and ocean processes, to analyse the dynamics at work and in the end monitor the earth system. Due to the advent of modern EO programs such as Copernicus of the European Space Agency (ESA) [14], a wide variety of open high-resolution and multi-temporal Remote Sensing (RS) data are available at the global level and can be exploited by research communities, agencies and industry. The fleet of satellites of Copernicus (i.e., Sentinels) includes a wide variety of RS instruments (i.e., active and passive sensors with different resolutions) and can acquire more than 12TB of data per day.

The use case represented by this application "Artificial Intelligence for Earth Observation (AI4EO)" provides a processing workflow able to automatically generate classification maps at large scale (i.e., country scale) in an unsupervised way (i.e., assuming that no new annotated training data will be collected). It leverages the availability of: (i) publicly available land-cover maps, (ii) high resolution RS data, and (iii) street-level crowdsourcing geo-tagged images. To combine the information provided by the satellite and the street-level images, the workflow leverages the capability of Machine Learning (ML) and Deep Learning (DL) models to extract high level semantic features from both the RS data and crowdsourcing images in order to automatically detect reference samples belonging to the same land-cover class. To enable the production of classification maps at large scale, the ML and DL models that are part of the workflow are based on parallel algorithms that can scale on High Performance Computing (HPC) systems.

The first step of the workflow is to retrieve the time series of Sentinel-2 images from the Copernicus Open Access Hub. The images are then pre-processed. The next step is to cluster the pre-processed data to identify the samples that have the highest probability of being accurately associated with their land cover classes. The output of this step is next fed to a classifier. Finally, the output of the classifier is the land classification map. A detailed explanation of the three steps is available in EUPEX Deliverable 3.1.[10]

2.3.1 Clustering Optimisation on A64FX with SVE

In this deliverable, we focus on the clustering of the remote sensing data which is a part of the sample extraction and model training stage of the workflow. In order to annotate large volumes of unlabelled remote sensing data, we need a clustering solution that can efficiently scale on HPC systems. We choose to use HPDBSCAN (Highly parallel density based clustering for applications with noise) which is a parallel implementation of the popular clustering algorithm DBSCAN (Density based clustering of applications with noise).

HPDBSCAN

Highly parallel DBSCAN (HPDBSCAN) [15] is a shared- and distributed-memory parallel implementation of the density based spatial clustering for applications with noise (DBSCAN) algorithm. The stages of clustering in HPDBSCAN can be broadly divided into four major stages.

1. In the first step, all processors load the entire dataset in equal-sized chunks in parallel. Each of the 'd' dimensional data points is assigned to a unique spatial cell corresponding to their location within the data space with respect to the given distance function.
2. The data points are then sorted so that points close to each other are placed together in memory. The data points are sorted using a hash map indexed by the unique cell number. Each entry in the hash table contains the pointer to the data point in the memory and the number of points in that cell. The points in a cell are placed consecutively in the memory and are redistributed among the parallel processors. Each parallel processor must also rebuild its data indices to account for newly distributed sorted data. To balance the computational load for each processor, a simple cost heuristic calculates the number of comparisons between each point in a cell and its neighbourhood. Based on the computed cost, the subspaces are divided among the processors. This helps balance the computation load, especially in highly skewed datasets.
3. The next step is the local computation of clusters carried out by each parallel processor. As this stage is the most compute-intense stage of the clustering process, we focus on vectorization of this stage using the Scalable Vector Extensions supported by the A64FX processor. Here, the points are iterated over in parallel using OpenMP threads, and the neighbours are computed using the Euclidean distance measure. Each point is classified as either a core point or not based on the number of neighbours and on whether its neighbours contain a core point so that the cluster labels can be updated accordingly. We discuss this stage in detail in the next section as we explain our different approaches to parallelization.
4. The final stage is the rule-based merging of clusters. Here we make use of the labels of the halo cells in order to merge clusters that are computed by more than parallel processor. Halo cells are cells that border the subspace of points that are allotted to a processor. They are points in a one-cell layer of thick cells surrounding a subspace. These points are redundant across neighbouring processors and cluster labels of these points are used to merge parts of the same cluster that lay across multiple grid spaces. The cluster labels are broadcast so that each node will directly map the local cluster label to a global one.

The stages of HDBSCAN are illustrated in Figure 18.

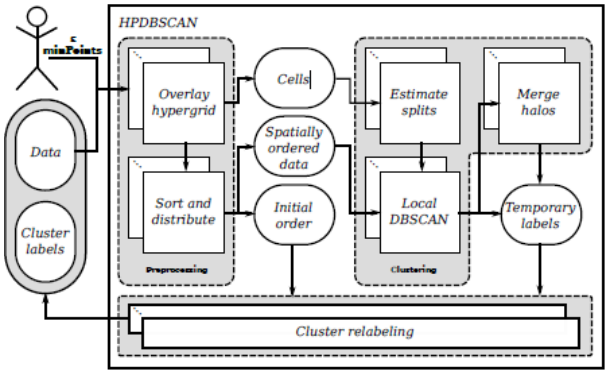


Figure 18: Schematic overview of HPDBSCAN

Optimization of HPDBSCAN

The heart of the HPDBSCAN application lies in the computation of distances from a given point to other points in the dataset. The C++ code to compute the Euclidean distance computation between 'point' and 'other_point' of dimension 'dimensions' is as follows.

```
1 for (size_t d = 0; d < dimensions; ++d) {
2    T distance = point[d] - other_point[d];
3    offset += distance * distance;
4 }
```

In the code snippet shown above, the variable 'offset' stores the computed distance. Please note that computing the square root of the distance has been omitted as the distance values only need to be checked against the squared ϵ value. We checked with GCC 11.2 and ARM CLANG compilers on the A64FX patrition of the Irene High Performance Cluster. The source code was compiled using the -mpcu=a64fx flag with the -O3 optimization flag. On studying the generated assembly using the objdump tool, we found that both the compilers failed to vectorize the loop using SIMD instructions. With the help of SVE intrinsics provided by the arm_sve.h header file, the straightforward way to vectorize the loop would be use predicates to check if the loop count is less than 'dimensions' and load the SVE register which is of size 512 bits with the maximum number of elements that is possible for an iteration and compute the distance.

The key drawback of this straightforward approach is that the number of elements that can be loaded per iteration is limited by the dimensions of the data point. The datasets which we experimented with had dimensions that were mostly within 5 and it is quite rare to come across a dataset that has dimensions that are more than 16 in the case of double dimension floating point values or 32 in the case of single-dimensional floating point values.

In order to exploit the large width of the SVE registers, we load each of the data point coordinates into separate SVE registers. HPDBSCAN sorts the data points in the indexing phase and lays them out consecutively in the memory. A point located close to each other spatially is also placed together in the memory. Hence, while we compute the neighbors of each point in a subspace, most of the neighbors are likely to be already loaded in the cache memory as the potential neighbors of points within a subspace allotted to the processor are likely to be the same. Our measurements of the cache hit rate using Performance Application Programming Interface (PAPI) revealed an average hit rate of 98% for L1 cache and 77% hit rate for the L2 cache implying that only 0.46% of the memory access was from the L3 cache or the RAM. This also implies that the application is mostly compute bound and effective use of SVE intrinsics is important to achieve any significant performance improvements. With the points and their coordinates placed consecutively in the memory, we could easily load the coordinates into different SVE registers.

Figure 19 illustrates the memory layout of a three dimensional dataset and how the SVE registers can be used to efficiently vectorize the Euclidean distance computation.

Here SVE_X, SVE_Y and SVE_Z are SVE registers that store the x, y and z values of the 'j' data points in each iteration. Note that for every iteration, 'j' coordinates can be loaded into 'd' SVE registers where 'j' is the number of floating point elements that can fit in the SVE register and 'd' is the dimension of a data point. The x, y and z coordinates of point 'P' from which the distances to its neighbours must be computed are broadcasted into the SVE_Px , SVE_Py and SVE_Pz SVE

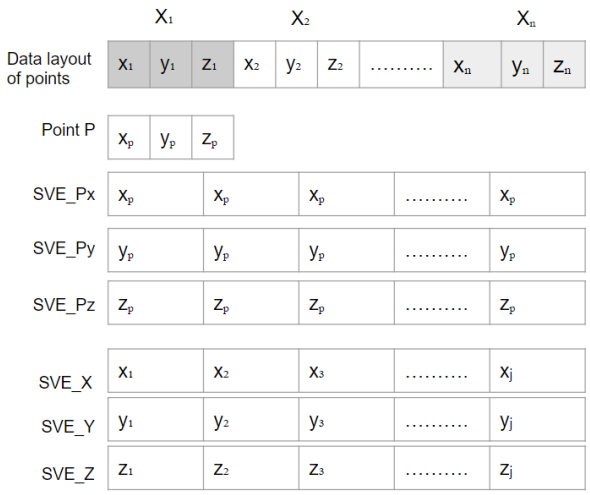


Figure 19: Memory layout of a three dimensional dataset

registers respectively. This has to be done just once before the distance computation loop begins. In order to compute the Euclidean distance, we need to execute the following equation in the distance computation loop.

$$SVE\_dist = (SVE\_X - SVE\_Px)^2 + (SVE\_Y - SVE\_Py)^2 + (SVE\_Z - SVE\_Pz)^2$$

Now that we have the distances stored in the SVE_dist register, we need to check if it is less than ϵ The conditions to check if the Euclidean distance is less than ϵ and if a point is a core point can be easily vectorized using SVE predicates. The complete C++ implementation of the 'indexQuery' function using SVE intrinsics for a single precision multidimensional floating point dataset is given below.

```
for (size_t i = 0; i < n; i += svcntw()) {

    svbool_t pg = svwhilelt_b32(i, n);

    svuint32_t sv_indices = svld1_u32(pg, &neighboring_points[i]);

    /*scaled_index = index * dimensions */
    svuint32_t sv_indices_scaled = svmul_n_u32_z(pg, sv_indices, dimensions);

    svfloat32_t results_v = svdup_n_f32(0.0f);

    for(size_t d = 0; d < dimensions; d++) {

        svfloat32_t point_coordinate_v = svdup_n_f32(point[d]);

        /*scaled_index + d */
        svuint32_t other_point_index = svadd_n_u32_z(pg, sv_indices_scaled, d);

        /*load points from memory onto SVE register */
```

```
20          svfloat32_t other_point_coordinate_v = svld1_gather_u32index_f32(pg,
                &neighbouring_points_ptr[0], other_point_index);
21
22          svfloat32_t diff_v = svsub_f32_x(pg, other_point_coordinate_v,
                point_coordinate_v);
23
24          svfloat32_t diff_square = svmul_f32_x(pg, diff_v, diff_v);
25
26          results_v = svadd_x(pg, results_v, diff_square);
27
28      }
29
30      /*check if distance is epsilon^2 */
31      svbool_t mask = svcmple_n_f32(pg, results_v, EPS2);
32
33      /*count the number of points within epsilon distance */
34      count += svcntp_b32(pg, mask);
35
36      /*load only cluster labels of distances less than ESP2 */
37      svint32_t cluster_labels_of_neighbours = svld1_gather_u32index_s32(mask,
            &clusters[0], sv_indices);
38
39      /*if cluster label < 0, then it is a core point */
40      svbool_t core_points = svcmplt_n_s32(mask, cluster_labels_of_neighbours, 0);
41
42      cluster_labels_of_neighbours = svabs_s32_z(core_points,
            cluster_labels_of_neighbours);
43
44      /*Get the lowest cluster label among all the core points */
45      cluster_label = std::min(cluster_label, svminv_s32(core_points,
            cluster_labels_of_neighbours));
46
47      svst1_u32(mask, &min_points_area[i], sv_indices);
48
49      }
50
51  return cluster_label;
```

Note that a count variable has been added to keep track of the number of points that lie within ϵ radius of point 'p'. Once the 'indexQuery' function returns the cluster label for the currently computed cluster, HPDBSCAN updates the label for all the core points encountered in the current cluster. Hence, we need to revisit the core points again. Here, we use the std::remove function to remove the references to INT_MAX leaving us only references to the core points in the min_points_area vector. The C++ snippet for updating the cluster rules is given below.

```
1  /* Keep only core points in min_points_area vector *
2  auto end = std::remove(min_points_area.begin(), min_points_area.end(), INT_MAX);
3
4  if (min_points_area.size() >= m_min_points) {
5      /* set the label to be negative as to mark it as core point */
6      atomic_min(clusters.data() + point, -cluster_label);
7
```

```
 8     for (size_t other : min_points_area) {
 9         // if point is a core point
10         Cluster other_cluster_label = std::abs(clusters[other]);
11         // check whether the other point is a cluster
12         if (clusters[other] < 0) {
13             const std::pair<Cluster, Cluster> minmax = std::minmax(cluster_label,
                   other_cluster_label);
14             rules.update(minmax.second, minmax.first);
15
16         }
17         // mark as a border point
18         atomic_min(clusters.data() + other, cluster_label);
19     }
20 }
21 else if (clusters[point] == NOT_VISITED) {
22     /* mark as noise */
23     atomic_min(clusters.data() + point, NOISE);
24 }
```

As the iterations localDBSCAN function are parallelized using shared memory parallelism offered by OpenMP, overlapping of the epsilon neighbourhood from different threads can happen leading to a data race condition. In order to avoid this, HPDBSCAN uses a simple atomic min operation to set the cluster label and the core property at once. Also since we know that the indexQuery function already returns the min cluster label, we can do away with the use of std::minmax() function and directly update the rules with the new cluster label. Finally, the check for NOT_VISITED can also be avoided as it is unnecessary. If a point has been assigned a label already, it will keep the label as the label will obviously be smaller than NOISE as NOISE is defined as the maximum integer value -1. And if it is not a part of any cluster, atomic min will result in its label being unchanged, hence keeping its label as NOISE.

Having seen around 40% improvement in execution times with the changes listed above, we then experimented by increasing the OpenMP dynamic chunk size further from 40 up to 4096 where we saw the highest performance improvements. Beyond 4096, we saw that the performance improvements due to shared memory parallelism began to stagnate. A large dynamic chunk size ensures that too many atomic min operations are not carried out in the same memory location.

2.3.2 Experimental Evaluation

HPDBSCAN is already proven to be highly scalable both at the node and thread levels [15]. In this section, we will describe the methodology and findings of the experiments conducted to evaluate the optimizations implemented in HPDBSCAN which were discussed above. The main focus of the investigation is to show the performance improvements over the non-vectorized implementation or one that was only subjected to vectorization by the compiler. The performance evaluation of the optimizations on various datasets with respect to computation time, memory consumption, and the parallel programming metrics speed and scale-up are described in detail.

Hardware Setup

We evaluated the performance of the vectorized code on a A64FX cluster of the Irene supercomputer. A description of the configuration of the A64FX cluster is presented in section 1.

Software Setup

The operation system running on Irene is Red Hat Enterprise Linux Fedora version 8.6. All applications in the test have been compiled with the FUJITSU compiler 4.8.1 using the optimization level '-Kfast' and architecture flag '-KA64FX '. The MPI distribution on Irene is Open MPI version 4.0.5. For the compilation of HPDBSCAN, a HDF5 development library including headers and C++ bindings is required. For HDF5, we used the preinstalled version 1.12.0.

Evaluation of Speed Up Using Vectorized HPDBSCAN

We benchmark the optimized HPDBSCAN application's speed-up using the remote sensing dataset first. The dataset consists of 1827972 entries where each entry consists of a 5 dimensional value of spectral indices. The spectral indices are stored in single precision floating point format. Our principal methodological approach is thereby as follows. Each benchmark is run five times, measuring the application's wall time at the beginning and end of the main() function of the process with the MPI rank 0 and the OpenMP thread number 0. After these five runs, we double the number of nodes and cores. We first evaluate the performance at the core level by increasing the number of OpenMP threads from 0 to the total number of cores in a node which is 48 so that each thread runs on a distinct core. We measure the runtimes of both the vectorized code and the non optimised implementation and plot the runtimes against the number of cores. In addition to that, we have run a base measurement with exactly one core on one node. We also evaluated the distributed memory scalability of the application using MPI keeping the number of threads or the number of tasks per MPI rank at one. We increase the number of cores from 1 to a maximum of 960. We calculated the efficiency for each set of runs using the average runtime. Efficiency was calculated as $Speedup/Number\ of\ workers$. The efficiency observed in only OpenMP parallelization and only MPI parallelization are illustrated in Figure 20 and 21 respectively.

There is a significant drop in the efficiency in the optimized code when only MPI parallelization is enabled. With a constant problem size, besides the increased MPI communication overhead in both the optimized and the non optimized versions, the vectorization overhead in the optimized version begins to offset the performance gains even further.

We then run the application to evaluate the performance of both node and core (MPI/OpenMP) hybrid scalability. We use all the cores available per node and increase the number of nodes from 1 to up to 76. Figure 22 shows the mean runtimes plotted against the number of nodes. Figure 23 shows the efficiency observed in hybrid parallelization. One can see that the runtime of the optimized HPDBSCAN reaches its lowest value of 4s in about 16 MPI ranks itself after which, the efficiency of the optimized implementation decreases drastically as the scope for parallelization of the task assigned to each MPI rank becomes almost negligible as the problem size remains constant.

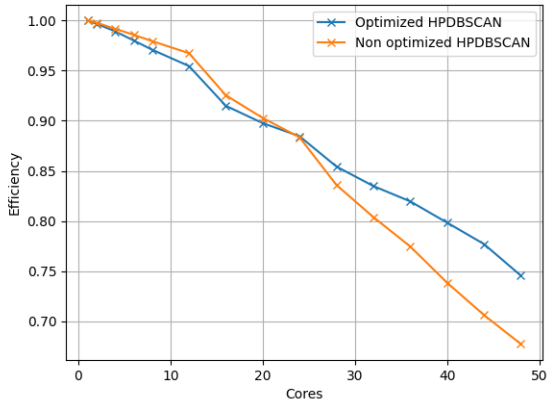


Figure 20: Efficiency observed in OpenMP paralellization

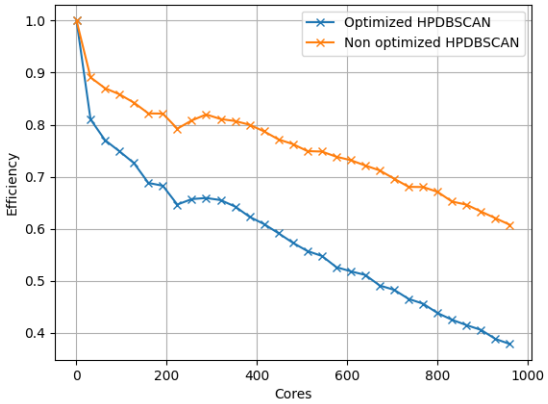


Figure 21: Efficiency observed in MPI only parallelization

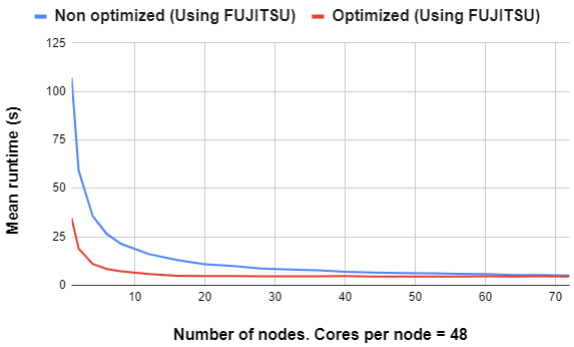


Figure 22: MPI/OpenMP hybrid scalabilty

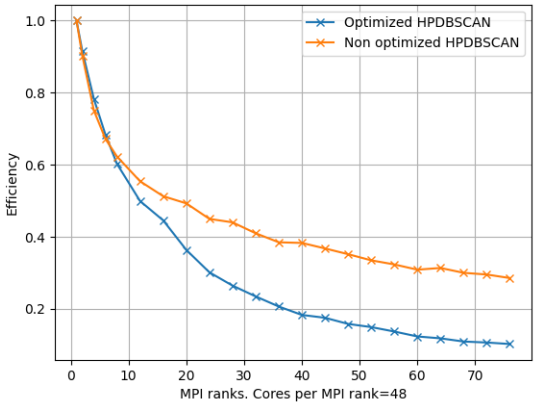


Figure 23: Efficiency observed in OpenMP/MPI parallelization

The speed-up due to the optimizations on a single core and on a single node comprising of all the 48 cores is illustrated in Figure 24 and Figure 25 respectively.

We analyzed the energy consumption of both the optimized and the non-optimized versions of HPDBSCAN. There is a 50% decrease in the energy consumption in the optimized version as shown in Figure 26. The energy consumption was measured using the perf tool to count the following PMU (Performance Monitoring Unit) counters - EA_CORE, EA_L2 and EA_MEMORY. According to the A64FX PMU manual, EA_CORE counts the energy consumption per cycle of the cores. EA_L2 counts the energy consumption of the L2 caches per cycle in a CMG. EA_MEMORY measures the energy consumption per cycle of the core memory group (CMG).

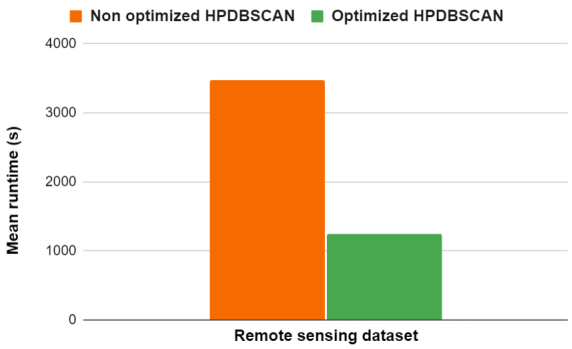


Figure 24: Performance on a single core

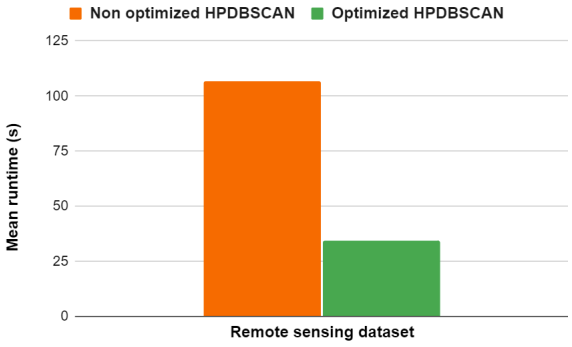


Figure 25: Performance on a single node.

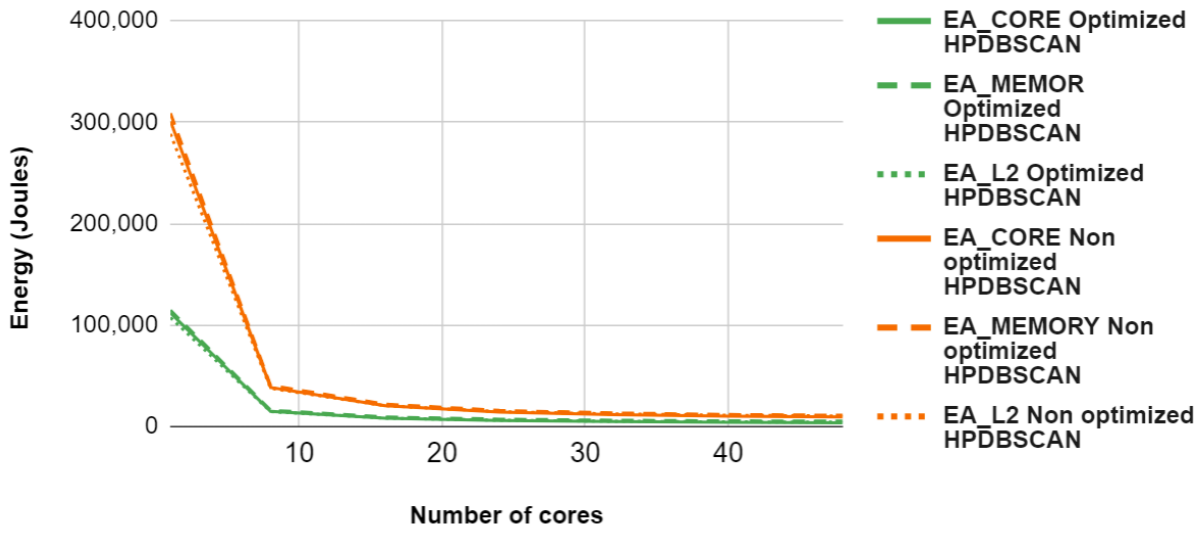


Figure 26: Comparison of energy consumption of optimized HPDBSCAN and non-optimized HPDBSCAN

2.3.3 Perspectives for Classifiers

We also worked on the classifier training phase, which is typically performed on GPUs. We have observed that on a large scale and for heavy models, the use of data parallelism alone has two types of limitations. Firstly, the communication costs incurred by the data parallel approach, which consists in updating the weights with an MPI_All_Reduce operation, becomes very high as the size of the model increases and the number of nodes involved in the Reduce operation grows. Furthermore, this approach increases the batch size linearly with the number of involved compute nodes, which in turn requires a fine control of the learning rate to maintain the same accuracy, thus slowing down the training process.

To limit these data-parallel drawbacks, one approach is to combine the data-parallel and model-parallel approaches. Model parallelism is an approach in which the model itself is distributed across several compute nodes. Thus, activations must be communicated between participating nodes, but

the reduction operation between N nodes of a W data volume can be replaced by k smaller reduction operations, each involving N/k nodes and a W/k data volume, which is much more favourable.

However, this approach is difficult to implement for several reasons.

On the one hand, to speed up computation, a pipelined approach is required to complement model partitioning, in which several micro-batches are injected in sequence to deliver parallel acceleration. The pipeline itself is complex to manage, because the optimal sequence of forward and backward tasks is non-trivial to determine. Increasing the number of micro-batches is one way of increasing resource utilization, but it also results in higher memory consumption. It is then possible to limit memory consumption by using approaches based on re-materialization of activations to save memory, which in turn modifies the load balancing between the GPUs involved in the model parallel approach. Recently, several strategies have been proposed to better organize pipelines to limit idle time on GPUs (Hanayo [16] , Chimera [17], Pipedream [18], MadPipe [19]) and practical and efficient tools are available for re-materialization [20], but these approaches do not explicitly take into account the characteristics of the models and the computing platform, and the combination of model parallelism and re-materialization approaches remains an open problem.

On the other hand, model parallelism alone cannot achieve very high scalability, and it needs to be combined with data parallelism to scale up to thousands or tens of thousands of GPUs. Determining how to partition resources between data parallelism and model parallelism, and how to allocate the different layers onto the platform, are very difficult problems. Recently, frameworks such as DeepSpeed [21], ColossalAI [16]or Megatron-LM [22] have been proposed (see [23] for a recent survey), have tackled the issue but without without explicit consideration of the characteristics of the model and the computing resources.

We have taken these tools in hand and started work on formalizing optimization algorithms combining data-parallelism, model-parallelism and re-materialization. In the coming period, our aim is to apply these techniques to classifier training to improve AI4EO accuracy and scalability.

2.3.4 Conclusion

Vectorization of the most compute intensive part of the HPDBSCAN gave a 66% reduction in execution time and a 50% equivalent decrease in the total energy consumption of the algorithm. Efficient usage of SVE intrinsics can result in significant improvements in application performance when the compiler is unable to exploit the opportunities to vectorize the code. Compiling the code using the Fujitsu compiler contributed to around 8% reduction in the execution time. We explored setting the environment variables with different page policies (demand/prepaging) but we did not see any significant performance improvements. We also tried the '-Kzfill' flag which automatically allocated space in the cache for writing array data in a loop without writing them to memory. However, that too did not yield any further performance improvements. This is probably due to the fact that the optimization works best for sequential access and for code without frequent branching which is not the case with HPDBSCAN. As the application is compute bound, it is prudent to optimize the most compute intensive kernels which in our case was the local DBSCAN computation to realize significant performance improvements.

2.4 SPECFEM3D

In the EUPEX project, we introduced a workflow for constructing a probabilistic scenario tailored for engineering applications. In this phase of the project, our emphasis has been on SPECFEM3D, which constitutes the component of the workflow demanding the highest computational resources.

The SPECFEM codes leverage the spectral-element method for simulating seismic wave propagation across various scales. Alongside specialized inversion tools, these codes form an ecosystem within computational seismology, serving to tackle a wide range of issues associated with seismic tomography and ground-shaking hazard analysis. In EUPEX, we focused on SPECFEM3D_Cartesian that simulates acoustic (fluid), elastic (solid), coupled acoustic/elastic, poroelastic or seismic wave propagation in any type of conforming mesh of hexahedra (structured or not). It employs a high-order spectral-element discretization for unstructured hexahedral meshes. The required input data encompasses topography, 3D wave speed, density, and attenuation fields. The code exhibits scalable performance that is exascale-ready ([24] and related deliverables) and is compatible with the largest supercomputers globally, including LEONARDO (CINECA) and NVIDIA H100 GPU clusters. This solver has been actively adopted within the seismological community for numerous years, as it supports MPI, OpenMP, CUDA, and HIP GPU acceleration.

The code is primarily written in Fortran and C, and there is an ongoing assessment in the community regarding the potential to rewrite it in C++ while considering the adoption of Kokkos. Kokkos is a C++ programming library designed for building high-performance applications on parallel architectures and GPU accelerators. It is created to enable programmers to write code that is portable across a wide range of hardware without having to craft separate implementations for each architecture.

2.4.1 Optimisations

In [10], we conducted a performance analysis of the SPECFEM3D solver, specifically following its porting onto the Irene system Irene@TGCC. The application analysis was specifically centered around porting the SPECFEM3D miniapp (OPT), developed in the framework of CHEESE CoE, onto the A64FX architecture. Subsequently, we tested its performance, taking into account the Scalable Vector Extension (SVE) and optimizations introduced by CHEESE. Following this analysis, the OPT had already proven to be efficient for SVE. In this deliverable, we assess the code's performance and efficiency, considering the the use of High Bandwidth Memory (HBM). The simulations have been performed on a use case on 4 nodes. The metric included in the table in Figure 27 illustrates that the optimized version (OPT) performed on Irene@TGCC better than the default version of SPECFM3D in all the metrics concerning HBM (Read/Write/Bandwith). Yet, the test on Galileo100 at Cineca still exhibits better overall memory performance.

Regarding computational power (MFLOPS) and Instructions Per Cycle (IPC), the OPT version consistently demonstrates superior performance, while Galileo100 stands out as the top performer (see table 1).

It is important to note that the benchmark is not complete in terms of bandwith analysis since it would have required of a largest use case in term of mesh size.

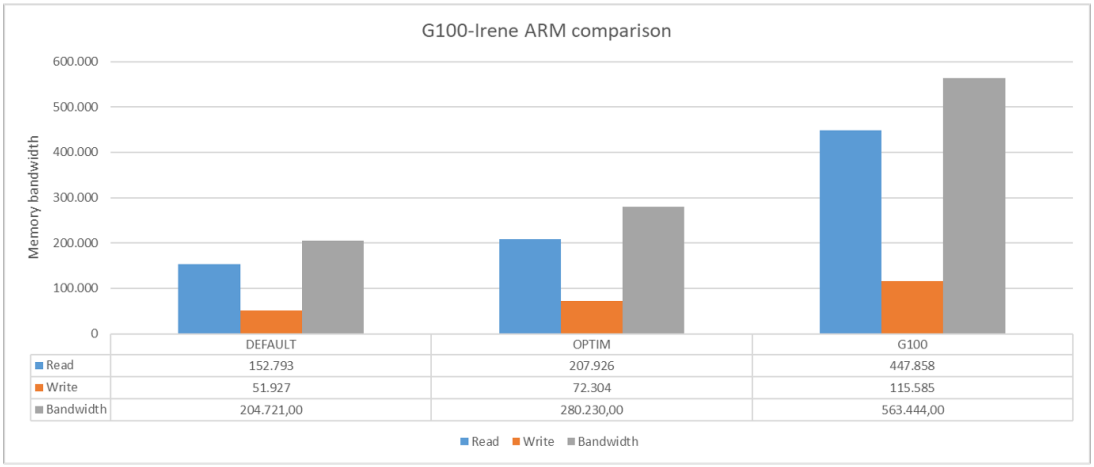


Figure 27: Comparison between OPT and DEFAULT version for HBM metric on Irene@TGCC and Galileo100@CINECA)

	IPC		MFLOPS	
	DEFAULT	OPT	DEFAULT	OPT
Irene@TGCC ARM	0.84	0.96	312.044	418.936
Galileo100 (CINECA)		1.44		769.003

Table 1: Computational power (MFLOPS) and Instructions Per Cycle (IPC) comparisons between OPT and DEFAULT version on 4 nodes each

The solver part is already optimized for GPUs, particularly NVIDIA GPUs, without bottlenecks [24] even at a large scale.

2.4.2 Conclusion

The application analysis is focused around the SPECFEM3D miniapp porting (OPT), developed in the framework of CHEESE CoE, onto the A64FX architecture and in particular on the impact of HBM.

Furthermore, the optimized version (OPT) outperformed by 25% the default version across all the HBM metrics on Irene@TGCC. These metrics list read, write, and bandwidth parameters. Nevertheless, the same version of the code on production system Galileo100 (x86 architecture) still shows better performance.

It is important to note that increasing the size of the use case (i.e. increasing the number of mesh elements) will lead to even better results for the optimized version. Regarding raw computational power (MFLOPS) and Instructions Per Cycle (IPC), the optimized version consistently exhibited better capabilities by 10-20%.

Considering the simulation optimization on GPUs, some analyses, performed in depth jointly with CHEESE/CHEESE-2p COE, show the absence of significant bottlenecks even at large scales [24].The

code thus turns out to be optimized for this type of architectures especially, but not only, with regard to NVIDIA GPUs, which within the seismological community is the most widely adopted.

2.5 ESPRESO FEM

ESPRESO is a highly parallel library for solving engineering problems. It contains several modules that were analysed in [10]. This report focuses on the improvement of the regions in the hot loop of the computation: (a) assembler of system matrices that are built from a description of physical properties defined by a user and (b) solution of the assembled system by a FETI (Finite Element Tearing and Interconnecting) solver.

In general, the assembler gathers coordinates (and possibly some physical parameters) for each element, computes required operations, and stores the element matrix into a system matrix. It was optimised by vectorization for the SVE instruction set. Optimisations are described in Section 2.5.1.

The assembled system matrix is solved by our in-house implementation of FETI-based algorithms. These methods decompose a problem into smaller non-overlapped parts, subdomains, glued together by Lagrange multipliers (reaction forces between subdomains). Then, the solution is computed by a combination of iterative and direct solvers - subdomain solutions are computed by the third-party direct solver and optimal Lagrange multipliers are computed by the iterative solver. Optimisations of the FETI solver are described in Section 2.5.2

2.5.1 Optimisations of FEM Kernels for SVE

ESPRESO solves engineering problems based on Finite Elements Methods [25] (FEM). Input for these methods is usually a set of physical parameters in the form of material properties, initial conditions, and boundary conditions defined on a numerical model. The numerical model is usually an unstructured mesh composed of nodes and elements. Both nodes and elements are usually divided into regions. These regions can be used to set different physical parameters for different model parts. Then, during the simulation, the physical parameters are transformed into a system of linear equations by local matrix *kernels* that are applied to each mesh element.

In the case of explicit evaluation of the global linear system, the kernels can represent a significant part of the overall simulation time. In the case of matrix-free methods, assembling a global system is avoided; thus, kernels dominate the computation time. Hence, the optimisation of the kernels towards efficient utilisation of modern architectures was investigated by many researchers. Usually, it includes some code modifications allowing SIMD instructions and hardware units to be used. In ESPRESO, we implemented vectorization combining cross-element vectorization and cache-blocking optimisation.

In **cross-element vectorization**, small matrix and vector operations in kernels are performed across multiple input elements, i.e., rather than computing operations of a single element in SIMD fashion, the same operation is applied to data from several independent elements inside the vector in parallel [26]. The advantage of this approach is its flexibility since it can be easily modified to work effectively for varying vector lengths. It only requires organising elements in groups of elements with the same attributes so that a single algorithm can be applied to all and a specific data layout in memory to effectively move data in and out of registers.

In the FEM assembler, kernels producing the output matrix are usually broken down into several distinct operations for re-usability and maintainability reasons. These operations can then be applied

consecutively to obtain the desired result for an entire range of input elements. This is, however, not optimal as it unnecessarily increases the cache pressure for larger domains. In **cache blocking optimisation** described in [27] in detail, all functions are applied to the subset of the input range before processing of another subset begins. This significantly improves caching performance and works seamlessly with cross-element vectorization. Usage of cache-blocking techniques in the context of FEM methods can be seen in [28].

The implementation of the above techniques with the achieved speed-up is described in the next subsections.

Original Implementation

This section describes an original implementation of the FEM kernel for heat transfer. It is a simple physics with one degree of freedom for each node selected for its simplicity of description. However, all other physics are based on the same principles. Thus, all techniques described can be straightforwardly applied to other physics in ESPRESO or other FEM libraries.

A concept of FEM kernels is described by a pseudo-code in Listing 2.1 where element matrix K is built for a general element. Several blocks in the kernel can be identified. On lines 14 – 17, values from global arrays are gathered, i.e., values are indirectly copied to local variables according to element nodes. Then, the integration loop over all Gauss points (GPs) is performed. An element is integrated on lines 21 – 23, and temperature-gradient interpolation matrix dND is computed. At the end, we compute the contribution to element matrix K inserted into a global matrix used in the FETI solver. The kernel would be straightforwardly enhanced if a user sets more initial or boundary conditions or requests some outputs from the computed values (e.g., temperature gradient).

```
1  void heat_transfer(
2      const Settings &settings,
3      const double *w,
4      const Vector<setting.nodes> *dN,
5      const Point3D *coo,
6      const double *temp,
7      const double *gradient,
8      const double &c,
9      Matrix &K
10 {
11   Matrix<settings.nodes,3> elm_coo;
12   Vector<settings.nodes,3> elm_temp;
13   for (int n=0; n<settings.nodes;++nodes) {
14     elm_coo[n][0] = coo[settings.node[n]].x
15     elm_coo[n][1] = coo[settings.node[n]].y
16     elm_coo[n][2] = coo[settings.node[n]].z
17     elm_temp[n] = temp[settings.node[n]]
18   }
19
20   for (int gp=0; gp<settings.gps; ++gp) {
21     // integration
22     Matrix<3,3> J = dN[gp] * elm_coo;
23     double det = determinant(J);
24     Matrix dND<3,settings.nodes> = inversion(det, J) * dN[gp];
```

```
25      // evaluation K,
26      K += det * w * c * tran(dND) * dND;
27      if (settings.physical_parameter_X) {
28        K += ...;
29      }
30      if (settings.compute_gradient) {
31        gradient = ...;
32      }
33    }
34  }
```

Listing 2.1: Pseudo-code of a basic FEM kernel

In general, each FEM kernel contains a large number of operations with small vectors and matrices. The number of these operations increases with the element size (due to more GPs; the loop counter on line 20) and with the boundary conditions (physical parameters) that must be considered during the computation (e.g., if statement on lines 27 – 29). The performance of these operations can be improved by vectorization. In an ideal case, the compiler vectorizes the majority of code automatically. However, according to the measurement, the original kernels compiled by the Fujitsu compiler contain no vectorized instructions even though the flags turning on the vectorization were used (*-march=armv8.2-a+sve -KSVE*). Hence, kernels were optimised by the techniques described in the following subsections to improve their performance.

Cache Blocking Optimisations

This section describes the first part of the optimisations focused on the better utilisation of caches and transforming run-time parameters to compile-time parameters allows the compiler to build a code tailored to a particular element type.

The implemented optimisation heavily utilises class templates. It allows the creation of temporal kernel data on the stack and calls a set of required operations on this data kept in caches. Hence, independent of the number of physical parameters, input data is loaded from the memory only once. In addition, to avoid re-implementing the code for processing a particular physical parameter, the kernel is divided into small sub-kernels from which a final kernel is built.

```
 1  template<int nodes, int gps, int dim, int param1>
 2  struct Physics {
 3    struct Element {
 4      alignas(SIZE*sizeof(double)) SIMD coords[nodes][dim];
 5      alignas(SIZE*sizeof(double)) SIMD cond[gps][dim*dim];
 6      // the rest of the element parameters
 7    }
 8
 9    virtual void apply(Element &e) = 0;
10  };
11
12  template<int nodes, int gps, int dim, int param1, class Physics>
13  struct ElementAction: Physics {
14    void apply(typename Physics::Element &e) { ... }
```

```
15 };
```

Listing 2.2: Structure for creation local data on the stack and element action (sub-kernel)

The main challenge of the implemented approach is passing local data computed by one sub-kernel to another. It is solved by a structure *Physics* described in Listing 2.2. This structure must be inherited by all sub-kernels available for an element described by the nested class *Element*. Inside the kernel, this nested structure (*Physics::Element*) is created as a local variable on the stack and passed to the pure virtual function *apply(Element &e)*. The structure must contain all variables available across more sub-kernels.

The datatype of all variables should be *SIMD* with proper alignment. Datatype *SIMD* can be an alias for *double* or an array of *SIZE* elements in the case of cross-element vectorization described in the next subsection.

Listing 2.2 also contains an example of a sub-kernel *ElementAction*. It must inherit from the *Physics* structure and implement the *apply* function. Both structures *Physics* and *ElementAction* are templates with chosen compile-time parameters. It allows the compiler to optimise sub-kernels for different parameters (e.g., element type). The partial specialization allows programmers to distinguish between different versions of the code if needed (e.g., different codes for 2D and 3D sub-kernels). The sub-kernel can be utilized by more physics as long as the variables in the *Physics::Element* structure used by the *apply* function are the same (e.g., gathering coordinates for all implemented physics).

```
 1 template<int nodes, int gps, int dim, typename Physics>
 2 void loop(Settings &settings, size_t elements) {
 3   Physics<nodes, gps,...>::Element element;
 4   SubKernel1<nodes, gps,...> subkernel1(...);
 5   SubKernel2<nodes, gps,...> subkernel2(...);
 6
 7   // main loop over all elements
 8   for (size_t c=0; c<element/SIZE; ++c) {
 9     if (subkernel1.isactive) {
10       subkernel1.apply(element);
11     }
12     if (subkernel2.isactive) {
13       subkernel2.apply(element);
14     }
15     // ...
16   }
17 }
```

Listing 2.3: A loop over all elements

An example of a modified kernel is in Listing 2.3. Initially, the kernel creates *element* with kernel local data and initiates all sub-kernels. Then, active sub-kernels are called inside the loop. Even though there are if statements in the hot loop, they have negligible overhead since conditions are always evaluated to the same value. It is achieved by grouping elements with the same type and parameters into the same group and calling the loop for each such group separately. Since the number of nodes, GPs, and other parameters that determine the size of matrices, vectors, and other temporal variables are template parameters, it is possible to create *element* on the stack.

To efficiently utilise SIMD registers within the loop, the *apply* function should be implemented to process more elements at once (cross-element vectorization). From the implementation point of view, it means using a vectorized datatype instead of the *double* datatype and appropriately reducing the loop counter (the loop is called only *element / SIMD* times). The next section describes the vectorized *SIMD* structure in more detail.

Cross-Element Vectorization for SVE

Typically, the compiler automatically drives the process of code vectorization with little to no control from the developer side. Auto-vectorization requires no effort besides setting correct compilation flags and is also very portable. However, its effectiveness might be limited in some cases. In the area of FEM kernels, this process is even more challenging due to the many elements with different dimensions, nodes, and GPs that must be compatible with SIMD register sizes.

An elegant way of vectorization is proposed by cross-element vectorization. In this approach, processing a single element is substituted by processing multiple elements simultaneously. If the number of simultaneously processed elements is equal to the width of SIMD registers, it is enough to substitute all operations with double by operations with arrays (e.g., *__m128d* in the case of the SSE2 intrinsic instruction set). In C++, a straightforward approach is to provide a special structure encapsulating an array of doubles with overloaded operators and required mathematical functions.

Examples of structures for cross-element vectorization can be found in Listings 2.4 and 2.5. Both structures provide the same functionality. The difference is in internal data representation. The structure in Listing 2.4 stores the array of doubles as *svfloat64_t* with 512-bit size. The structure in Listing 2.5 stores data as *std::array<double, 8>*. Whereas the former is the datatype provided in header *<arm_sve.h>*, the latter is a workaround if the former is unavailable.

The essential requirement of cross-element vectorization, as implemented in ESPRESO, is the known size of SIMD registers. This requirement is automatically fulfilled by intrinsics provided by x86 CPUs architecture since the provided intrinsic functions are tied to register sizes. In the case of Arm and SVE vector extensions, only sizeless datatypes are provided by default. This inconvenience can be solved by *typedef* on line 1 in Listing 2.4 that creates a new datatype with a fixed length [6]. This datatype can be directly used in the *SIMD* structure.

However, this *typedef* must be supported by a compiler since it is created via a compiler-dependent *__atribute__* keyword. Unfortunately, the Fujitsu compiler in version 1.1.0 does not support it. In practice, it prevents the creation of *svfloat64_t* with fixed length and its usage in *SIMD* structures. Listing 2.5 describes a workaround for the Fujitsu compiler. Data in this implementation are stored in fixed-length *std::array*. Contrary to the former implementation, *std::array* cannot be directly used as a parameter of SVE intrinsic functions. Data must always be converted to *svfloat64_t* by function *svld1_f64* with predicate *svptrue_b64()*. Even though it makes the structure slightly more complicated, it does not have a negative impact on the performance.

The *SIMD* structure can be seamlessly combined with the cache-blocking optimisations described in the previous section. To minimise overhead of the *SIMD* structure and calling tiny functions, the *INLINE* macro is used. Depending on the compiler used, it is replaced by a compiler-dependent keyword forcing the compiler to always inline the code. Hence, in practice, most of the kernel code produced by the compiler comprises intrinsic functions for a particular architecture. From the

programmer's point of view, the *SIMD* structure allows the generalisation of the kernel code to have a single implementation for each architecture and arbitrary SIMD register sizes. Adding a new vector extension is about adding a new *SIMD* structure similar to in Listing 2.4 without any change in already written code. Such implementation allows adding new features and physical parameters to kernels with predictable performance and sufficient code portability.

```
1  typedef svfloat64_t __sved __attribute__((arm_sve_vector_bits(512)));
2  typedef svbool_t __svep __attribute__((arm_sve_vector_bits(512)));
3
4  struct SIMD {
5    static __svep mask;
6    __sved data;
7    enum: size_t { size = __ARM_FEATURE_SVE_BITS / 64 };
8
9    INLINE SIMD() noexcept
10     : data(svdup_n_f64(0.0)) { }
11   INLINE SIMD(__sved value) noexcept
12     : data(value) { }
13   INLINE SIMD(const SIMD &other) noexcept
14     : data(other.data) { }
15
16
17
18   INLINE SIMD& operator=(const SIMD &other) noexcept
19     { data = other.data; return *this; }
20
21   INLINE double& operator[](size_t i) noexcept
22     { return reinterpret_cast<double*>(&data)[I]; }
23
24   INLINE const double& operator[](size_t i) const noexcept
25     { return reinterpret_cast<const double*>(&data)[I]; }
26
27   INLINE SIMD operator-() const noexcept
28     { return svneg_f64_x(mask, data); }
29
30
31   INLINE SIMD operator+() const noexcept
32     { return data; }
33 }; // end of SIMD struct
34
35 INLINE const SIMD load1(const double &from) noexcept
36   { return svdup_n_f64(from); }
37
38 INLINE const SIMD load(const double *from) noexcept
39   { return svld1_f64(SIMD::mask, from); }
40
41 INLINE void store(double *to, const SIMD& value) noexcept
42   { svst1_f64(SIMD::mask, to, value.data); }
43
44
45 INLINE const SIMD operator+(const SIMD& v1, const SIMD& v2)
       noexcept
46   { return svadd_f64_x(SIMD::mask, v1.data, v2.data); }
47
48
49
50 INLINE const SIMD operator*(const SIMD& v1, const SIMD& v2)
       noexcept
51   { return svmul_f64_x(SIMD::mask, v1.data, v2.data); }
52
53
54
55 INLINE const SIMD operator-(const SIMD& v1, const SIMD& v2)
       noexcept
56   { return svsub_f64_x(SIMD::mask, v1.data, v2.data); }
57
58
59
60 INLINE const SIMD operator/(const SIMD& v1, const SIMD& v2)
       noexcept
61   { return svdiv_f64_x(SIMD::mask, v1.data, v2.data); }
62
63
64
65 // other non-member operators
```

Listing 2.4: fixed SVE data length structure

```
1  typedef std::array<double, 8> __sved;
2
3
4  struct SIMD {
5   // no fixed mask is possible
6    __sved data;
7    enum: size_t { size = 8U };
8
9    INLINE SIMD() noexcept
10     : data{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0} { }
11   INLINE SIMD(__sved value) noexcept
12     : data(value) { }
13   INLINE SIMD(const SIMD &other) noexcept
14     : data(other.data) { }
15   INLINE SIMD(svfloat64_t value) noexcept
16     { svst1_f64(svptrue_b64(), data.data(), value); }
17
18   INLINE SIMD& operator=(const SIMD &other) noexcept
19   { data = other.data; return *this; }
20
21   INLINE double& operator[](size_t i) noexcept
22     { return data[i]; }
23
24   INLINE const double& operator[](size_t i) const noexcept
25     { return data[i]; }
26
27   INLINE SIMD operator-() const noexcept
28     { return svneg_f64_x(svptrue_b64(),
29         svld1_f64(svptrue_b64(), data.data())); }
30
31   INLINE SIMD operator+() const noexcept
32     { return data; }
33 }; // end of SIMD struct
34
35 INLINE const SIMD load1(const double &from) noexcept
36   { return svdup_n_f64(from); }
37
38 INLINE const SIMD load(const double *from) noexcept
39   { return svld1_f64(svptrue_b64(), from) }
40
41 INLINE void store(double *to, const SIMD& value) noexcept
42   { svst1_f64(svptrue_b64(), to,
43       svld1_f64(svptrue_b64(), value.data.data())); }
44
45 INLINE const SIMD operator+(const SIMD& v1, const SIMD& v2)
         noexcept
46   { return svadd_f64_x(svptrue_b64(),
47       svld1_f64(svptrue_b64(), v1.data.data()),
48       svld1_f64(svptrue_b64(), v2.data.data())); }
49
50 INLINE const SIMD operator*(const SIMD& v1, const SIMD& v2)
         noexcept
51   { return svmul_f64_x(svptrue_b64(),
52       svld1_f64(svptrue_b64(), v1.data.data()),
53       svld1_f64(svptrue_b64(), v2.data.data())); }
54
55 INLINE const SIMD operator-(const SIMD& v1, const SIMD& v2)
         noexcept
56   { return svsub_f64_x(svptrue_b64(),
57       svld1_f64(svptrue_b64(), v1.data.data()),
58       svld1_f64(svptrue_b64(), v2.data.data())) }
59
60 INLINE const SIMD operator/(const SIMD& v1, const SIMD& v2)
         noexcept
61   { return svdiv_f64_x(svptrue_b64(),
62       svld1_f64(svptrue_b64(), v1.data.data()),
63       svld1_f64(svptrue_b64(), v2.data.data())); }
64
65 // other non-member operators
```

Listing 2.5: workaround with std::array

Table 2: List of tested elements with their number of nodes, Gauss Points (GPs), and abbreviations

	nodes		GPs		abbrev.	
polynomial degree.	1	2	1	2	1	2
triangle	3	6	6	6	tr3	tr6
square	4	8	4	9	sq4	sq8
tetra	4	10	4	15	te4	te10
pyramid	5	13	8	14	py5	py13
prism	6	15	9	9	pr6	pr15
hexa	8	20	8	8	he8	he20

Performance

This section summarises the performance of a new assembler. Measurements were performed on the Irene cluster on the A64FX partition with fujitsu/1.3.0, openmpi/4.0.5.2 modules. The baseline is the auto-vectorized kernel. However, according to performance counters, the auto-vectorized kernels do not include any vectorized instructions, i.e., the compiler could not use any vectorized instruction. Hence, a substantial performance improvement is achieved with cross-element vectorization.

Performance comparison of kernels auto-vectorized by the compiler and with cross-element vectorization is shown in Figures 28, 29, and 30. The kernels for auto (compiler) and cross-element vectorization were the same, except that *double* variables in the auto-vectorized kernel were substituted by the SIMD class. The kernels were tested with twelve types of elements listed in Table 2 and with six different settings: 0 – *isotropic conductivity*, 1 – *symmetric conductivity*, 2 –*anisotropic conductivity*, 3 – cartesian coordinate system, 4 – cylindric cartesian system, and 5 – advection. These settings influence the kernel complexity. Settings 0, 1, and 2 impact the pattern of assembler matrices; settings 4, 5, and 6 add more operations to the kernel that must be called.

As shown in the figures, cross-element vectorization significantly improves the performance of the kernels, except kernel 4. This kernel contains an evaluation of trigonometric operations, which have high latency and low throughput in the core and are hard to vectorize. In other kernels, cross-element vectorisation speeds up from 2.41 to 5.88. The average speed-up is 4.27.

2.5.2 Optimizations of FETI Solver

In ESPRESO, the solution of the assembled linear system is performed by the in-house FETI solver. The input for the solver is a set of domain matrices K with glueing matrices B. As was described in more detail in Deliverable 3.1, the FETI solver processing can be divided into two stages: (i) the pre-processing and (ii) the solution of the system. During pre-processing, the most time-consuming tasks include assembling the distributed inverse matrix of the coarse problem $(GG^T)^{-1}$ and factorising the subdomain stiffness matrices K. During the solution phase, forward and backward substitution of the sparse direct solver using the Cholesky decomposition of the matrices K is the most time-consuming part.

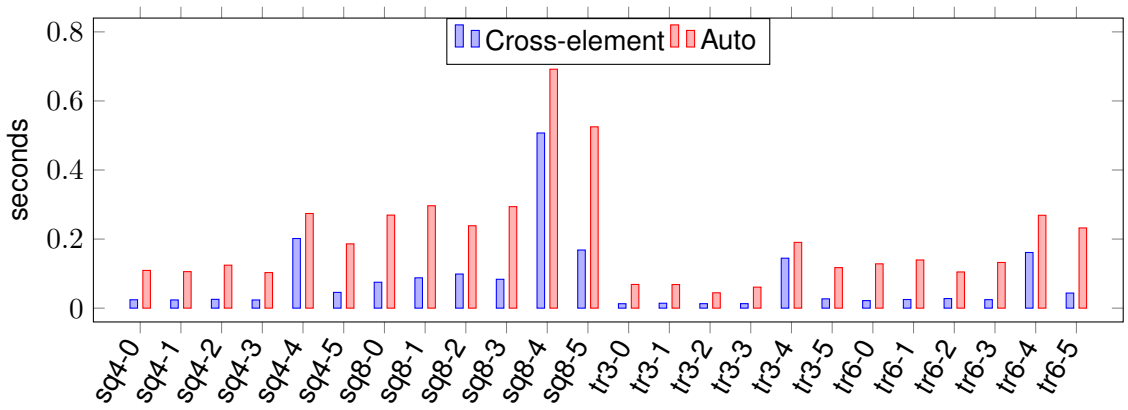


Figure 28: Assembling time of 2D kernels

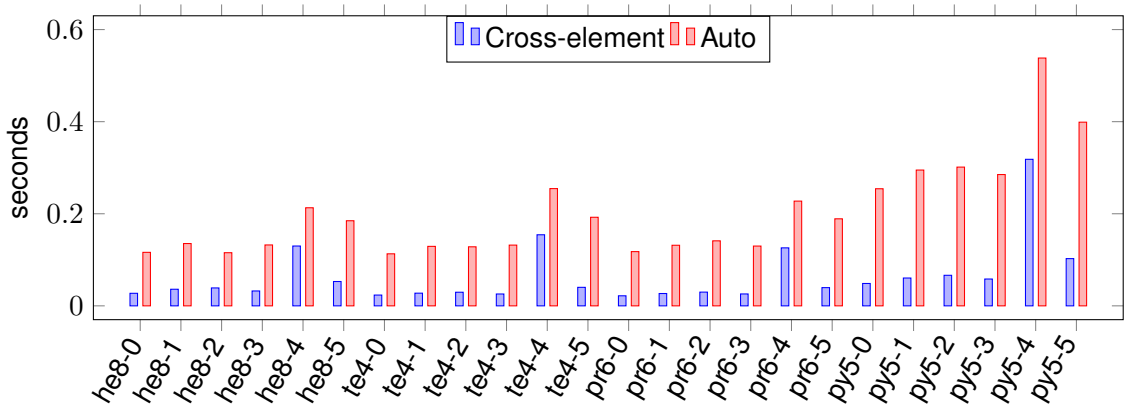


Figure 29: Assembling time of 3D kernels with linear elements

Depending on user settings, the pre-processing phase can be avoided. For example, when solving time-dependent simulations, it is possible to call pre-processing at the beginning of the computation only. Then, most of the run time is spent in the solution phase.

In general, the solution phase is memory-bound. According to the measurement in [10], HBM positively impacts performance itself. The solution for domains that do not fit into caches was faster on Irene with A64FX and HBM than Karolina with AMD EPYC 7H12 and DDR memory. In this section, we investigate another improvement that can be achieved when a solution needs a lot of iterations. The approach for speed-up is similar to the acceleration of FETI for GPU accelerators [29]. It is based on explicitly evaluating the FETI dual operator F.

Explicit Evaluation of FETI Dual Operator

The dual operator F is a product of several matrices: $F = BK^{+}B^{T}$, where B is a very sparse matrix with usually two non-zero values per row representing the glueing of boundary nodes between domains. K is a sparse stiffness matrix of the spatial domain, and K^{+} denotes the pseudo-inverse

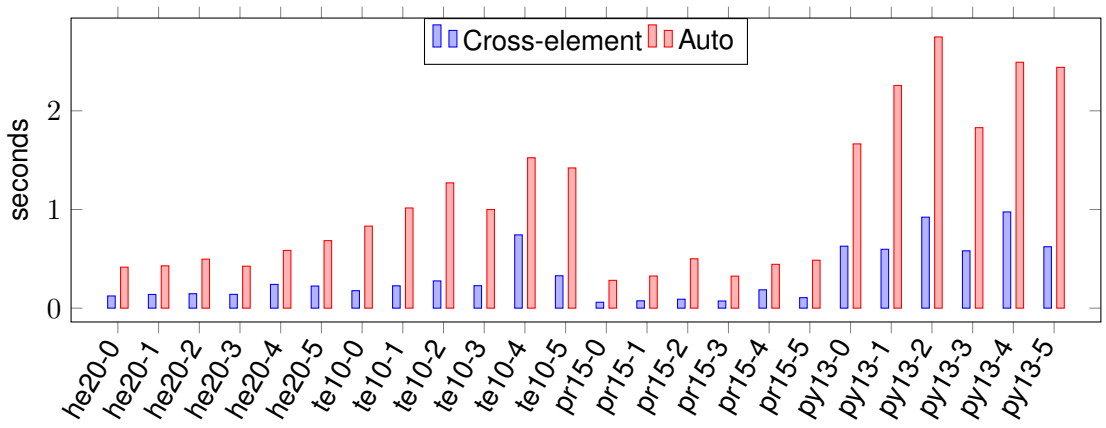


Figure 30: **ESPRESO FEM** : 3D quadratic element kernel assembly time

of K (please see [10] or [30] for more details). The matrix F is dense with its dimension given by the size of a domain surface rather than its volume. The operator must be applied to a vector in each iteration in the solution phase.

In the standard approach, this application (multiplication) is implemented implicitly, i.e., matrix by matrix, in the following way: $Fx = (B(K^{+}(B^{T}x)))$. The implicit evaluation includes two sparse matrix-vector multiplications (SpMV) with glueing matrix B and one forward and backward substitution of the sparse direct solver. Implicit application of the operator takes 95 % of the time of the solution phase. These three sparse matrix operations can be substituted by a single dense general matrix-vector multiplication (GEMV) with the explicitly evaluated F. The measurement section shows that it can substantially speed up the system's solution. On the other hand, an explicit representation of the F operator must be computed during the pre-processing phase. It prolongs the pre-processing phase. Hence, the effect of the substitution is dependent on a particular example. Examples with a relatively small number of iterations of the FETI solver are better to solve implicitly. Examples with many iterations, such as time-dependent simulations, are better to solve explicitly. The particular threshold depends on the quality of the third-party sparse solver available in the system.

In the original implementation of the library, the explicit evaluation was calculated using a special routine provided by Intel MKL. This routine implements an algorithm optimised especially for explicitly evaluating the F operator. Since Intel MKL is not available on Irene, the explicit evaluation was implemented by the following algorithm:

1. convert sparse matrix B^T to dense matrix,
2. use SuiteSparse to compute a solution of $K+$ with dense B^T as right-hand side matrix,
3. multiply the solution with B,
4. store the solution to F.

Compared to the routine in Intel MKL, this algorithm cannot utilise the sparsity of matrix B^T. Hence, its evaluation is proportionally more time-consuming than is reported in [29]. This non-optimality increases the number of iterations the solver must do to make the explicit evaluation faster than

the implicit version. A particular threshold, when switching to explicit evaluation, is shown in the measurement section.

Measurements

In this section, we compare the performance of the FETI solver with implicit and explicit F operators on examples from Deliverable 3.1. Equally to the previous measurement, the ESPRESO FEM library was built with modules fujitsu/1.3.0, openmpi/4.0.5.2, and suitesparse/5.8.1. The measurements are shown in Tables 3 and 4. In tables, *explicit evaluation* is the time spent by computation of the explicit F operator. Apply times denote applying the F operator in the solution phase. Single iteration improvement denotes the absolute difference between implicit and explicit application of F (the single iteration is faster by this time). Speed up denotes how much faster the explicit F operator is. In the time-dependent simulation without recalculating F, the speed-up of the FETI solver converges to this value. Iterations to improvement denote the number of iterations in the FETI solver for which the solution is computed at the same time, no matter whether implicit and explicit F is used.

Table 3: Comparison of implicit and explicit dual operator F for 2D examples

domains	2304	1152	576	288	144	72	36
K+ rows	961	1891	3721	7381	14641	29161	58081
numerical factorisation [s]	0.3729	0.4764	0.4472	0.4791	0.5279	0.6026	0.6651
explicit evaluation [s]	1.3020	2.0383	4.8983	8.1323	12.444	23.301	34.276
implicit apply [s]	0.0312	0.0282	0.0657	0.0651	0.0653	0.0659	0.0665
explicit apply [s]	0.0062	0.0053	0.0038	0.0034	0.0024	0.0021	0.0016
single iteration improvement [s]	0.0250	0.0229	0.0618	0.0618	0.0629	0.0639	0.0649
single iteration speed up	5.04	5.37	17.25	19.32	27.67	32.10	41.70
iterations to improvement	52	89	79	132	198	365	528

Table 4: Comparison of implicit and explicit dual operator F for 3D examples

domains	512	256	128	64
K+ rows	729	1377	2601	4913
numerical factorisation [s]	0.1521	0.1762	0.2563	0.4403
explicit evaluation [s]	3.7615	5.9064	10.608	20.512
implicit apply [s]	0.0173	0.0158	0.0153	0.0168
explicit apply [s]	0.0100	0.0089	0.0081	0.0074
single iteration improvement [s]	0.0073	0.0069	0.0072	0.0094
single iteration speed up	1.73	1.78	1.89	2.28
iterations to improvement	514	851	1469	2175

As seen in the tables, the time for explicit evaluation significantly increases with the size of K, especially for 3D examples. It is caused by the fact that SuiteSparse cannot utilise the sparsity of matrix B. To prove this, we compare the SuiteSparse library with PARDISO from Intel MKL on the Karolina cluster, where both libraries are available. Table 5 shows the ratio between factorisation and

explicit evaluation of the F operator, i.e., relative slow-down of the pre-processing phase. This ratio represents the difference between algorithms implemented in these libraries since the factorization is computed in both libraries at similar times. Even though it is better to use the algorithm used in SuiteSparse for the small matrices, for large matrices, Intel MKL is significantly faster than SuiteSparse (the ratio is lower). We expect that the performance will be similar if we have a library with a similar algorithm. On the other hand, if time-dependent simulations are solved, the pre-processing is performed only once. Hence, it does not influence overall speed up significantly since applying F in the solution phase dominates the run-time.

Table 5: Comparison of Intel MKL and SuiteSparse on Karolina

K+ rows (2D)	25	81	289	1089	4225	16641	66049
Intel MKL	2.60	3.91	1.66	1.34	1.39	1.11	1.39
SuiteSparse	0.07	0.28	0.48	0.81	1.64	2.92	10.20
K+ rows (3D)	125	343	1331	4913	17576		
Intel MKL	2.50	1.59	2.31	4.39	11.55		
SuiteSparse	0.76	1.32	2.03	12.05	66.77		

2.5.3 Conclusion

The optimisations of the ESPRESO FEM libraries include optimisation of the assembler of matrices for the SVE instruction set since auto-vectorization of this part failed, and explicit evaluation of the F operator in the FETI solver to improve its overall performance.

The optimisation of the assembler is based on cross-element vectorization that allows calling kernels on several elements at once. It allows straightforward mapping of expressions in the code to vector units in the processor. Even though SVE introduces the concept of sizeless functions, we have shown that it is possible to use this concept seamlessly for structures with fixed data lengths. The cross-element vectorization speeds up the assembler four times on average.

In the FETI, explicit evaluation of F substitutes several operations with sparse matrices by a single operation with a dense matrix. It positively impacts the speeds of the solution phase of the FETI solver. On the other hand, it prolongs pre-processing time. Hence, the effect of the substitution is dependent on a particular example. Examples with a relatively small number of iterations of the FETI solver are better to solve with the original implementation. Examples with many iterations, such as time-dependent simulations, are better to solve with a new implementation. A particular threshold depends on the quality of the third-party sparse solver available in the system. Unfortunately, the SuiteSparse solver does not contain a special routine to compute substitution as efficiently as the sparse solver in Intel MKL. The availability of this solver on A64FX would improve the performance even more.

During the restructuring of the code, we also observed significant differences between BLAS libraries provided by Fujitsu and other BLAS libraries. The Fujitsu version was up to eighteen times faster. Hence, it must be used to achieve good performance.

2.6 LiGen

One of the main features of the software *LiGen* is the molecular docking, an operation as common, in the field of molecular dynamics, as it is computationally intense. It does so by creating a number of rotated copies of the input ligand (called *poses*) and performing the docking of each one against the target protein. Each one of this docking attempts is then evaluated and assigned a score, which in the end is used to select the best pose. This approach allows a high degree of parallelization and scalability since all the different poses are independent and therefore can be evaluated in parallel. In this work, we focused on the optimization of the serial portion of *LiGen* in order to better exploit the hardware features which the ARM platform provides, in particular the Scalable Vector Extension (SVE) and its 512-bit implementation available on Fujitsu's A64FX processor.

Nowadays the HPC field is not all about performance anymore, it also requires a certain degree of **performance portability** due to the high number of combinations of CPU architectures, ISA implementations, compilers which may arise. For this reason, we wanted to avoid *having to* trust the compiler for performance and we decided to use the Highway[1] library which provides a unified interface for platform specific SIMD intrinsics.

2.6.1 Optimizations

The main optimization we applied is manual vectorization of specific micro-kernels identified through an extensive usage of *flamegraphs*[2]: a particular type of chart which shows the percentage of CPU time used by each function call during code execution. With this approach, we identified 3 micro-kernels (highlighted and marked in Figure 31):

1. `rotate`: a function which applies rigid 3D rotations with discrete steps to the whole molecular structure, accounting for 5.65% of total execution time;
2. `fragment_is_bumping`: a function which checks whether two *fragments* (portions of a ligand) overlap, accounting for 49.38% of total execution time;
3. `sum_over_grid`: a scoring function which computes a discrete estimate of the geometric distance between the input ligand and the protein pocket, not visible in Figure 31 because of *inlining* but accounting for approximately 15% of total execution time.

According to aforementioned measurements, we deemed it useful to tackle the top three hotspots accounting for 71% of the whole docking runtime. Since the second micro-kernel is responsible for nearly half of the total execution time, we will analyze it in detail while the others will be analyzed *indirectly* through performance evaluation and scaling plots.

[1] https://github.com/google/highway
[2] https://github.com/brendangregg/FlameGraph

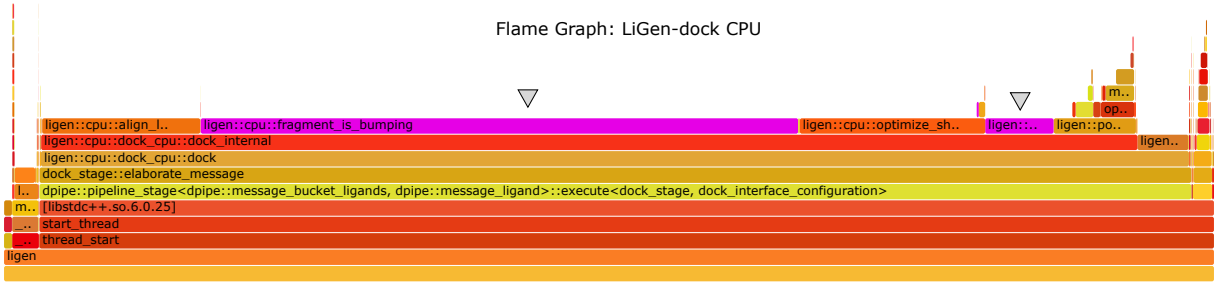


Figure 31: Identification of application hotspots via execution time *flamegraph*. Functions considered in this work are marked in purple.

2.6.2 `fragment_is_bumping`

As introduced in the previous section, the micro-kernel `fragment_is_bumping` is crucial during *LiGen*'s high performance docking workflow since its output is used as an *early exit condition* to determine whether a given ligand's *fragment* overlaps with other parts of the same ligand. If this happens, the ligand is discarded since it may never lead to a successful docking scenario.
In algorithm 1, the pseudo-code for this micro-kernel is shown. The input data of this function is divided into three components:

- `coords`: the array of three-dimensional coordinates of the atoms in the ligand;
- `mask`: an array of bytes used as a mask to select the atoms in the ligand's fragment;
- `length`: the length of the input arrays described above.

```
Input: coords, mask, length
for i = 0; i < length; i = i + 1 do
    for j = i + 1; j < length; j = j + 1 do
        if mask[i] AND mask[j] then
            d = distance(coords[i], coords[j]);
            if d < limitDistance then
                return True;
            end
        end
    end
end
return False;
```

Algorithm 1: Pseudocode of the original `fragment_is_bumping` micro-kernel

It is assumed that `length` is positive and both `coords` and `mask` have at least `length` elements. This function looks for a single unordered pair of different and marked atoms, whose euclidean distance is less than a constant specified limit (*limitDistance* in the pseudo-code) and returns `true` if it succeeds.

The vectorization strategy applied is straightforward: we fix the data of the outer loop by broadcasting it to all available lanes. Then, we vectorize the inner loop by using a predicate vector, which Highway calls *mask vector*, to select the lanes for which the condition holds true as shown in lines 2-5 (listing 2.6). You may notice that we did not divide the inner loop in two parts, *main* and *remainder*, as it's usually done when vectorizing loops; we instead used directly a predicate vector in each iteration since we noticed no performance loss due to all SVE load instructions being implemented as masked operations. This can be seen directly in the generated ASM: in the auto-vectorized version of the code (listing 2.7) we have a total of six `ldr` instructions to load the coordinates of atoms *i* and *j* and

one `ldrb` instruction to load the mask; in the SVE-vectorized version (listing 2.8) there are only three `ld1d` predicated load instructions on SVE registers and one `ld1b` predicated load instruction for the mask. Moreover, a critical difference in the generated auto-vectorized ARM assembly code is the lack of vectorized instructions: only double precision floating point registers are touched (the `dXX` ones) while the manually vectorized code makes heavy use of SVE registers (the `zXX` ones).

```
for (; j < length; j += num_lanes) {
  const maskD iteration_mask = hn::FirstN(d, num_atoms - j);
  const vecM mask_j = hn::LoadN(mask_vector_tag, mask + j, num_atoms - j);
  const auto cond = hn::PromoteTo(mask_to_data_tag, hn::And(mask_i, mask_j));
  const maskD evaluate_fragment = hn::RebindMask(d,
      hn::Ne(hn::Zero(mask_to_data_tag), cond));

        const vecD x_j = hn::MaskedLoad(iteration_mask, d, atoms.x + j);
  const vecD y_j = hn::MaskedLoad(iteration_mask, d, atoms.y + j);
  const vecD z_j = hn::MaskedLoad(iteration_mask, d, atoms.z + j);

        const vecD diff_x = hn::Sub(x_i, x_j);
  const vecD diff_y = hn::Sub(y_i, y_j);
  const vecD diff_z = hn::Sub(z_i, z_j);

        vecD distsq = hn::Mul(diff_z, diff_z);
  distsq = hn::MulAdd(diff_y, diff_y, distsq);
  distsq = hn::MulAdd(diff_x, diff_x, distsq);

        const auto inside = hn::Lt(distance_squared, limit_distance2);
  if (!hn::AllFalse(d, hn::And(iteration_mask, hn::And(evaluate_fragment,
      inside)))) {
    return true;
  }
}
```

Listing 2.6: Vectorized version of Algorithm 1 using Highway abstraction for SIMD instructions

```
.outer_loop_body:
// ...outer loop control flow...
.inner_loop_header:
    add x4, x4, #8
    add x2, x2, #1
    subs x3, x3, #1
    b.eq .outer_loop_body
.inner_loop_body:
    ldrb w5, [x2]
    tst w5, w16
    b.eq .inner_loop_body
    ldr d1, [x0, x12, lsl #3]
    ldr d4, [x3, #8]
    fsub d1, d1, d4
    ldr d2, [x17]
    ldr d5, [x3, #1544]
    fsub d2, d2, d5
    ldr d3, [x18]
    ldr d6, [x3, #3080]
    fsub d3, d3, d6
    fmul d1, d1, d1
    fmadd d1, d2, d2, d1
    fmadd d1, d3, d3, d1
    fcmp d1, d0
    b.ge .inner_loop_body
    mov w14, w13
    and w0, w14, #0x1
    ret
```

Listing 2.7: Auto-vectorized assembly

```
.outer_loop_body:
// ...outer loop control flow...
.inner_loop_body:
    add w7, w5, w2
    mov z7.d, z5.d+
    add w7, w7, #1
    whilelo p2.d, wzr, w7
    cmp x9, x6
    csel x7, x9, x6, lo
    cmp x7, #32
    csel x7, x7, x14, lo
    whilelo p3.b, wzr, w7
    add x7, x0, x13, lsl #3
    ld1d {z16.d},p2/z, [x7, x17, lsl #3]
    ld1b {z6.b},p3/z, [x10, x13]
    fsub z16.d, z4.d, z16.d
    and z7.b, p1/m, z7.b, z6.b
    uunpklo z6.h, z7.b
    ld1d {z7.d},p2/z, [x7, x16, lsl #3]
    fmul z16.d, z16.d, z16.d
    uunpklo z6.s, z6.h
    fsub z7.d, z3.d, z7.d
    uunpklo z6.d, z6.s
    cmpne p3.d, p1/z, z0.d, z6.d
    ld1d {z6.d},p2/z, [x7, x15, lsl #3]
    fmad z7.d, p1/m, z7.d, z16.d
    fsub z6.d, z2.d, z6.d
    fmad z6.d, p1/m, z6.d, z7.d
    fcmgt p4.d, p1/z, z1.d, z6.d
    and p3.b, p4/z, p4.b, p3.b
    and p2.b, p3/z, p3.b, p2.b
    ptest p0, p2.b
    b.ne .return_true
// ...inner loop control flow...
    b.hi .inner_loop_body
    b .outer_loop_body
```

Listing 2.8: Manually vectorized assembly

2.6.3 Datasets

In constructing the experimental dataset, we extracted representative structures from real-world chemical datasets, known to be of pharmaceutical importance during previous large-scale experiments [31] [32]. Considering both dimensions that are taken into account to define a workload's computational complexity, namely number of atoms (including hydrogens) and number of rotamers, three structure classes have been defined: *small*, with up to 64 atoms and 1 rotamer; *medium*, with up to 96 atoms and 12 rotamers, and *large*, with up to 160 atoms and 20 rotamers. For each of those classes, a sample molecule from the testing dataset has been randomly selected and then duplicated to produce a uniform input batch.

2.6.4 Results

As we can see in Figures 33, 35, 34 and 36, we observed a steady improvement in FLOPS per second (with an overall improvement of 272%), vectorization ratio (peaking at approximately 95%), throughput (107% peak improvement) and execution time (with a peak speedup of 102%). However, the same can not be said about memory bandwidth, which was basically halved by the vectorization of the first kernel, `rotate`, as we can see in Figure 37. Then, this effect appears to be mitigated in the next optimization iterations resulting, in the end, in an overall improvement of roughly 15% of the memory bandwidth. These results show us that the auto-vectorized version of *LiGen* has a really different behavior, in terms of both instruction and data access patterns, compared to the manually SVE-vectorized version.

This result is also reflected by the movement of the application's point on the roofline plot in Figure 32: the points move upward (getting closer to the CPU-bound limit) and to the right (getting further from the Memory-bound limit). The roofline shows the improvement of performance of the application across all optimization iterations. The memory bandwidth value of 33.79 GBytes/s has been obtained on a single A64FX core with the `STREAM`[3] benchmark (version `5.10`), compiled with the LLVM C compiler (`clang` version `16.0.6`) and flags `-O3 -DNDEBUG -DSTREAM_ARRAY_SIZE=100000000`, by running it 10 times and taking the maximum value of the `Copy` category.

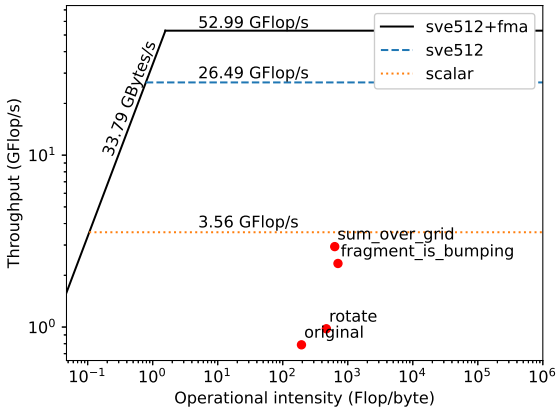


Figure 32: Roofline plot of A64FX CPU on a single core

The throughput values have been obtained by running, on a single A64FX core, the benchmarks included in the `likwid`[4] software (version `5.2.2`) with the following parameters:

- Scalar throughput: `likwid-bench -t peakflops -w S0:4GB`
- 512-bit SVE throughput: `likwid-bench -t peakflops_sve512 -w S0:4GB`
- 512-bit SVE and FMA throughput: `likwid-bench -t peakflops_sve512_fma -w S0:4GB`

Each of these benchmarks has been run 10 times and the maximum value of the `MFlops/s` category has been used for the chart.

In Figures 38 and 39, we show the overall improvement in execution time and scaling between the original auto-vectorized and the last manually vectorized versions of *LiGen*. Each colored band represents the improvement between the original version (the lower bound) and the last vectorized version (the upper bound) for each of the three molecules identified in Subsection 2.6.3. The values used to create these plots have been extracted as the mean value of of 10 independent executions of the whole application, after varying the number of cores to be used.

[3] https://www.cs.virginia.edu/stream/FTP/Code/stream.c
[4] https://github.com/RRZE-HPC/likwid

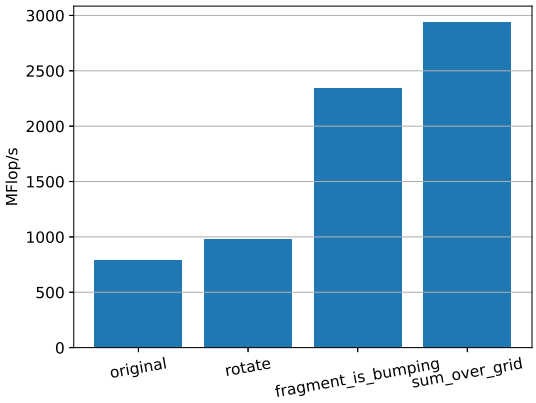


Figure 33: Throughput measurements, expressed as MFLOPS per second, across successive optimization iterations

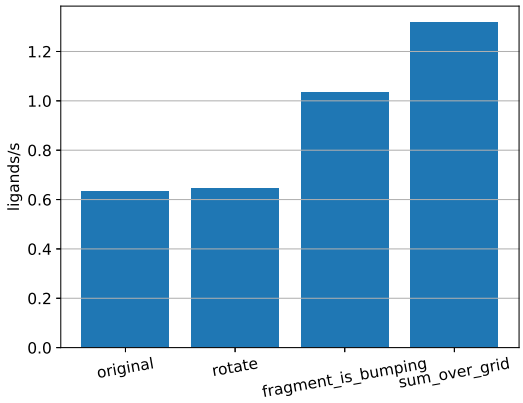


Figure 34: Throughput measurements, expressed as *ligands processed per second*, across successive optimization iterations

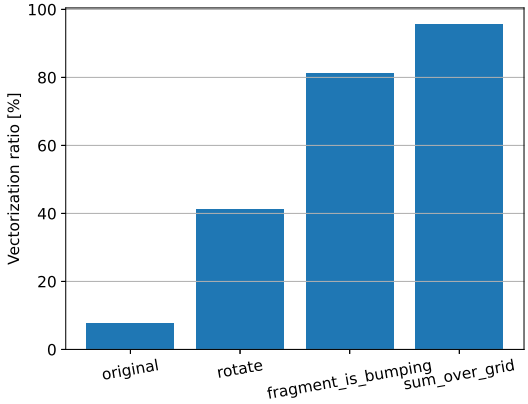


Figure 35: Vectorization ratio improvement across successive optimization iterations

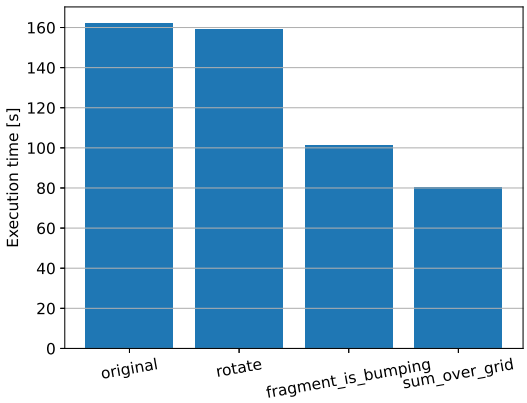


Figure 36: Execution time measurements across successive optimization iterations

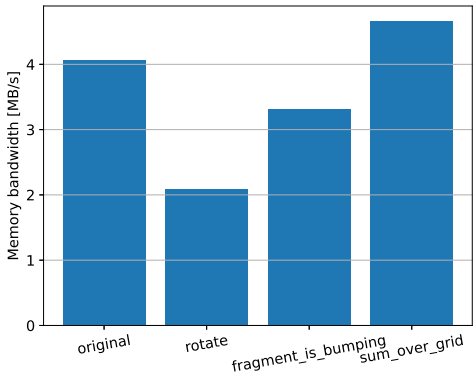


Figure 37: Memory bandwidth measurements across successive optimization iterations

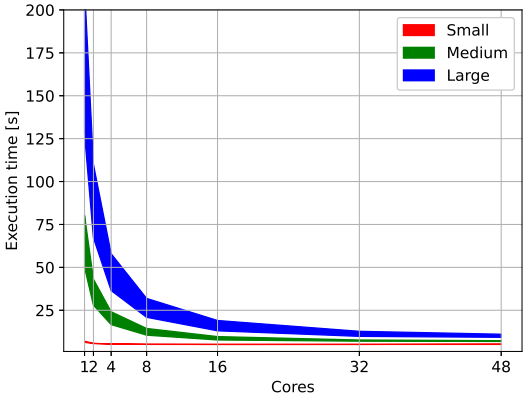


Figure 38: Execution time difference across the original version and the last vectorized version

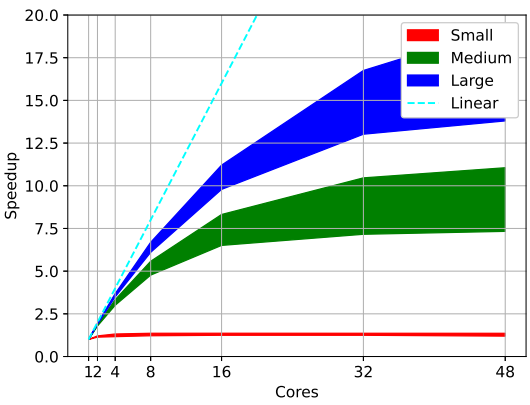


Figure 39: Speedup difference across the original version and the last vectorized version

2.6.5 Conclusion

In this work, we have presented the problem of virtual screening and how it's solved via geometrical docking-based virtual screening at scale. We have presented the steps taken to port, tune and optimize its core algorithms for a platform that provides ARM SVE instructions by leveraging a retargetable, industry-proven SIMD programming paradigm. We then compared the results obtained by the SVE-optimized algorithms to the former generic version that is currently used in production on HPC clusters.

2.7 BIGDFT

BigDFT is a wavelet-based density functional theory (DFT) code with two main operation modes: an approach that scales cubically with the number of atoms and a second one that is linear scaling (LS). In the LS approach, the total work, divided among MPI tasks, grows in proportion to the number of atoms, so large core counts are more naturally accessible to LS-BigDFT than cubic scaling (CS) approaches. The DFT workflow is based on a double-loop structure, where the orbitals are optimized for a given electronic density, and then utilized to build a new density operator, until convergence, see Figure 40. The CS approach still remains interesting as it does not suffer from the additional constraints imposed by the locality of the orbitals in LS mode, and it enabled the treatment of more accurate DFT approximation. A notable case is represented by the calculation of the Fock operator in the so-called exact exchange functionals. According to the Hartree-Fock model, the calculation of the exact exchange energy E_X requires a double summation over all the N occupied orbitals ψ_i, $i = 1, \cdots, N$.

$$E_X = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\int\int \mathrm{d}\mathbf{r}\,\mathrm{d}\mathbf{r}'\,\frac{\psi_i^*(\mathbf{r})\,\psi_j(\mathbf{r})\,\psi_j^*(\mathbf{r}')\,\psi_i(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\,. \tag{2.3}$$

Each orbital contributes to the one-particle density matrix of the DFT problem, which is related to the eigenproblem of the Kohn-Sham Hamiltonian operator, of which the ψ_i are the eigenstates. In the EXX functionals the Hamiltonian contains the so called Fock operator $\hat{D}_X$, whose action onto a KS orbital reads:

$$\left[\hat{D}_X\psi_i\right](\mathbf{r}) = \sum_j \int \mathrm{d}\mathbf{r}'\frac{\psi_j^*(\mathbf{r}')\psi_i(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\psi_j(\mathbf{r}) \tag{2.4}$$

The numerical evaluation of this quantity has a computational cost which might constitute a severe limitation for calculations with highly precise basis sets. Systematic approaches like plane-wave and wavelet basis set density-functional codes evaluate the exact exchange in a similar way. They form all the $N(N+1)/2$ charge densities $\rho_{i,j}(\mathbf{r}) = \psi_j^*(\mathbf{r})\,\psi_i(\mathbf{r})$ – also named co-densities – and then solve the Poisson equation (PEq) for each of them. The Poisson equation can be solved in the basis with $M\log(M)$ scaling for all boundary conditions, where M here indicates the number of points of the (uniform) grid of the direct space simulation domain. The same scaling can be obtained in a plane wave program for periodic boundary conditions. Thus, the number of operations behaves as $N^2 M\log(M)$ for sufficiently large M and N. The underlying convolutions require $M\log(M)$ operations, utilizing Fourier techniques, essentially based on zero-padded FFTs [33].

2.7.1 Optimisations

To increase the performance and enable the solution of increasingly large problems, BigDFT has been GPU enabled since 2009 [34] utilizing NVIDIA's CUDA language [35]. CUDA has been employed in the acceleration of the Fock operator calculations (2.4) with the intensive usage of CuFFT [36] calls in the Interpolating Scaling Function Poisson Solver of the code. In particular, as alluded to above in Section 2.7 and as was shown in [37], the expensive evaluation of the EXX required in the cubic-scaling PBE0 approximation can be offloaded to GPUs to decrease the computing time and, consequently, to achieve computing times which are competitive to the less accurate PBE approximation. Relying solely on CUDA for the GPU acceleration is no longer sufficient — it only

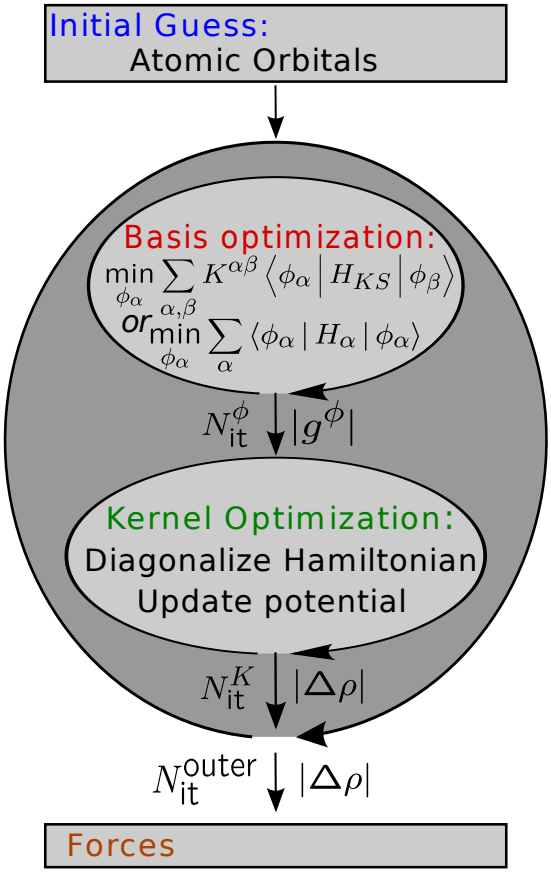


Figure 40: Double loop workflow overview

allows to offload to NVIDIA GPUs — considering the recent usage of other companies' GPGPUs (e.g. AMD's Instinct series [38] and Intel's Max Series [39]) in several of the fastest supercomputers in the world [40, 41].

There are several cross-platform alternatives to CUDA for the development of heterogeneous codes [42, 43, 44]. In the present contribution, we focus on porting BigDFT to SYCL [45]. SYCL is an open standard developed by the Khronos group which was released in 2014 to enable the development of cross-platform code for heterogeneous processors in C++ and which has been implemented in several compilers [46, 47, 48, 49]. Our focus lies on Intel's oneAPI DPC++ compiler due to two reasons. First, it enables the execution of the code with various backends. Of particular interest in the present contribution, aside from the performance on Intel GPUs, is the SYCL performance on CPUs to show the viability of removing the default OpenMP-parallelized Fortran code in favor of the SYCL code. Secondly, CUDA and SYCL are in many ways similar, which permits a swift migration from CUDA code to SYCL. More specifically, in CUDA, each kernel, i.e., the code executed on the GPU, is launched over a "grid" of thread "blocks" consisting of so-called "warps" each of which represents a set of 32 "threads". SYCL, on the other hand, launches kernels similarly over a "nd-range" of "work-groups" each comprised of "sub-groups" consisting of either 16 or 32 "work-items". The difference is that the sub-group size in SYCL is variable whereas it is fixed to 32 work-items in the CUDA case. Moreover, CUDA provides access to grid properties (e.g. blockIdx,

```
//CUDA
__global__ void post_computation_kernel(int nx, int ny, int nz, double *rho, double
↪  *data1, int shift1, double *data2, int shift2, double hfac)
{
    int tj = threadIdx.x;
    int td = blockDim.x;.
    int blockData = (nx*ny*nz) / (gridDim.x*gridDim.y);
    int jj = (blockIdx.y*gridDim.x + blockIdx.x)*blockData;

    for (int k=0; k<blockData/td; k++) {
        int idx = jj + tj + k*td;
        data1[idx+shift1] = data1[idx+shift1] + hfac*rho[idx]*data2[idx+shift2];
    }
}


//SYCL
void post_computation_kernel(int nx, int ny, int nz, double *rho, double *data1, int
↪  shift1, double *data2, int shift2, double hfac, const sycl::nd_item<3> &item)
{
    int tj = item.get_local_id(2);
    int td = item.get_local_range(2);
    int blockData = (nx*ny*nz) / (item.get_group_range(2)*item.get_group_range(1));
    int jj = (item.get_group(1)*item.get_group_range(2) +
    ↪  item.get_group(2))*blockData;

    for (int k=0; k<blockData/td; k++) {
        int idx = jj + tj + k*td;
        data1[idx+shift1] = data1[idx+shift1] + hfac*rho[idx]*data2[idx+shift2];
    }
}
```

Figure 41: Example of one of the BigDFT CUDA kernels compared to the SYCL equivalent. The SYCL code was automatically generated using Intel ® DPC++ Compatibility Tool version 2023.1.0. Note that there is a simpler version of the above SYCL code.

threadIdx, gridDim) implicitly whereas SYCL wraps these parameters in, e.g, an “nd_item” which has to be passed to kernels explicitly. To illustrate the commonalities and differences between SYCL and CUDA, Figure 41 shows the same kernel written in CUDA and SYCL. To further simplify the code migration from CUDA to SYCL, Intel provides the DPC++ compatibility tool (dpct). It performs the code migration automatically. In fact, the SYCL code shown in Figure 41 was automatically generated with this tool. The downside of this automatized approach is that an additional dpct interface layer may be added to the generated code, which may increase code complexity and impact the performance. The SYCL implementation presented in this contribution is based on the automatically generated code from the dpct tool[5] which was manually cleaned and optimized to achieve the performance demonstrated in this contribution.

[5]Intel DPC++ Compatibility Tool version 2023.1.0. Codebase:(89a0192e122343c2a13cec7dc6d57cab899c7b64)

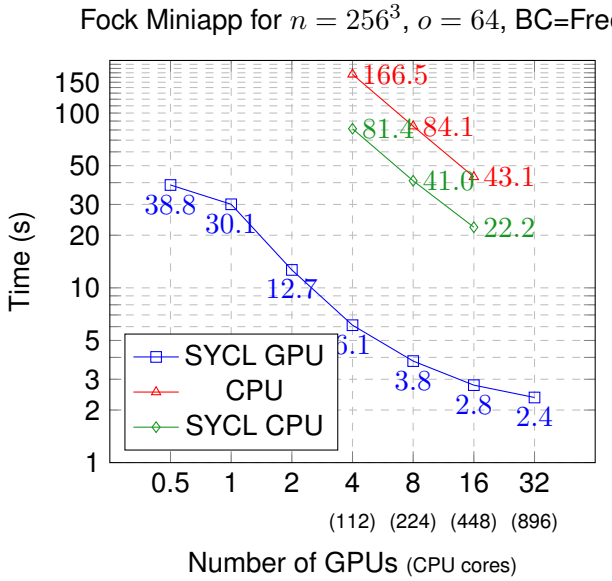


Figure 42: Computing time of the Fock miniapp for free boundary conditions, a grid size of $n = 256^3$ and $o = 64$ orbitals. In the GPU case, one MPI rank was pinned to each stack of a GPU up to a maximum of 8 MPI ranks per compute node. In the CPU cases 16 MPI ranks per node were used. Note that the CPU tests on 8 nodes are missing since the number of MPI ranks cannot exceed the number of orbitals. The first data point was performed on one stack of the two stacks of the Intel GPU.

We have tested the performance of the new SYCL implementation on the Ponte Vecchio (PVC) Intel GPU and the CPU to the existing CUDA and CPU (OpenMP) implementations on the basis of the "Fock miniapp". The Fock miniapp is a small test program to evaluate the performance of the computation of the Fock operator (2.4) in a way which is representative for the Fock operator evaluation performed in the execution of the full BigDFT suite. Extensive tests have been performed on several runtimes in a PVC-based architecture, which are under non-disclosure and will be presented in a forthcoming publication. The present section compares the performance of the different implementations on the basis of the Fock miniapp. The results in this section are highly representative for the performance of the full suite in many situations, since the Fock operator evaluation constitutes the most time-consuming computations in the full suite when using the PBE0 approximation. The first workload is related to a single Fock operator evaluation for a grid size of 256 points in each of the three dimensions (for a total of 256^3 points), 64 orbitals, and free boundary conditions (which results in a grid size for the Poisson solver of 512^3 points). The results of three different code versions, namely, the original CPU implementation, the SYCL implementation on the CPU and the SYCL implementation on Intel Max 1550 GPUs, are shown in red, green, and blue, respectively. One can observe that, i) the SYCL implementation is significantly faster on the CPU than the original CPU implementation by approximately a factor two, ii) the SYCL implementation on the Intel GPU significantly outperforms the other implementations, and iii) the scaling of the GPU implementation levels off from 1 to 8 nodes (i.e. 4 to 32 GPUs) due to the increasingly small computing times which fail to hide the communications. For the GPU runs, 1 MPI rank per GPU stack was used (i.e., 2 MPI ranks per GPU, 8 MPI ranks per node), whereas 16 MPI ranks per node were used for the CPU runs.

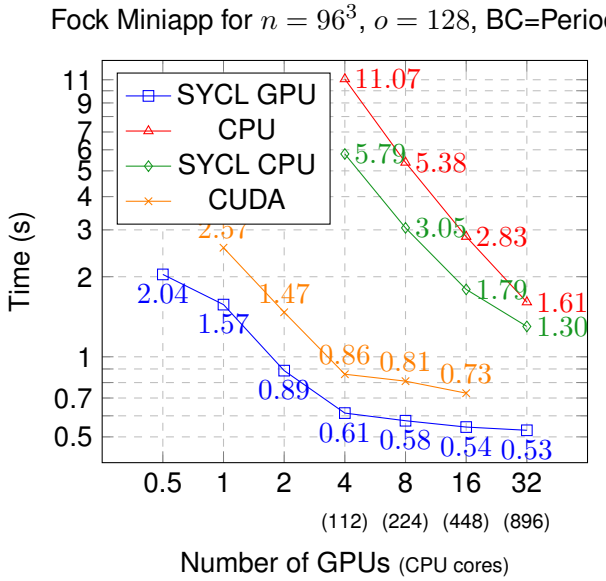


Figure 43: Computing time of the Fock miniapp for periodic boundary conditions, a grid size of $n = 96^3$ and $o = 128$ orbitals. In the SYCL GPU case, one MPI rank was pinned to each stack of a GPU up to a maximum of 8 MPI ranks per compute node. In the CPU cases 16 MPI ranks per node were used. Note that this workload coincides with the exact exchange computation performed during a full solve of the H2O-32 case presented in Figure **??**. Note that in the CUDA case, two GPUs constitute a node whereas in all other cases it is four GPUs per node. The first data point was generated on one of the two stacks of the Intel GPU.

The second workload shows a Fock operator evaluation for a grid size of 96 points in each of the three dimensions (for a total of 96^3 points), 128 orbitals, and periodic boundary conditions (which does not require zero-padding and thus maintains a grid size of 96^3 points for the Poisson solver). In addition to the previous three configurations, the graph also shows results achieved on NVIDIA A100 GPUs with the CUDA implementation. The CUDA results were generated by pinning two MPI ranks to each A100 GPU to utilize the same number of MPI ranks per GPU as in the SYCL tests. One can observe that the SYCL implementation on the Intel Max GPU is highly competitive in terms of computing times. Similarly to the workload shown above, the scaling in the SYCL GPU case degrades significantly for more than a single node and the computing times even increase from four nodes to eight nodes due to increasing time required for the communication. The different CPUs in the SYCL-GPU and CUDA cases have minimal impact on the presented execution times since the majority of the timed code is executed on the GPUs.

Table 6 compares the average HBM and L3 bandwidths during the single-stack executions on the Intel GPU of the above described workloads. From the table it is evident that the smaller workload ($n = 96^3$) fits into L3 cache and thus induces minimal HBM traffic. The larger workload, in contrast, does not fit in L3 and induces heavy HBM traffic. In the case of the larger workload, the code utilizes on average 783 GB/s of HBM bandwidth, which represents approximately 48% of the theoretical peak bandwidth of a single stack. The complex-to-complex FFT with the maximal averaged bandwidth out of all complex-to-complex FFTs achieved a bandwidth of 938 GB/s.

Case	Read (GB/s)		Write (GB/s)		Read+Write	
	L3	HBM	L3	HBM	L3	HBM
$n = 256^3, o = 64$	0.02	468	336	315	336	783
$n = 96^3, o = 128$	1172	66	1744	48	3516	114

Table 6: Comparison of the average read and write L3- and HBM-bandwidths achieved by the Fock miniapp on a single stack of the Intel GPU. The smaller case ($n = 96^3$, $o = 128$) fits in L3 cache and therefore hardly utilizes HBM. The larger case ($n = 256^3$, $o = 64$) does not fit in L3 and induces heavy HBM traffic.

2.7.2 Conclusion

The future guidelines of BigDFT developer groups will therefore be based on such a blueprint, where we will inspect the optimal sources of optimisation in view of:

1. Extending the FUTILE library API to include in the same structures CUDA and SYCL calls to accelerated kernels, thereby enhancing the readability of the host code
2. Measuring the performance figures on the basis of the percentage of the total bandwidth of the node
3. Providing a fully portable programming paradigm which has the ambition to be executed on the present-day and emerging technologies, with limited intrusivity for the developer and minimal guidelines for the user

We plan to use the Just-In-Time programming paradigm (like an underlying OpenCL layer) to explore the portability of the approach. This can be a useful strategy to, on one hand, abstract the portability layer of this application to a single programming paradigm, and on the other hand, lower the barrier of portability to other architectures (ARM, AMD GPU, Grace Hoppper,. . .).

2.8 OpenGADGET

OpenGADGET [50] [51] is a Cosmological, N-body Simulation code used to simulate the evolution of cosmic structures, namely the galaxies and the clusters of galaxies, in the framework of a relativistic expanding background. While gravitation is the physical process mainly responsible for the growth of those objects in early times or at large scales, at minor scales the baryonic processes also play an essential role in shaping the observable properties of the luminous matter.
Hence, two types of physical interactions must be modelled: *(i)* long-range forces (the gravity) that act at all distances, and *(ii)* local processes that act in the neighbourhood of every point.
The gravitational interaction, in turn, is calculated in three different "regimes of distance": the "close particles" are accounted for via a direct-summation approach, the "distant" particles are treated via a multi-polar expansion (basically, groups of particles that reside in a distant volume are treated as a unique equivalent particle). Finally, the contribution of "very distant" particles is calculated via a Particle-Mesh approach (the particles are distributed over a grid, and the resulting large-scale density field is used to solve the Poisson's equation via FFT; the gradient of the obtained gravitational potential returns the gravitational force from the large-scale distribution of matter).
Gravity and local processes need to retrieve the "neighbour particles" of any target particle, which is implemented through an Oct-Tree data structure that serves as a backbone for the entire code; the Oct-Tree provides a fast neighbour search with a search time of the order $O(log\, N_p)$ and $O(d)$ in the best (a reasonably homogeneous particles' distribution) and worst (a strongly clustered distributions) cases respectively, where N_p and d are the number of particles and the depth of the tree.
The OpenGADGET code is fully parallelized with a hybrid MPI+OpenMP approach and can scale reasonably well on very large number of nodes in typical x86-based HPC environments.

OpenGADGET's main loop is broken down into phases and sub-phases, whose execution times have been analysed in Deliverable 3.1, as follows:

1. Domain-Decomposition: since particles are moving in space under the effect of the physical forces acting on them, their spatial distribution continuously changes in the direction of a higher clustering; as a consequence, the distribution of particles among the MPI ranks needs to be constantly re-defined and refreshed to keep a good work-balance. The goal is achieved by defining the computational domains as segments of a space-filling Peano curve used to map to 1D the 3D particles' distribution.

2. The first kick to velocities: OpenGADGET implements a kick-drift-kick formulation of a simple leapfrog integrator to advance particle orbits in time, which coupled with a power-of-two decomposition of the timeline, leads to a symplectic integrator. This phase implements the first half-kick using the force estimated in the previous timestep.

3. Gravity; the forces due to the gravitational interaction are calculated using the TreePM approach explained above. In the following analysis we study the entire region (named "Gravity") and, separately, the calculation performed via the tree (labelled as "GravTree").

4. Density: the density of baryonic particles is estimated in an iterative loop where a suitable number of neighbours is found for every active particle.

5. Hydrodynamics: the hydrodynamical forces are calculated for all active particles with an SPH solver that uses the density as a weight function.

6. Extra-Physics: all the rest of the local physical processes are calculated. A partial list of those effects includes radiative cooling, star formation, stellar evolution and feedback, non-equilibrium chemistry, cosmic rays, dust production, magnetic fields, black-holes formation, dynamics and feedback.
7. Drift and second half-kick: the particles are displaced according to the estimated velocities, and the velocity is updated based on the new estimated forces (note: since the behaviour of this section and the first half-kick section are very similar, for the sake of clarity we show only this last region in the plots that are presented below).

2.8.1 Evaluation and Analysis Using Hardware Counters: Methodology

Since each phase of OpenGADGET behaves differently, it might be misleading - or even simply impossible - to analyse and try to optimise OpenGADGET as a whole. Instead, for the optimisation of OpenGADGET for the A64FX platform, we follow a "per phase" performance analysis. There are two targets in this effort of performance analysis and optimization of OpenGADGET: namely, the level of vectorisation of the code, and the proper utilization of HBM memory. This activity, in general, serves as an assessment of the mutual performance of both the codes and the platform. To identify performance optimisation opportunities, we have instrumented the OpenGADGET code with code that captures and reports hardware performance counters. To do this we have built from scratch a performance counter tracer, on top of the Performance Application Programming Interface (PAPI). For each optimisation target we select a set of performance counters.

Performance Counter Tracer

Analysing the performance of such complex code as *GADGET* requires tracing that allows temporal and spatial correlation of each measurement and its cause. Additionally different sets of performance counters are required for each type of analysis, so the events should be easily set and changed. Therefore we decided we needed to build a tracer, that will enable us to collect the performance counter events, in a manner that fulfils the requirements of our analysis. Our tracer, which we will refer to as *pmu tracer*, is built on top of Performance Application Programming Interface (PAPI) and it is built as a separate object file that is linked to *GADGET*. The main goals in the design of the *pmu tracer* are low memory footprint, ease of use and configurability. In order to keep the footprint of the tracer low, we collect aggregate counts among all threads, per iteration. By aggregating the counts of all threads, we manage to fit the data for one iteration in a single cache line. Additionally, *pmu tracer* operates in a lock-less way, with the exception being the initialization, which happens once per execution. This way, the instrumentation doesn't interfere with the control flow and the synchronisation of the application threads. The main constructs of *pmu tracer* are:

- event sets
- tracepoints
- tracefiles

An event set is a collection of events which are supported on the A64FX. A tracepoint is a segment of the code for which we want to collect performance counter events. a tracepoint is marked by a start and a stop call as such:

```
main_loop()
{
/*
...
*/
      pmu_tracepoint_start(0);
      domain_decomposition_intensity_execute();
      pmu_tracepoint_stop(0);
      pmu_tracepoint_start(1);
      compute_grav_accelerations();
      pmu_tracepoint_stop(1);
/*
...
*/
}
```

Listing 2.9: example of instrumented code

The aggregate counts for each event, in the event set per iteration for each tracepoint, is written in the corresponding tracefile in raw, binary format. To evaluate the overhead of the tracer we run the same configuration of *GADGET* several times with and without tracing, using 5 tracepoints. We found that, overall, the overhead of the pmu tracer is roughly 10%. Finally, we have created a post-processor utility that translates the binary trace files into csv files.

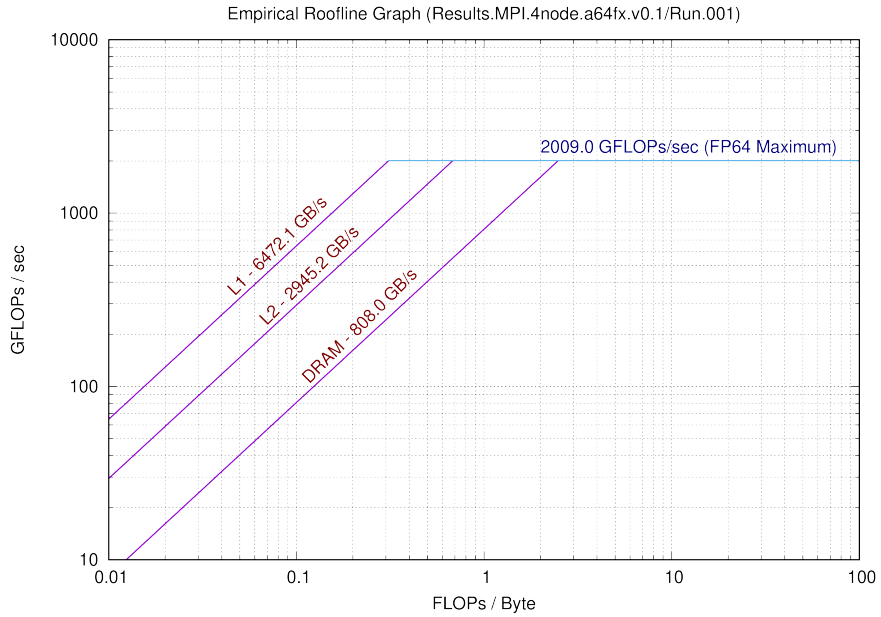
2.8.2 Evaluation and Analysis using hardware counters: experimental findings

Roofline Analysis

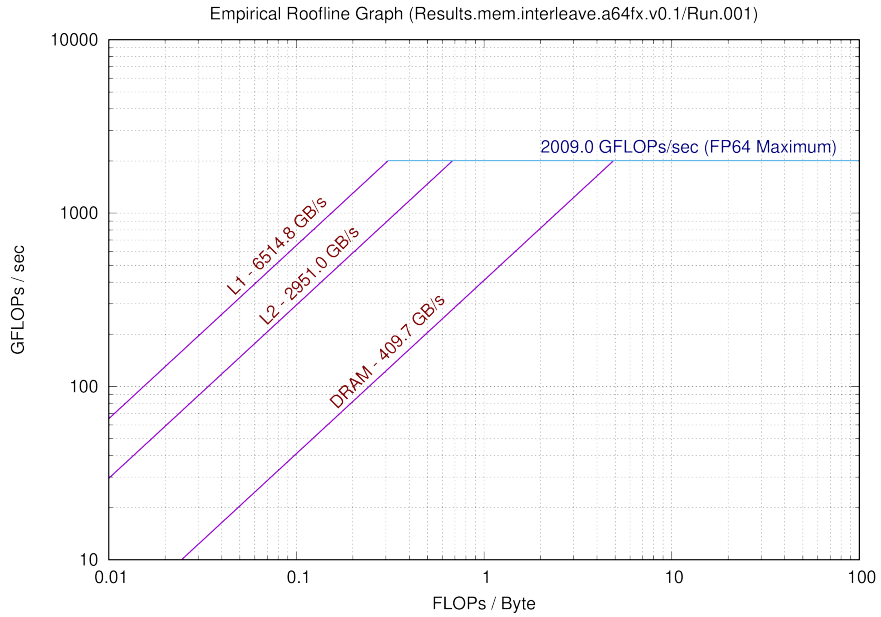
In order to characterise each phase of the execution of OpenGadget, we use a roofline model [52] analysis. To do this, we combine the output of the Empirical Roofline Tool (ERT) [53] with the Hardware counter output we get from the instrumented OpenGADGET code.

As first, we created two ERT configurations: the first ERT configuration creates a single process and populates all the cores of the CPU with 48 OpenMP threads. This configuration uses the *interleave* policy of *numactl* in order to allow the threads to utilise all the memory channels. The second ERT configuration creates 4 MPI ranks spawning 12 OpenMP threads. The MPI rank and its spawned threads are pinned on one of the 4 NUMA nodes of the CPU, and they access only their local memory, which means that their memory accesses do not cross NUMA boundaries. The results of the two configurations can be seen in Figure 44. The configuration with the four MPI ranks corresponds to Figure 44a while the ERT output for the configuration that uses a single MPI rank is shown in Figure 44b. Although both configurations achieve the same peak performance, we observe that the configuration with the four ranks achieves peak performance for a much lower (roughly half) arithmetic intensity. This is because in the four-rank configuration, each MPI rank only accesses memory that belongs to the NUMA node it is pinned on, hence achieving higher throughput.

When running *GADGET*, we set the affinity of threads and the memory policy similarly to the *four-rank* ERT configuration; This ERT configuration creates ideal, but unrealistic memory access patterns (no cross NUMA accesses etc); yet such memory access patterns are far from the ones *GADGET* creates. Therefore, we use the *one rank* ERT configuration as basis for our analysis.



(a) Four MPI ranks



(b) One MPI rank

Figure 44: Empirical Roofline Tool output for A64FX for two access patterns

execution phase	Stall Ratio	Memory-exclusive Stall Ratio
hydro	0.545	0.292
density	0.515	0.325
gravacc	0.542	0.286

Table 7: Stall ratios

In this roofline part of our analysis, we focus on three phases of *GADGET*, namely *hydro, density* and *gravacc*. In this analysis we try to give both the empirical results as measured, as well as a composite metric that takes into account structural constraints of the machine. For the FLOPs/sec and the Arithmetic Intensity, we use the performance counters along with the elapsed max time (per iteration) as shown in Table 8. In addition we use the stall cycles to create a projection of the performance had the system not been stalled due to reasons other than memory. In Table 7 we report the stall ratios for the 3 phases we examine in this section.

With $FLOPs_{measured}$ being the FLOPs estimated using the performance counters as described in Table 8, we can now state

$$FLOPs_{measured} = FLOPs_{ideal} * (1 - StallRatio)$$

from which we can derive

$$FLOPs_{ideal} = FLOPs_{measured} * (1 - StallRatio)^{-1}$$

The measured FLOPs as well as the projected FLOPs are shown in combination with the roofline curve in Figure 45.

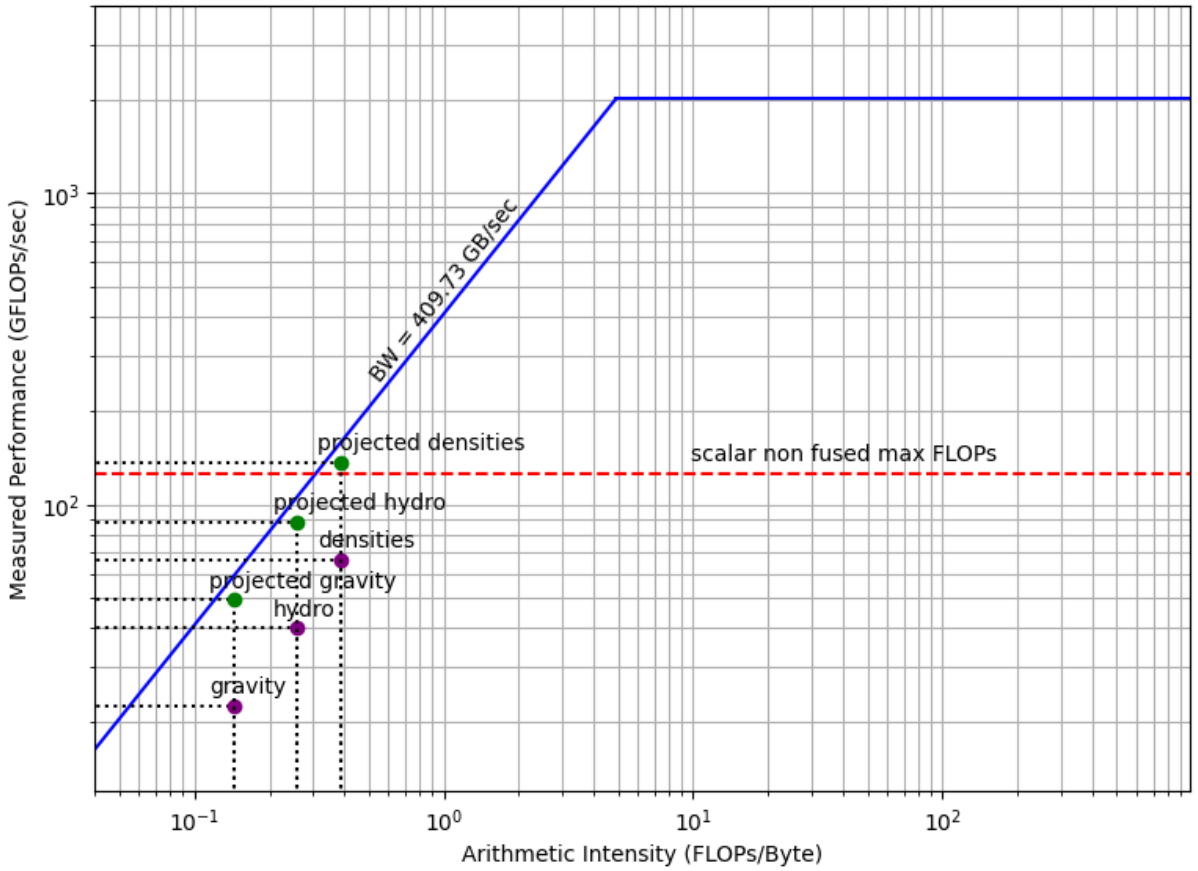


Figure 45: Plot of *OpenGADGET*'s phases on the A64FX *roofline.* Purple dots refer to the measured performance, while green dots are the rescaling of the purple ones by the inverse of the fraction of stalled cycles (see text for details). In other words, the green points represent the potential improvement after removing some general code inefficiencies.

Per-phase Code Analysis: Methodology

To assess the performance of the code on the target platform, we addressed all the phases listed above in the introduction to the code's structure. Since the behaviour of the two half-kick phases is similar, we show only one of them in the following. Also, in addition to profiling the whole gravity section, which includes the PM- and the tree-based calculations, we separately profile the pure tree-based phase.

For each code region, we have run the profiler with several different sets of events, aiming to extract various metrics. In Table 8, we illustrate how we have defined the adopted metrics: on the first column, we report the label with which we reference the metrics both in the following text and the figures. On the central column, we define the metric in terms of architectural events defined in the manuals mentioned above. Finally, we illustrate the precise meaning of the metrics and the rationale behind the adoption.

We stress that there is not a straightforward definition of the "FP vectorization ratio" as one would expect it, i.e. as FP_VECTOR_OPS / ALL_FP_OPS. That stems from 2 combined facts, namely:

1. the SVE operations have no fixed multiplicity, at odds with SIMD operations. In other terms, the SVE registers can be used by chunks of 128 bits; as such, every SVE instruction may be equivalent to a different number of scalar instructions.
2. while the FP ops are counted separately for SVE instructions (with the correct amount of equivalent single ops, i.e. by multiplying each instruction per 128 / type_size), the SIMD ops are counted together with the pure scalar ops.

As a consequence, we have defined 2 metrics that we use to estimate upper- and lower- bound for the vectorization ratio, namely the VECTORIZAITON RATIO and SVE+SIMD FRACTION, and an additional metric, the VECTORIZATION DEGREE, that we use as a general indicator of the prevalence of vector FP operations.

In addition, as explained in paragraph 2.8.2, we predict how well the code will perform in various regions using the platform's roofline model. Coupling this evaluation with the analysis of the number of stalled cycles, we can estimate the potential impact of different actions and determine whether they are feasible and/or advisable within the current code's framework.

Per-phase Code Analysis: Main Results

As we detail in the next sections, for the most suited kernels the auto-vectorization achieved by the FUJITSU compiler is surprisingly good, with an estimated vectorization fraction as high as 70%. In more complex kernels the vectorization is capped by several factors that can just be optimized specifically for A64FX but require a number of more general activities in the code.

Our main findings are as follows:

1. The performance porting of the OpenGadget code is reasonably successful (see the following discussion for a quantitative definition) on the target platform once:
   - the appropriate compiler (we used the `fujitsu` software stack) and flags are adopted. Specifically, we use `-O3 -fopenmp -KA64FX -KSVE -KARMV8_3_A -Kfast` throughout this work; see section 2.8.2 for a comparison with the GNU compiler;
   - the appropriate run set-up for threads placement and memory binding is specified. In this work, we limit the run to the usage of a single node; as such, given the fact that the A64FX architecture exposes 4 NUMA nodes per socket, we opted for running 1 MPI task per NUMA node while filling all the node cores with OpenMP threads. The rationale behind the choice is to fully exploit the computational power offered by the socket and, at the same time, to minimize the inefficient memory accesses. The OpenMP set-up is:

     ```
     export OMP_PLACES=sockets
     export OMP_BIND=close
     export OMP_NUM_THREADS=12.
     ```

Label	Definition	Meaning
SVE+SIMD ins. fraction	$\frac{\text{SVE\_INST\_RETIRED+SIMD\_INST\_RETIRED}}{\text{INST\_RET}}$	The fraction of all retired SVE+SIMD instructions respect to *all* retired instructions. This metric's rationale is that when a code is vectorized, not only the pure FP instructions are vector ones but also loads, stores, moves, etc. As such, we consider this ratio a general indicator of issuing vector instructions in a given code region.
IPC	$\frac{\text{CPU\_CYCLES}}{\text{INST\_RETIRED}}$	That is the usual definition of cycles-per-instructions, which is a broad estimate for the code's efficiency
SVE Vectorization ratio	$\frac{\text{FP\_SCALE\_OPS}\times 4}{\text{FP\_SCALE\_OPS}\times 4+\text{FP\_FIXED\_OPS}}$	The SVE+SIMD instructions are accounted by their true multiplicity, which however is not fixed in the SVE instructions because the SVE registers can be used by "modules" of 128bits. This metric assumes an average utilization of half the registers (4 doubles) to get an upper-bound estimate of the prevalence of pure SVE ops. We stress that this metric does not account for the SIMD ops that are counted in the FP_FIXED_OPS event along with the scalar FP ops.
Vectorization degree	$\frac{\text{FP\_SCALE\_OPS}\times 4+\text{FP\_FIXED\_OPS\_SPEC}}{\text{FP\_SPEC}}$	This metric amounts to estimate how many "vector" FP ops are performed per FP op issued. This number is expected to be 16 in the extreme case in which all the FP ops are SVE FMA with 8 doubles, 8 when all the FP ops are SVE, and 1 when all the FP ops are scalar. While assuming full utilization of registers would be more optimistic, we opted for assuming an average of half-register with FMA (from which it descends the factor 8).

Table 8: The definition of adopted metrics as functions of architectural events on ARM64FX

Label	Definition	Meaning
L[1|2] miss rate	$\frac{\text{L1D\_CACHE\_REFILL}}{\text{EFFECTIVE\_INST\_SPEC}}$	Following the definition from the manuals, this returns the fraction of instructions that lead to a refill of L[1|2] data cache.
L[1|2] misses	$\frac{\text{L1\_MISS\_WAIT}}{\text{CPU}_\text{C}\text{YCLES}}$	this returns the average number of L[1|2] data cache refill per cycle.
L[1|2] miss wait	$\frac{\text{L1D\_CACHE\_REFILL}}{\text{L1\_MISS\_WAIT}}$	this returns the average cost in cycles of data cache refill.
L[1|2] TLB miss rate	$\frac{\text{L1D\_TLB\_REFILL}}{\text{EFFECTIVE\_INST\_SPEC}}$	this returns the fraction of instructions that lead to a refill of L[1|2] TLB cache.
FP, loads, stores, branch fraction	$\frac{\text{X}}{\text{EFFECTIVE\_INST\_SPEC}}$	X = [FP_SPEC | LD_SPEC | ST_SPEC | BR_PRED]

Table 9: The definition of adopted metrics as functions of architectural events on ARM64FX [continues from the previous table]

Label	Definition	Meaning
Measured FLOPs	$\text{FP\_SCALE\_OPS} \times 4 + \text{FP\_FIXED\_OPS}$	Measured (double) Floating point operations.
Stalled cycles	$\text{STALL\_FRONTEND} + \text{STALL\_BACKEND}$	Every cycle that no operation was issued for any reason.
Stall ratio	$\frac{\text{STALL\_FRONTEND+STALL\_BACKEND}}{\text{CPU\_CYCLES}}$	the ratio of stalled cycles to the total number of cycles
Front-end stalled cycles	STALL_FRONTEND	Every cycle that no operation was issued because there are no operations available to issue from the frontend.
Front-end stall ratio	$\frac{\text{STALL\_FRONTEND}}{\text{CPU\_CYCLES}}$	the ratio of stalled cycles, due to front-end, to the total number of cycles
Back-end stalled cycles	STALL_BACKEND	Every cycle that no operation was issued because the backend is unable to accept any operations.
Back-end stall ratio	$\frac{\text{STALL\_BACKEND}}{\text{CPU\_CYCLES}}$	the ratio of stalled cycles, due to back-end, to the total number of cycles
Memory stalled cycles	LD_COMP_WAIT	Every cycle that no operation was committed because because the oldest and uncommitted load/store/prefetch operation waits for L1D cache, L2 cache and memory access.

Table 10: The definition of adopted metrics as functions of architectural events on ARM64FX [continues from the previous table]

The `mpirun` set up is:

```
ccc_mprun -e "-bind-to-numa -map-by numa -np ${SLURM_NTASKS}" \
numactl -localalloc
```

with the addition of the flags `-E"-exact -exclusive -enable_perf"` to access the `papi` framework and the `pmu`.

2. Inspecting the profiling data as defined above in Tables 8-10, we can conclude that the code has some overall inefficiencies that are not "local" but ascribable to the memory layout, especially, we speculate, to the large size (typically of the orders of hundred of bytes) of data structures that retain the particles' variables (namely the mass, the position, the velocity, the accelerations, to name some, and other physical quantities peculiar to different particles types).
   For instance, in Table 7, we report the measured average number of stalled cycles in different code regions; noteworthy, that figure is stable all across the run and among the various regions (that exploit different data structures).
   The analysis conveys that about *half of the cycles* are stalled in all those regions (similar results hold for all the regions), and the $\sim$ 40% of the stalls are due to memory starvation. Indeed, from the estimate of the arithmetic intensity, we know that the code resides in the ramp area of the roofline, which, by definition, is where the platform under analysis is memory-bound. However, our code's placement well below the ramp line suggests that the inefficiencies revealed by the analysis further limit the performance, leaving a significant potential for improvement. Projecting a $\times$ 2 factor on the roofline (green point in Figure 45) shows that this would bring the code's performance in line with the saturation in the memory-bound area.

Discussion of The Results Obtained With The FUJITSU Compiler

This section briefly reviews and describes the details of our tuning and analysis; the following plots report measurements collected via our PAPI-based framework to collect performance counters during the code execution. All the plots report the data from the regions defined at the beginning of this section 2.8, colour-coded as indicated in the legends.
We have run a small cosmological box of $15Mpc/h$ size and 32^3 particles from early times ($t = 0$) down to the present ($t = 1$). Our choice was due to the fact that such a small case could easily fit in a single node and run in a short time, allowing us to perform multiple tests and collect the data along the entire simulation. In the plots, the solid lines represent the measured values averaged over bunches of 10 successive measures. At the same time, the shaded areas encompass the region within $\langle v \rangle \pm \sigma$ where σ is their standard deviation.

Vectorization Figure 46 shows our results for collecting `pmu`-based metrics on issuing vector instructions within various code regions. The regions have been described in the introduction to this chapter. In the "vector" set, we include both the SVE and the SIMD sets. To trace the related instructions and operations, we rely on the counters described in Table 11.

In the left panel of the upper row, we show the estimate of the upper limit for the SVE vectorization fraction, which, with somehow large oscillations, lies at 70% for most regions, while reaching $\sim 80\%$ and even $\sim 100\%$ for the ExtraPhysics and the Density regions. This estimate, which does not include the SIMD flop, assumes that the SVE registers are always used at full size and with no FMA.

event	Definition	comment
FP_SCALE_OPS_SPEC	"This event counts architecturally executed SVE arithmetic operations; This event counter is incremented by (128/CSIZE) and by twice that amount for operations that would also be counted by SVE_FP_FMA_SPEC"	We interpret this counter to be incremented by 2 in case we use double precision types (as it is actually stated in the description of the dedicated FP_DP_SCALE_OPS_SPEC, and by 4 if the operation consists in a FMA. The SVE ins can be executed with a variable number of elements in the registers, which are 512bits wide. Since we have set the code to use only double precision, the maximum possible multiplicity is 8. Taking into account the 2 factor already accounted for, we then multiply this counter by a factor of 4 to have a conservative **upper-limit** of the prevalence of pure SVE operations (neglecting the presence of FMA ops).
FP_FIXED_OPS_SPEC	"This event counts architecturally executed v8SIMD&FP arithmetic operations; This event counter is incremented by the specified number of elements for Advanced SIMD operations or by 1 for scalar operations, and by twice those amounts for operations that would also be counted by SVE_FP_FMA_SPEC".	This counter is incremented by the true number of FP ops in SIMD and normal registers.
SIMD_INST_RET SVE_INST_RET	"This event counts architecturally executed [SIMD|SVE] instructions".	This counters returns *all* the [SIMD|SVE] instructions, which includes not only purely floating-point but also loads, stores, byte reshuffling etc.
INST_RETIRED	"This event counts every architecturally executed instruction."	–
CPU_CYCLES	"This event counts every cycle."	–

Table 11: The architectural events on ARM64FX relevant to determine the vectorization

Compiler	Options
fujitsu	-O3 -fopenmp -KA64FX -KSVE -KARMV8_3_A -Kfast
gnu	-Ofast -O3 -fopenmp -march=armv8.3-a+sve+simd -msve-vector-bits=512 -param aarch64-autovec-preference=4 -param aarch64-vect-unroll-limit=8 -fno-stack-protector -ftree-vectorize -fomit-frame-pointer

Table 12: The options used to tune the compilation.

The right panel of the same row shows the metrics that we call "vectorization degree", defined as the number of flops executed per flop instruction. If all the flops were pure SVE(SIMD), that figure would range from 8(4) to 16(8) for `double precision` operands; hence, how this metrics is larger than 1 is related to how many flop operations are conducted via vector registers.
We note that the relative behaviour of the different regions is the same for the two metrics, as expected. The fact that the DomainDecomposition region has lower values for the first metrics may be due to the fact that the compiler has given preference to SIMD instructions there. The left panel of the bottom row shows a lower limit for the vectorization fraction, defined as the ratio between the summation of all the **SVE** *and* SIMD *instructions* and all the instructions. The rationale behind this choice is that a successful flop vectorization does not come without ancillary operations like loads, stores, memory operations and others. We then consider this metric as a lower limit for the flop vectorization.
Finally, the bottom right panel displays the Instructions-per-Cycle metric; values below 2 are usually not considered optimal ones; in the presence of a significant amount of vectorization, they may be seen as a sufficient achievement (see Sec. 2.8.2 below). However, the medium value obtained is reflected by the poor placement in the roofline chart (see discussion in section 2.8.2 and Table 7).

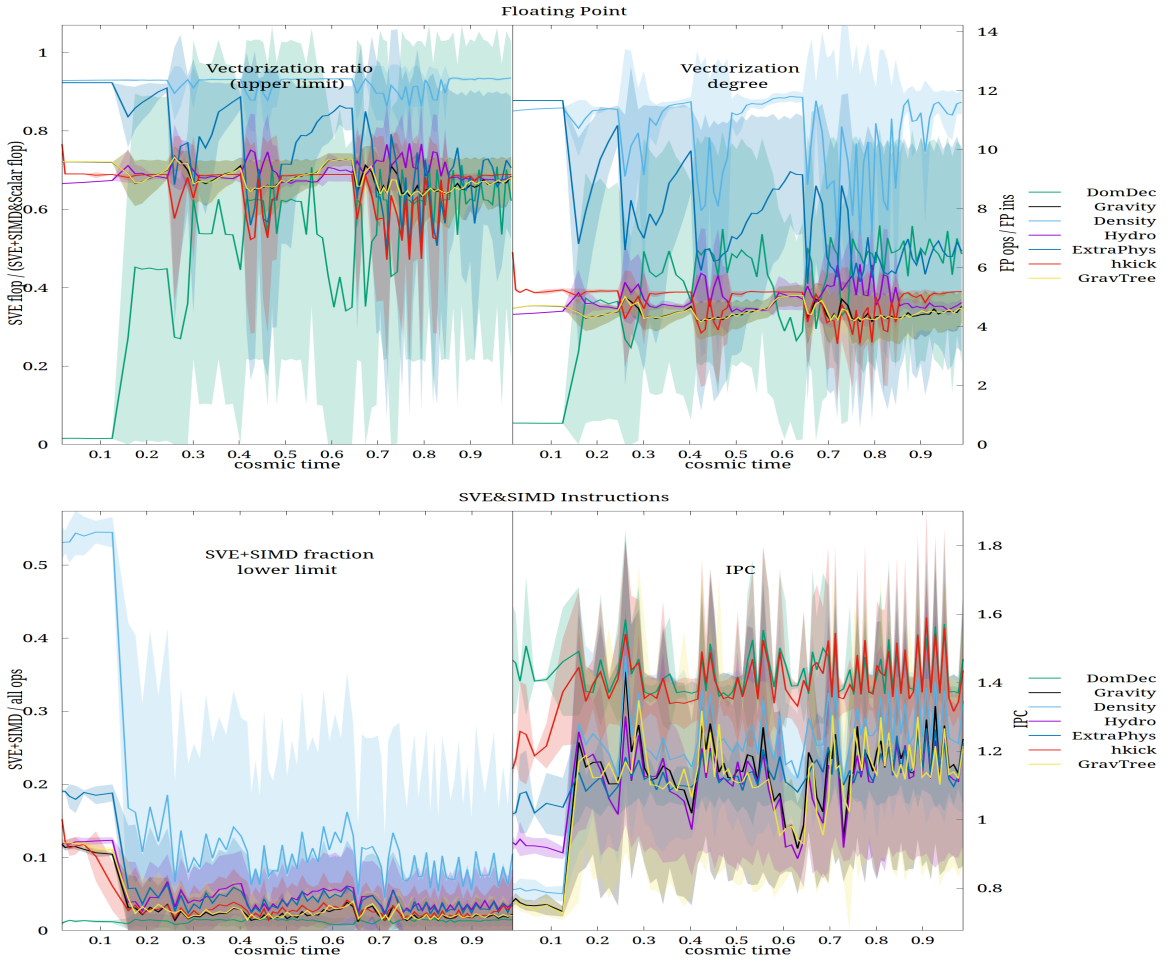


Figure 46: Estimates of the vectorization ratio in different code regions as a function of the run time (*x-axis*). **Top row.** In the *left panel*, we show the upper limit for how many SVE flop ops are issued in units of the total number of flop ops; in other words, we show the fraction of flop operations that could be interpreted as SVE. In the *right panel*, we report the "vectorization degree" defined as how many flop ops are performed per flop instruction (note that this figure would be 16 for full utilization of SVE registers with doubles and FMA and equal to 8 without FMA). **Bottom row**. Left panel: the bottom limit of SVE+SIMD flop ops fraction; right panel: the average Instructions-per-Cycle. See the text for more details and a discussion.

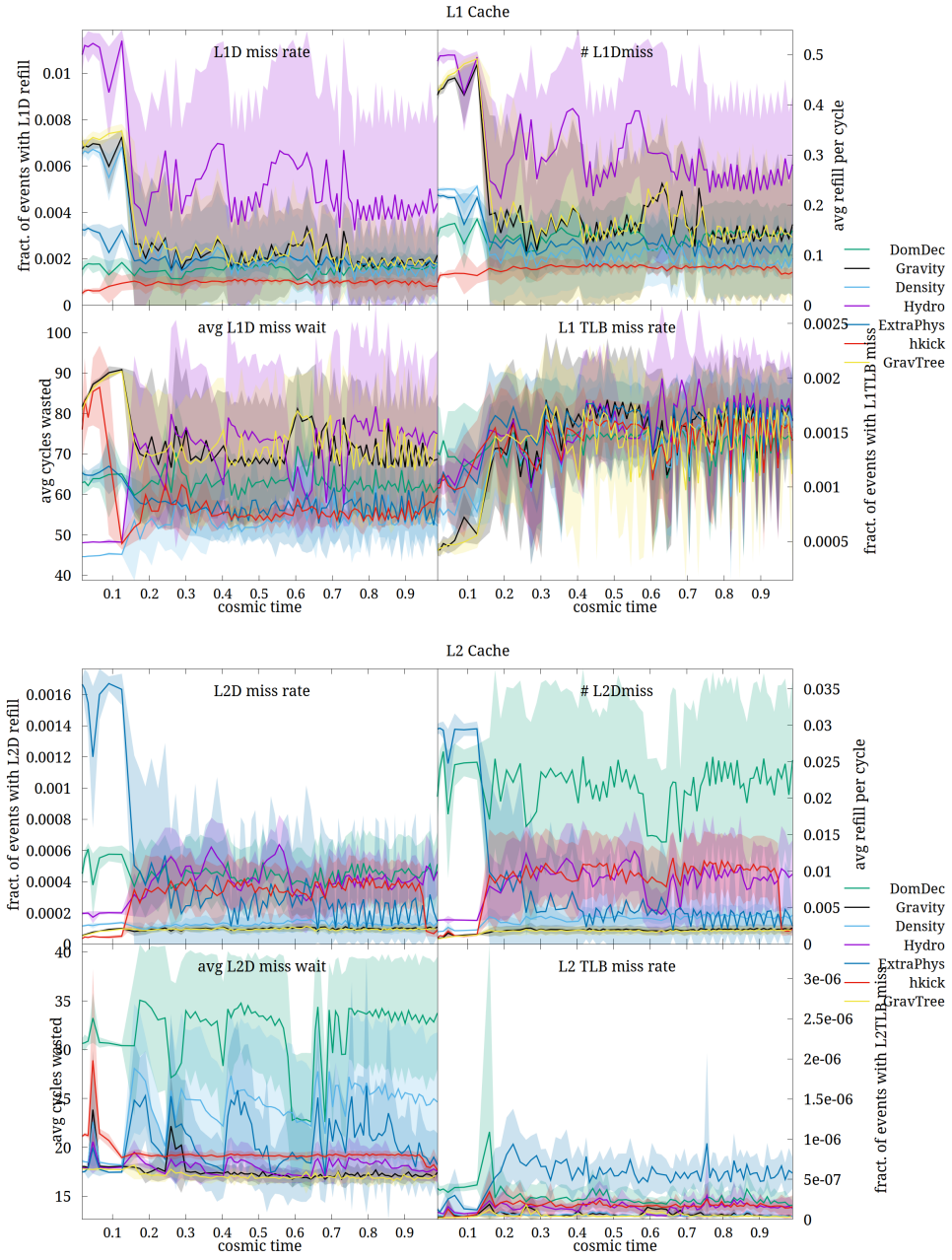


Figure 47: Metrics for the usage of L1 (top 4 panels) and L2 (bottom 4 panels) caches. **Top row of each figure**. *Left panel*: the fraction of events that led to a L[1:2]-Data refill. *Right panel*: the actual number of miss in terms of avg cache refill per cycle. **Bottom row of each figure**. *Left panel*: how many cycles were wasted in average per cache refill. *Right panel*: the fraction of events that led to a L[1:2]TLB miss.

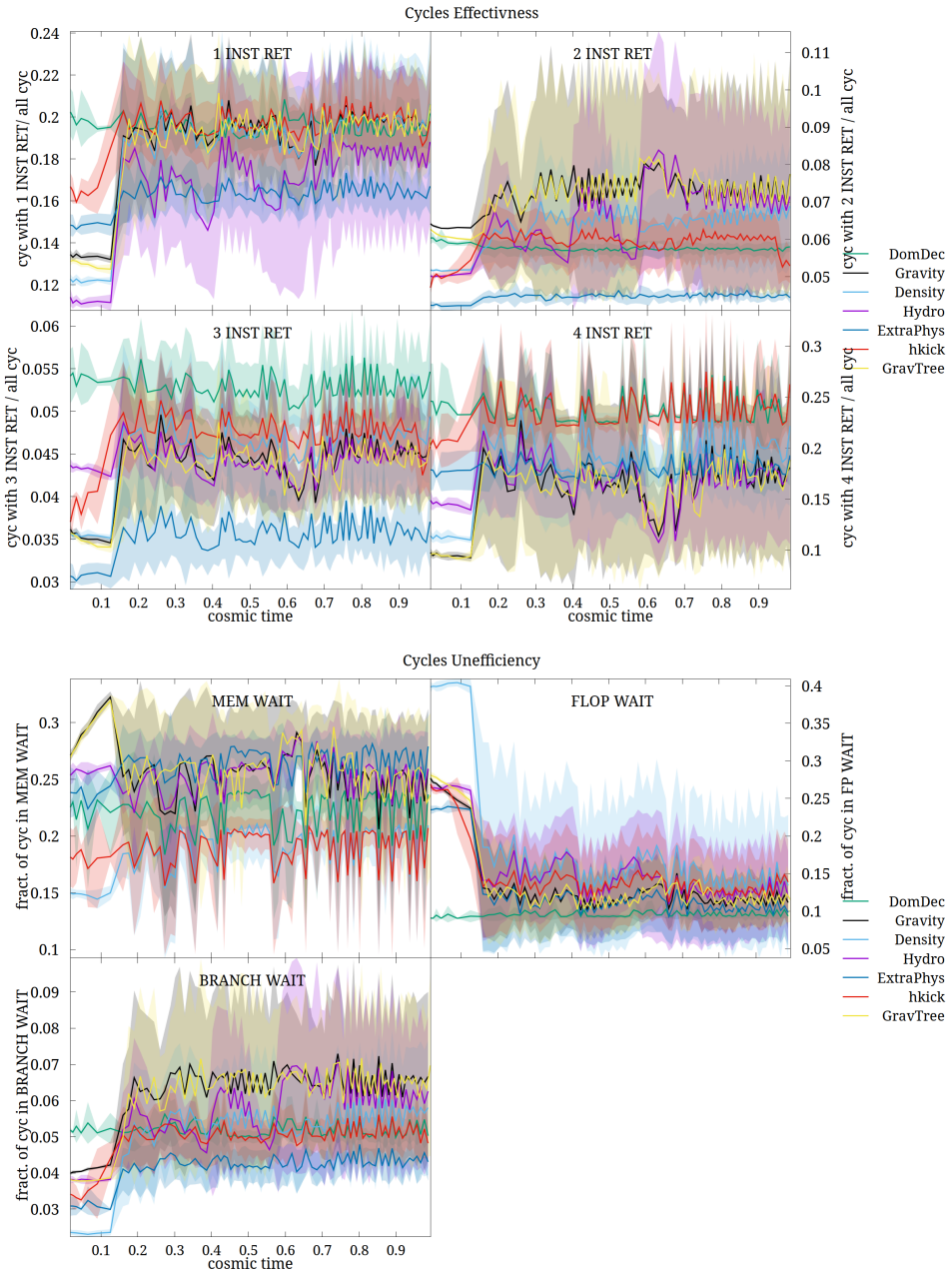


Figure 48: **Top figure** Metrics for IPC effectiveness: the fraction of cycles in which 1,2, 3 and 4 instructions were retired (from top left to bottom right). **Bottom figure** Metrics for some inefficiencies.

Figure 47 condenses our findings about the L1 (upper sub-figure) and L2 (bottom sub-figure) caches. Top rows, left and right panels, contain the average fraction of events that led to cache refill miss rates and the number of cache refills per cycle, respectively. The bottom rows report the average cost of a refill (in cycles) and the average fraction of events that led to a TLB refill (left and right panels, respectively). The L2 cache exhibits good behaviour in all the metrics; although that is partially due to the very small size of the case under exam, that is a strong point in favour of the A64FX architecture, the L1 suffers, on average, from many refills per cycle. Given the large size of the L1 in the target architecture, we interpret that as a consequence of the large size of the data structures that contain each particle's data.

Finally, to conclude this analysis, we inspected some broad indicators of the code's efficiency and inefficiency: the fraction of cycles in which one, two, three or four instructions are retired and the fraction of cycles with no instructions issued due to the waiting for a previous event, in the top and bottom sub-figures of Figure 48 respectively.
It can be appreciated that a significant fraction of cycles, from ~ 0.2 to ~ 0.4 appear to be very efficient, with 4 instructions retired. However, almost the same number of cycles only amount to 1 instruction and a noteworthy fraction falls in the group of stalled cycles (see the discussion about the stalled cycles analysis).
In fact, about 30% to 40% of cycles is stalled either on a memory wait or on a flop instruction, that is obviously in accordance with the figures in Table 7.

Comparison Between GNU and FUJITSU Compilers

To bring evidence of the crucial role of the compiler, we have conducted the same set of simulations compiling the code with a GNU-based software stack. However, in this section we limit the discussion to the metrics related to vectorization.
Figure. 49 collects the direct comparison between the results obtained with the two compilers: the FUJITSU and the GNU in the left and right columns of each pair of plots, respectively.
Table 12 reports the precise flags used to tune the compilation in both cases. Panels 49a, 49b and 49c directly contrast *(i)* the upper-limit estimate for the vectorization ratio, *(ii)* the vectorization degree and *(iii)* the SVE+SIMD instructions fraction. The plots speak for themselves (even: dropping the +sve specifier in the GNU string would result in no SVE at all). As expected, the GNU compiler is much worse at delivering efficient vectorized code on the A64FX architecture, while the average IPC results look to be slightly better. However, with a very low vectorization, that is far from sufficient.
In fact, the total run-time of the GNU code is $\sim 30\%$ larger than the one achieved with the FUJITSU compiler: 217 sec instead of 164 sec.

2.8.3 Conclusion

In the sections of this chapter we describe our performance-port of OpenGadget on the A64FX CPU. We present our methodology for analysing the performance of OpenGadget on the A64FX and we report our findings. We use the Empirical Roofline curves to present the measured performance of OpenGadget in context of an ideal kernel (i.e. the tight kernel that the ERT is using) with similar arithmetic intensities. We also report time-series of several metrics, such as vectorisation fractions, cache misses, IPC, etc. Our findings show that we have managed to achieve a high degree of

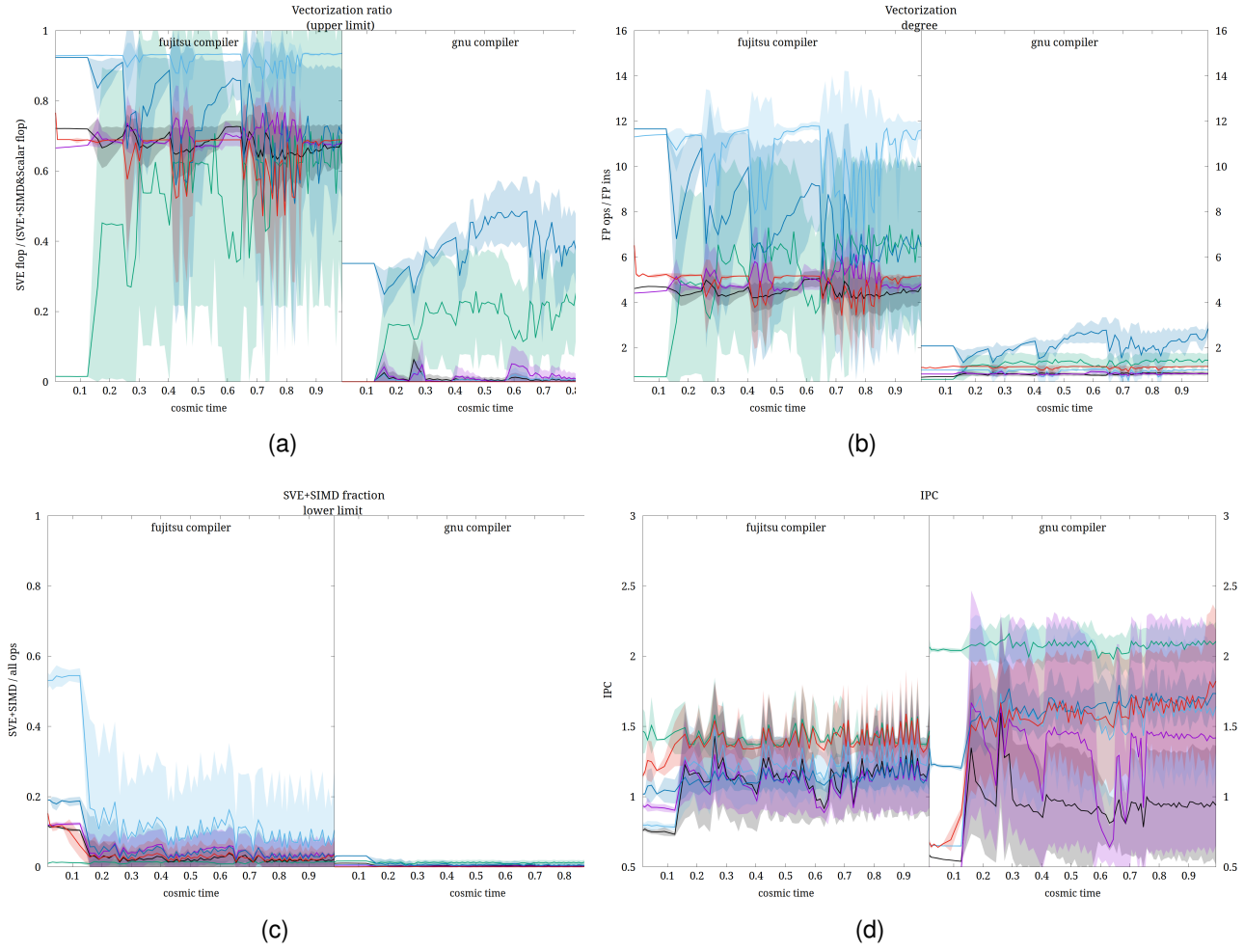


Figure 49: Comparison between results obtained with FUJITSU (left columns of each panel) and GNU compilers. The same colour code for the different regions applies than for plots in the previous section.

vectorisation, especially for some phases of OpenGadget and that we are utilising well all the micro-architectural components of the CPU (memory channels, vector units, cores, etc).

Despite the high degree of vectorisation and the seemingly proper orchestration of the run (task placement, NUMA-aware allocations etc.), we measure lower performance than what the roofline model predicts. The results we present indicate that the difference between the measured and the ideal performance can be attributed to the large number of cycles in which the pipeline is stalled. We believe the cause for those stalls to be a combination of the irregularity of accesses of OpenGadget, with some micro-architectural design limitations of the A64FX, such as small TLBs, HBM only memories, lack of L3 caches etc. Finally, we believe that further optimisations for the A64FX require large scale structural changes of OpenGadget code which we believe are outside of the scope of this effort.

3 Tailoring EUPEX Mini-applications To SVE and HBM

In this section, we detail the analysis done on the mini-applications in order to make use of the Scalable Vector Extension (SVE) instruction set and the High Bandwidth Memory (HBM) features that will be present in the EUPEX prototype.

3.1 Bolt65

Bolt65 is a performance-optimized HEVC (High Efficiency Video Coding) hardware/software suite for Just-in-Time video processing. It is a "clean-room" suite that consists of an encoder, decoder, and transcoder based on the HEVC standard [54]. The special focus in the development of the Bolt65 is set on the performance efficiency achieved by low-level optimizations and hardware-software co-design adapted for the efficient exploitation of different underlying hardware architectures. As described in previous chapters, the main focus for the application optimization in this phase of the project is the utilization of SVE vector extensions and High-bandwidth memory (HBM) that will be available on the RHEA chip.

3.1.1 Optimizations for SVE

In the context of optimizing the Bolt65 application through the utilization of ARM SVE vector extensions, we examined and tested various vectorized implementations of the application. This included assessing auto-vectorization, manual vectorization, and a hybrid approach that combines both methodologies. The objective was to compare the performance benefits of three different implementations and try to find the optimal trade-off between the investment of effort required to adapt the application source code and the resulting performance gains.

Auto-vectorization

Most compilers will try to automatically vectorize code without any additional input from the user. Therefore, the auto-vectorization implementation does not require any changes to the source code. This implementation requires minimal effort from the developers, but the efficiency of the application optimization is entirely dependent on the compiler's ability to identify and vectorize part of the source code.

Manual Vectorization

In a manually vectorized implementation, developers are tasked with rewriting specific segments or the entirety of the source code, employing dedicated SVE intrinsic to utilize the processor's vectorization capabilities. While this approach demands a specialized understanding of the application and its implementation, it offers the advantage of tailoring the code to the specific architecture of the platform, which can result in increased performance efficiency.

The primary challenge of this implementation is to identify the parts of the code that are suitable for manual vectorization. Completely rewriting the application using SVE intrinsics would be a challenging and time-consuming task. Consequently, we analyzed and profiled the application to identify key kernels and to pinpoint methods and algorithms that significantly contribute to the overall application runtime. For a more in-depth understanding, the findings of this profiling effort are outlined in the deliverable "D3.1 Application Analysis Report" [10].

In the Bolt65 manually vectorized codebase, we identified and implemented ten kernels using SVE intrinsics:

- Discrete Cosine Transform (DCT) and Inverse Discrete Cosine transform (IDCT)
- Quantization and De-quantization
- Sum of absolute differences (SAD)
- Residual substraction
- Intra-prediction kernels: 3-tap filter, Planar, DC, and Angular prediction

Figure 50 illustrates the high-level structure of the Bolt65 transcoder, featuring highlighted manually vectorized blocks. Modules highlighted in red indicate that all kernels within that module are manually vectorized, whereas orange modules signify that only some of the kernels have undergone manual vectorization.

To illustrate the transition from scalar to manually vectorized code using SVE intrinsics, the following code snippet provides two examples of kernels implemented with SVE: SAD and quantization.

```
1  //Scalar implementation
2  int Scalar::SAD(unsigned char* block_1, unsigned char* block_2, int puWidth, int
   ↪  puHeight)
3  {
4          int result = 0;
5          for (int i = 0; i < puHeight; ++i)
6          {
7                  for (int j = 0; j < puWidth; ++j)
8                  {
9                          result += abs(block_1[i * puWidth + j] - block_2[i * puWidth +
                           ↪  j]);
10                 }
11         }
12
13         return result;
14 }
```

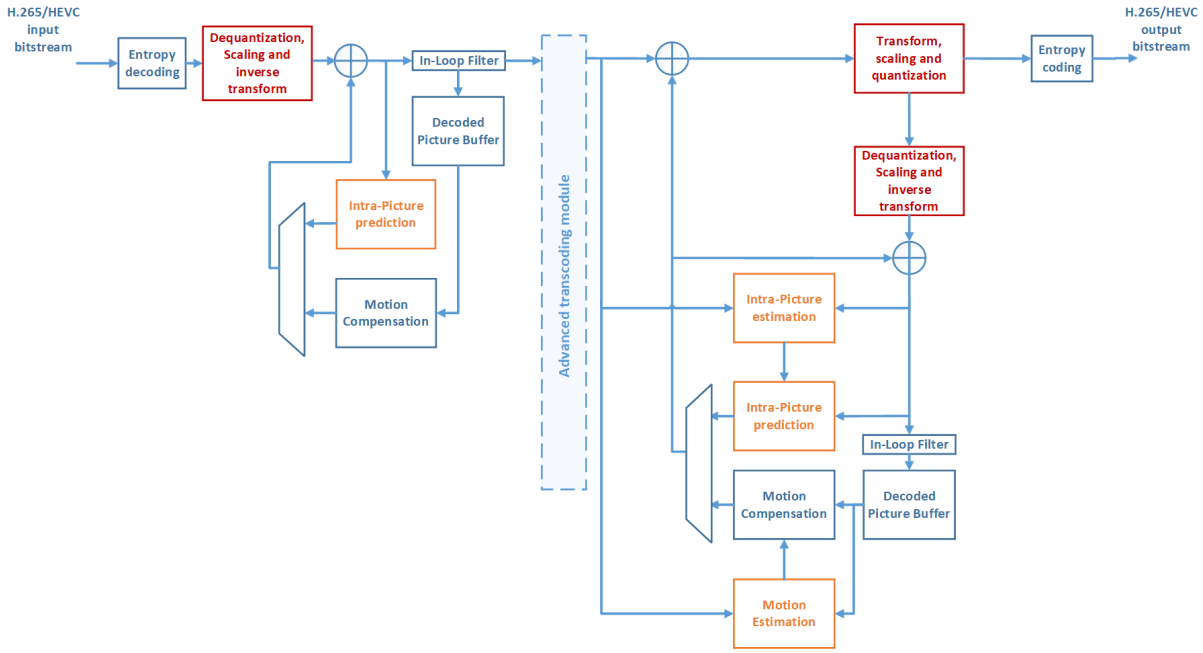


Figure 50: Top level scheme of Bolt65 transcoder with highlighted manually vectorized code

```
15 //Manually vectorized implementation
16 int SVE::SAD(unsigned char* block_1, unsigned char* block_2, int puWidth, int
   ↪  puHeight)
17 {
18         size_t svewidth = svcntb();
19         int maxNum = svewidth / sizeof(unsigned char);
20
21         int matrixSize = puHeight * puWidth;
22
23         if (matrixSize < maxNum)
24                 maxNum = matrixSize;
25
26         svbool_t predicate = svwhilelt_b8_s32(0, maxNum);
27
28         svuint8_t orig, pred, absdiff;
29         int64_t sum = 0;
30
31         for (int i = 0; i < matrixSize; i = i + maxNum)
32         {
33                 if (i + maxNum > matrixSize)
34                         predicate = svwhilelt_b8_s32(0, matrixSize - i);
35
36                 orig = svld1_u8(predicate, block_1 + i);
37                 pred = svld1_u8(predicate, block_2 + i);
38
39                 absdiff = svabd_u8_x(predicate, orig, pred);
40
41                 sum = sum + svaddv_u8(predicate, absdiff);
```

```
42          }
43
44          return (int)sum;
45  }
46
47
48  //Scalar implementation
49  void Scalar::quantization(int16_t *A, int QP, int NB, int N, bool* is_zero_matrix,
    ↪  int16_t* C)
50  {
51          int M = ComUtil::logarithm2(N);
52          int bdShift = 29 - M - NB;
53
54          int coeffMin = -32768;
55          int coeffMax = 32767;
56
57          int mScalingFactor = 16;
58          int i;
59          *is_zero_matrix = true;
60
61          for (i = 0; i < N; i++)
62          {
63                  for (int j = 0; j < N; j++)
64                  {
65                          int sign = 1;
66                          if (*(A + i * N + j) < 0)
67                                  sign = -1;
68
69                          *(C + i * N + j) = (((abs(*(A + i * N + j)) * f[QP % 6] + (1
                            ↪  << (bdShift                                    - 1))) >> QP
                            ↪  / 6) >> bdShift);
70                          *(C + i * N + j) = ComUtil::clip3(coeffMin, coeffMax, *(C + i
                            ↪  * N + j) *                                     sign);
71
72                          if (*(C + i * N + j) != 0)
73                                  *is_zero_matrix = false;
74                  }
75          }
76
77
78  }
79  //Manually vectorized implementation
80  void SVE::quantization(int16_t * A, int QP, int NB, int N, bool * is_zero_matrix,
    ↪  int16_t * C)
81  {
82          int M = ComUtil::logarithm2(N);
83          int bdShift = 29 - M - NB;
84          int mScalingFactor = 16;
85          *is_zero_matrix = true;
86          int matrixSize = N * N;
87
88          size_t svewidth = svcntb();
```

```
89          int maxNum = svewidth / sizeof(int32_t);
90          if (matrixSize < maxNum)
91                  maxNum = matrixSize;
92
93          svbool_t predicate = svwhilelt_b32_s32(0, maxNum);
94          svint32_t data, absdata;
95
96          svint32_t mult = svdup_n_s32(f[QP % 6]);
97          svint32_t shiftOffset = svdup_n_s32(1 << (bdShift - 1));
98
99          for (int i = 0; i < matrixSize; i = i + maxNum)
100         {
101                 data = svld1sh_s32(predicate, A + i);
102
103                 absdata = svabs_s32_x(predicate, data);
104
105                 absdata = svmul_s32_x(predicate, absdata, mult);
106                 absdata = svadd_s32_x(predicate, absdata, shiftOffset);
107                 absdata = svasr_n_s32_x(predicate, absdata, ((QP / 6) + bdShift));
108
109                 data = svneg_s32_m(absdata, svcmplt_n_s32(predicate, data, 0),
                    ↪  absdata);
110
111                 if (svptest_any(predicate, svcmpne_n_s32(predicate, data, 0)))
112                         *is_zero_matrix = false;
113
114                 svst1h_s32(predicate, C + i, data);
115         }
116 }
117
```

It's worth noting that the definitions of the scalar and manually vectorized methods are identical. The only difference is in its implementation. To exploit this fact, we established an interface that outlines all methods with differing implementations. Subsequently, specific classes (Scalar, SVE, etc.) were created to implement that interface. This design ensures a straightforward transition between manually vectorized and scalar implementations.

Combination of Manual and Auto-vectorization

In the manual implementation, we incorporated SVE intrinsics to implement ten key kernels, while the remaining code retained a conventional scalar approach. To explore different scenarios, we introduced a third implementation, in which we continued to employ manually vectorized code as previously described, but the rest of the application's code was auto-vectorized by the compiler.

Experimental Results

Use cases and test environment for conducting experiments on Bolt65 are described in more detail in the preceding deliverable "D3.1 Application Analysis Report" [10]. Since the initial experimen-

tation setup, we have decided to incorporate an additional Bolt65 encoder configuration into our experiments. In addition to the low-complexity and high-complexity benchmarks, we introduced the All-Intra benchmark. In this configuration, the Bolt65 encoder is set so that every frame in a video sequence is intra-coded. All-intra compression demands less processing power for both encoding and decoding video sequences. Furthermore, it exhibits increased resilience to errors as it doesn't rely on information from previous frames. However, coding efficiency is not as good as when using Inter prediction. Consequently, for the experiments, we have three different benchmarks based on the encoding configuration:

- Low-complex - high computational complexity benchmark (INTER and INTRA prediction)
- High-complex - low computational complexity benchmark (INTER and INTRA prediction)
- All-intra - all intra-frame coding

For each of the three benchmarks, we employ the same test set of video sequences, encompassing a spectrum of spatial resolutions and frame rates. In addition to the initially defined set of video sequences, we've introduced an extra FullHD sequence. The comprehensive list of 10 test video sequences is provided in the table 13:

	Video name	Resolution	Frames	Frame rate
1	Akiyo	176x144	300	30
2	Mobile	352x288	300	30
3	Soccer	704x576	600	60
4	City	704x576	600	60
5	Johnny	1280x720	600	50
6	ParkRun	1280x720	500	60
7	Shields	1280x720	500	60
8	BasketballDrive	1920x1080	500	50
9	BlueSky	1920x1080	217	25
10	RushHour	1920x1080	500	25

Table 13: Test video sequences

All experiments were conducted using two different compilers: ARM's armclang and Fujitsu's FCC. This approach allowed us to thoroughly explore and compare the vectorization capabilities of these particular compilers. It also provided insights into the distinctions between manual and auto-vectorization.

Compilation and Execution The Bolt65 codebase remains consistent across all mentioned implementations including scalar, auto-vectorization, manual vectorization, and a combination of manual and auto-vectorization. Therefore, there is no need for maintaining multiple versions of the code or introducing changes between these various implementations.

To determine which implementation to employ, we can simply set the appropriate flags during compilation and/or configure specific parameters during execution. The combination of compilation

flags and runtime parameters uniformly determines which implementation is being used. This adaptable approach, based on the specified compilation and runtime settings, allows for a seamless transition between different optimization strategies. Table 14 shows which flags have to be set in order to run a specific implementation.

	Compile time flags	Parameters during execution
Scalar	–fno-vectorize	
Manual vectorization	-march=armv8-a+sve –fno-vectorize	–sve
Auto-vectorization	-march=armv8-a+sve	
Auto and manual vectorization	-march=armv8-a+sve	–sve

Table 14: Implementation parameters

The compiler *"-fno-vectorize"* flag explicitly instructs the compiler not to use any form of automatic vectorization. It is valid for both compilers used in these experiments. During execution, the *"–sve"* parameter serves as an application-specific flag, dictating whether the application should utilize the regular scalar implementation or the manually vectorized one for key kernels. When the *"–sve"* flag is set, manual vectorization will be applied; otherwise, a scalar version of the methods will be utilized.

Results The primary objective of conducting these experiments was to observe the performance difference among different implementations and to investigate the influence of the compiler on the obtained results. Therefore, our initial measurements focused on collecting the runtimes of the Bolt65 encoder.

In Figure 51, the results of encoding all ten video sequences are presented across four different encoding configurations (scalar, auto, manual, and auto+manual), utilizing two different compilers (arm-clang and fcc). The graphs in the first column depict results for the armclang compiler, while those in the second column illustrate the outcomes obtained with the FCC compiler. Table 15 and 16 show the speed-up of auto, manual, and combined implementations compared to the scalar implementation without any optimizations.

The results indicate that auto-vectorization generally provides a speed-up compared to scalar implementation in almost all cases. For armclang, the average speedup of auto-vectorization is x1.16, x1.27, and x1.22 for all-intra, low-complex, and high-complex use cases respectively. In the case of FCC compiler, the average speedups are x1.17, x1.17, and x0.96. It's worth noting that for auto-vectorization in the high-complexity encoding configuration with FCC, the performance is even worse than the scalar version. All speedups are calculated as a ratio between scalar and vectorized implementation.

While the speed-ups from auto-vectorization are noticeable, they are not substantial. However, the introduction of manually vectorized code demonstrates fairly better speed-up. For armclang, the observed speed-up ranges from x1.84 to x2.50 on average, depending on the encoder configuration. Similarly, for FCC, the speed-ups vary from x1.69 to x2.44. Given that the combination of auto and manual vectorization implements the same manually vectorized kernels as pure manual vectorization, with the rest of the code being auto-vectorized, a modest performance improvement is expected compared to manual vectorization alone. For armclang, speedup of combined implementation compared to scalar ranges from x1.85 to x2.57, and for FCC, it varies from x1.70 to x2.52.

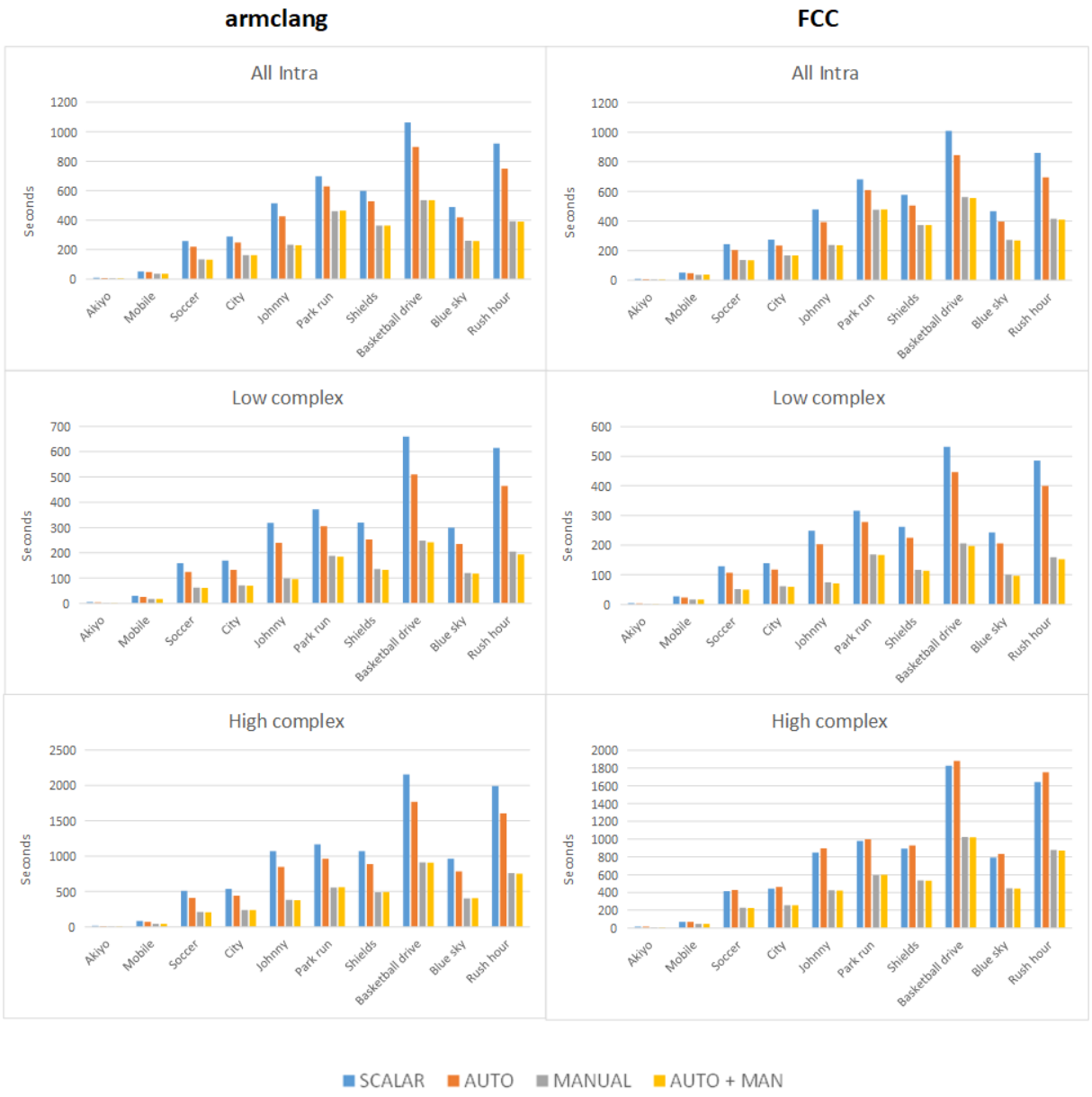


Figure 51: Performance comparison

To facilitate a more comprehensive comparison between two different compilers, we computed the overall time required to encode all video sequences for each configuration. The results are illustrated in Figure 52.

The comparison between Fujitsu's FCC compiler and ARM's armclang reveals that FCC is faster than armclang for the scalar implementation across all encoding configurations. In the case of auto-vectorization, FCC exhibits faster performance for the All-intra and Low-complex encoding configurations, although the gaps are narrower compared to the scalar implementation. However, for

Video	Intra			Low-complex			High-complex		
Name	**AUTO**	**MAN**	**COMB.**	**AUTO**	**MAN**	**COMB.**	**AUTO**	**MAN**	**COMB.**
Akiyo	x1.13	x1.68	x1.68	x1.27	x2.62	x2.70	x1.22	x2.57	x2.59
Mobile	x1.10	x1.43	x1.43	x1.17	x1.70	x1.70	x1.18	x1.90	x1.88
Soccer	x1.18	x1.93	x1.95	x1.29	x2.58	x2.64	x1.24	x2.44	x2.46
City	x1.16	x1.77	x1.78	x1.27	x2.38	x2.43	x1.22	x2.28	x2.28
Johnny	x1.21	x2.21	x2.24	x1.33	x3.23	x3.35	x1.26	x2.81	x2.85
Park run	x1.11	x1.51	x1.50	x1.22	x1.98	x2.01	x1.21	x2.09	x2.08
Shields	x1.13	x1.65	x1.65	x1.27	x2.36	x2.41	x1.21	x2.19	x2.18
Basketball D.	x1.18	x1.98	x1.98	x1.29	x2.66	x2.73	x1.22	x2.36	x2.37
Blue sky	x1.17	x1.89	x1.89	x1.28	x2.50	x2.55	x1.23	x2.40	x2.39
Rush hour	x1.23	x2.35	x2.37	x1.32	x3.01	x3.18	x1.24	x2.63	x2.65
Average	**x1.16**	**x1.84**	**x1.85**	**x1.27**	**x2.50**	**x2.57**	**x1.22**	**x2.37**	**x2.37**

Table 15: Speedups with armclang compiler

Video	Intra			Low-complex			High-complex		
Name	**AUTO**	**MAN**	**COMB.**	**AUTO**	**MAN**	**COMB.**	**AUTO**	**MAN**	**COMB.**
Akiyo	x1.13	x1.56	x1.56	x1.14	x2.52	x2.58	x0.94	x1.94	x1.94
Mobile	x1.10	x1.36	x1.34	x1.10	x1.59	x1.60	x0.98	x1.53	x1.52
Soccer	x1.19	x1.77	x1.79	x1.20	x2.50	x2.60	x0.97	x1.83	x1.84
City	x1.17	x1.64	x1.65	x1.18	x2.28	x2.35	x0.96	x1.74	x1.74
Johnny	x1.22	x2.00	x2.03	x1.23	x3.34	x3.53	x0.95	x2.00	x2.02
Park run	x1.12	x1.43	x1.42	x1.13	x1.87	x1.90	x0.98	x1.64	x1.64
Shields	x1.14	x1.54	x1.55	x1.17	x2.25	x2.31	x0.96	x1.68	x1.69
Basketball D.	x1.19	x1.79	x1.81	x1.19	x2.58	x2.69	x0.97	x1.78	x1.79
Blue sky	x1.18	x1.72	x1.73	x1.18	x2.43	x2.50	x0.95	x1.78	x1.79
Rush hour	x1.24	x2.07	x2.10	x1.21	x3.05	x3.18	x0.94	x1.87	x1.89
Average	**x1.17**	**x1.69**	**x1.70**	**x1.17**	**x2.44**	**x2.52**	**x0.96**	**x1.78**	**x1.78**

Table 16: Speedups with FCC compiler

the High-complex encoding configuration, armclang outperforms FCC. This suggests that armclang demonstrates greater efficiency in optimizing for vectorization.

This trend is further emphasized in manual vectorization, where armclang outperforms FCC in the All-Intra configuration as well. These observations highlight the efficiency of armclang in vectorization optimization scenarios.

To gain a better understanding of the differences between different implementations, we extracted a few hardware counters, focusing on the overall number of instructions and vectorization ratio. The

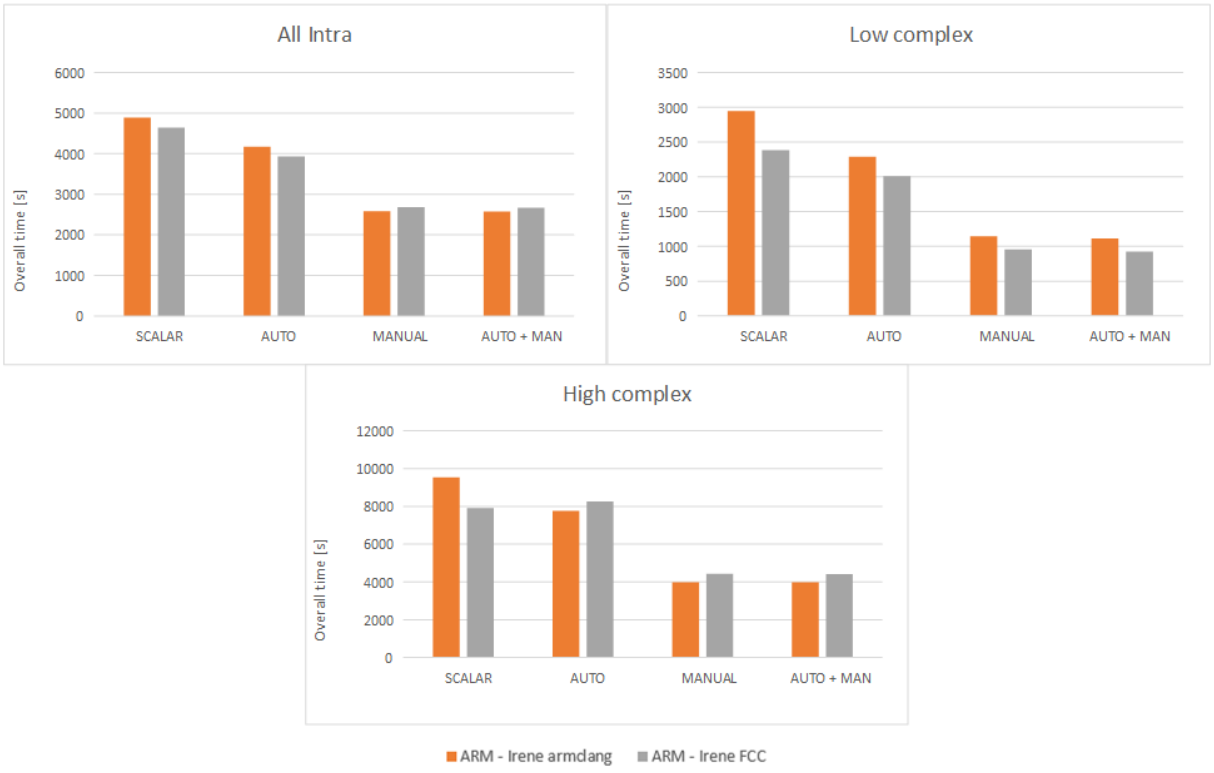


Figure 52: Comparison between amrclang and fcc

perf tool, which is readily available on the Irene cluster, was employed for this data extraction. For these measurements, we used ARM's compiler.

Figure 53 presents the total number of instructions used to encode all video sequences. For these experiments, we used armclang compiler.

As expected, the total number of instructions closely mirrors the trends observed in the measurements of runtime. This alignment suggests that the primary contributor to performance improvement is the reduction in the number of instructions achieved through vectorization. This insight leads to another crucial metric for understanding the application's optimization through vectorization - the vectorization ratio.

Figure 54 shows the average vectorization ratio for each implementation (using armclang compiler). The vectorization ratio is calculated as sve_inst_retired / instructions, where sve_inst_retired and instructions are hardware counters defined in the perf tool.

As evident from the graphs, the vectorization ratio of the auto-vectorization implementation is relatively low, ranging from 0.75% to 4.34%. In contrast, pure manual vectorization achieves a vectorization ratio between 4.9% and 12.67%, while in the combined vectorization implementation, 5.1% to 13.52% of the code is vectorized.

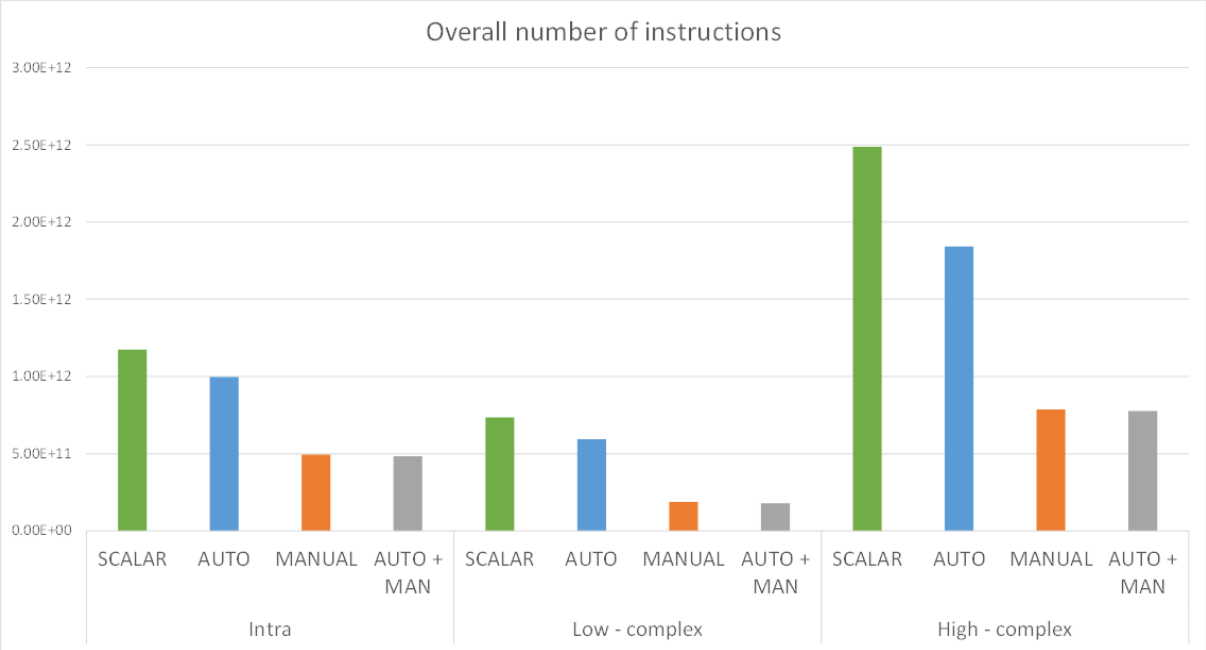


Figure 53: The overall number of instructions

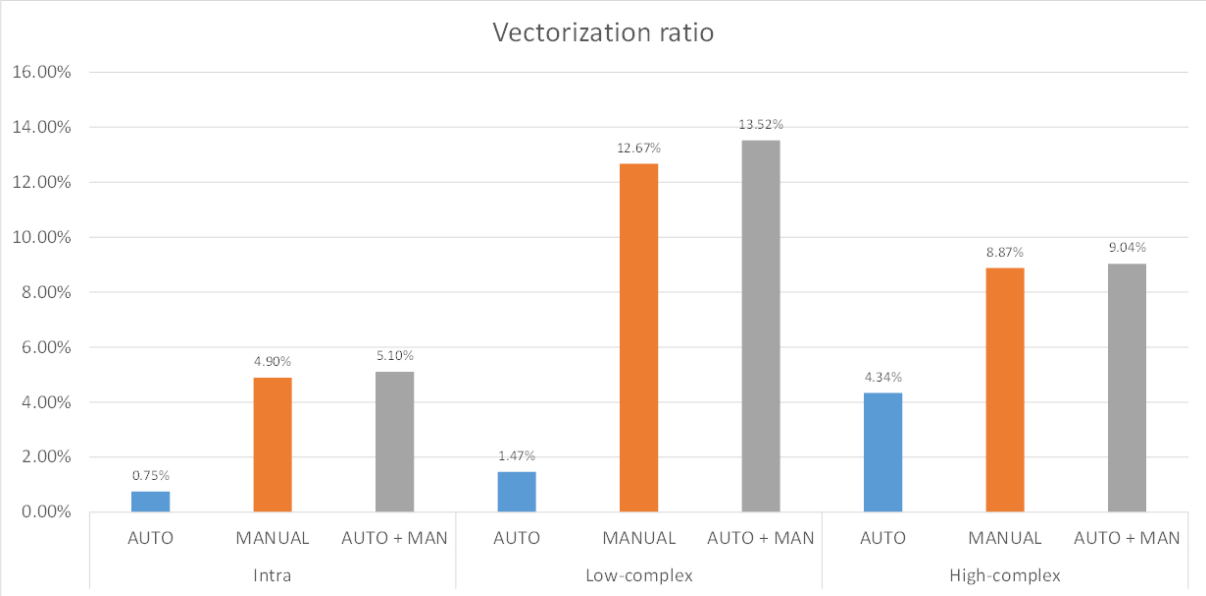


Figure 54: Vectorization ratio

Conclusion

The experiments reveal that auto-vectorization performed automatically by the compiler, without any additional input from the user, can give some performance improvements. However, the benefits can be significantly higher by manually vectorizing the application's code. Although manual vectorization

of the source code potentially requires a lot of effort, we showed that by selecting and manually vectorizing only a small subset of kernels, noticeable speedup can still be achieved.

Given that the Bolt65 codebase is extensive, containing hundreds of different kernels, rewriting the entire application and adapting it to SVE intrinsics would demand an effort comparable to creating the application from scratch. Thus, through extensive profiling and analysis of the application, we identified several key kernels, that consume most of the time in the application life cycle. With this targeted approach, we achieved a vectorization ratio of 4.9%-13.52% and an average speedup ranging from x1.60 to x2.52. It is important to note that all the numbers were extracted at the application level, encompassing all parts of the code in the analysis, including memory-related operations and non-vectorizable code, influencing the vectorization ratio.

Furthermore, we demonstrated that although different compilers yield similar conclusions in performance analysis of various implementations, there are still subtle differences that can potentially have an impact on the application running in real, production-level, systems.

3.1.2 Optimizations for HBM

The preliminary analysis and profiling of the basic scalar Bolt65 implementation on A64FX revealed a notable decrease in the share of memory-related operations compared with the same analysis conducted on a regular desktop Intel i5 CPU. The details of the analysis were given in the previous deliverable D3.1 [10], but the overall results are summarized in Figure 55. The graph depicts profiling on two distinct architectures: on an Intel i5 CPU profiled with Intel Vtune Amplifier and on the Irene cluster with A64FX profiled using perf. The results illustrate the average contribution of each function to the Bolt65 encoder runtime.

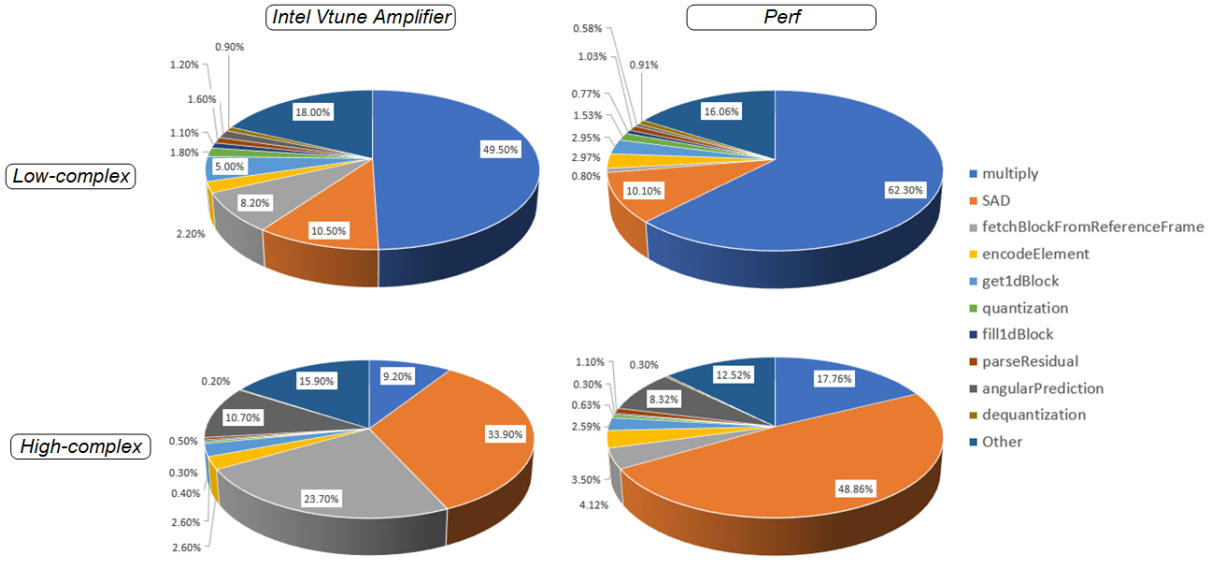


Figure 55: Profiling results

Share of memory-related methods, such as “fetchBlockFromReferenceFrame”, “get1dBlock”, and “fill1dBlock” are much lower on A64FX than on Intel CPU. The most significant difference is observed for the “fetchBlockFromReferenceFrame” method, where the share drops from 23.7% to 4.12% for the high-complex use-case, and from 8.2% to 0.8% for the low-complex use-case. To visually represent the disparity in memory-focused methods between the two architectures, we selectively extracted only these methods for both architectures and illustrated them in Figure 56

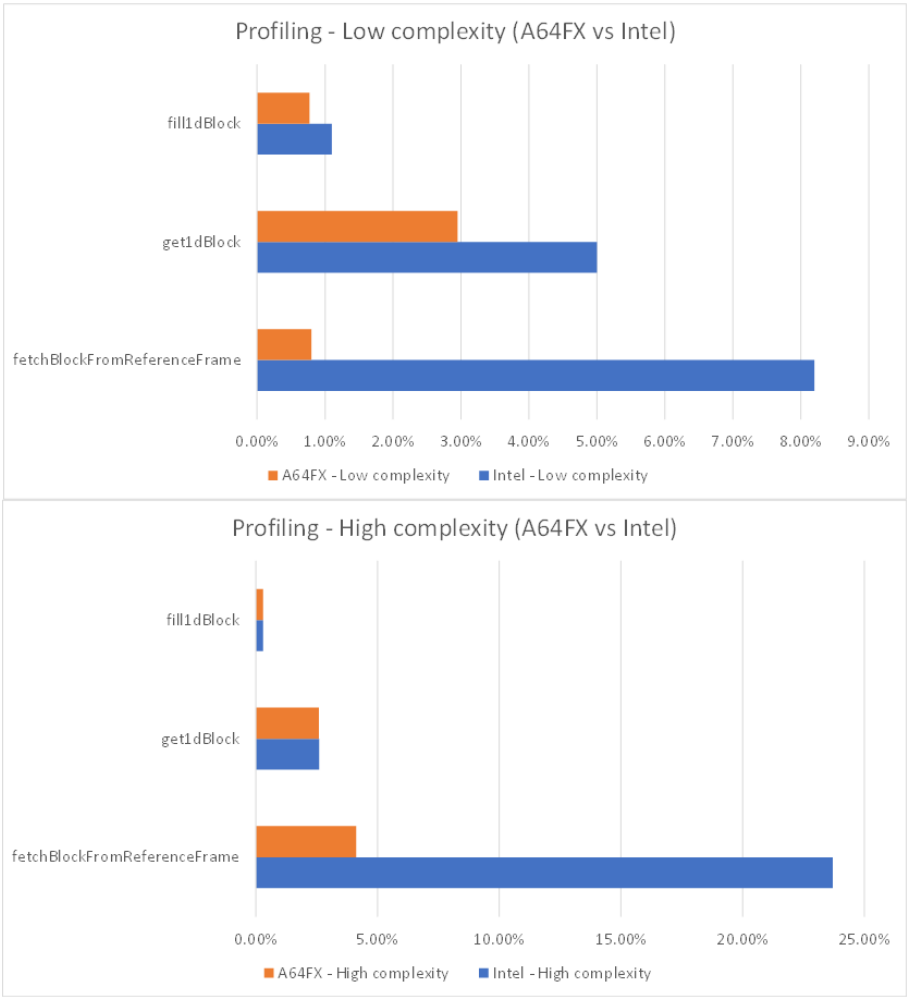


Figure 56: Share of memory-focused methods

To further investigate memory impact in the overall application life cycle, we extended our investigation to measure the application’s memory footprint. Following the same experimental setup as described in previous chapter, but utilizing the perf tool to gather several hardware counters required to calculate memory data volume: L2D_CACHE_REFILL, L2D_CACHE_WB, L2D_SWAP_DM, L2D_CACHE_MIBMCH_PRF. The formulas employed for calculating the memory bandwidths are as follows:

Memory read bandwidth [MBytes/s]

$1.0E{-}06{*}(L2D\_CACHE\_REFILL{-}(L2D\_SWAP\_DM{+}L2D\_CACHE\_MIBMCH\_PRF)){*}256.0/runtime$

Memory read data volume [GBytes]

$1.0E{-}09{*}(L2D\_CACHE\_REFILL{-}(L2D\_SWAP\_DM{+}L2D\_CACHE\_MIBMCH\_PRF)){*}256.0$

Memory write bandwidth [MBytes/s]

$1.0E-06*(L2D\_CACHE\_WB)*256.0/runtime$

Memory write data volume [GBytes]

$1.0E-09*(L2D\_CACHE\_WB)*256.0$

Memory bandwidth [MBytes/s]

$Memory\ read\ bandwidth + Memory\ write\ bandwidth$

Memory data volume [GBytes]

$Memory\ real\ data\ volume + Memory\ write\ data\ volume$

It's worth noting that the Likwid tool employs the same set of formulas for calculating these metrics.

In Table 17 and Table 18 we present the data obtained for the most memory-intensive video sequence in the test set, RushHour, which uses FullHD 1920x1080 resolution and contains 600 frames. Table 17 shows the overall data volume (in GBytes) when encoding the RushHour sequence in all encoding scenarios, while Table 18 shows the corresponding memory bandwidth (in MBytes/s).

		SCALAR	AUTO	MANUAL	AUTO + MAN
Intra	**Read**	68.89	69.17	69.28	69.26
	Write	45.23	45.32	45.54	45.82
	Overall	114.12	114.49	114.82	115.08
Low-complex	**Read**	43.02	43.21	42.96	43.14
	Write	30.56	30.61	30.61	30.45
	Overall	73.59	73.82	73.57	73.60
High-complex	**Read**	112.96	112.80	113.40	113.06
	Write	65.65	65.54	66.23	66.06
	Overall	178.61	178.34	179.63	179.11

Table 17: Overall memory volume [GBytes]

The overall data volume remains consistent across all implementation types (scalar, auto, manual, auto+manual), as expected, given that the application architecture remains the same in all cases, with the only variation being in the vectorized implementation of specific kernels. However, due to the faster execution of vectorized implementations, the memory bandwidth increases for the faster implementations.

		SCALAR	AUTO	MANUAL	AUTO + MAN
Intra	**Read**	75.13	92.12	177.54	180.17
	Write	49.33	60.35	116.70	119.21
	Overall	124.45	152.48	294.24	299.39
Low-complex	**Read**	70.10	92.92	211.49	221.19
	Write	49.80	65.81	150.70	156.14
	Overall	119.90	158.73	362.19	377.33
High-complex	**Read**	56.76	70.55	150.53	150.64
	Write	32.99	41.00	87.91	88.01
	Overall	89.74	111.55	238.44	238.66

Table 18: Overall memory bandwidth [MBytes/s]

In the high-complex encoding configuration, encoding RushHour with the auto+manual implementation stands out as the most memory-intensive scenario, with an overall memory bandwidth reaching 377 MBytes per second.

Conclusion The Bolt65 encoder's maximum overall memory bandwidth in the experiments reaches approximately 377 MB/s, which is notably lower than the bandwidth of any DDR or HBM memory. Furthermore, there is a consistent increase in memory bandwidth depending on the implementation. With faster computation, a proportional increase in overall memory bandwidth is achieved. This data, derived from profiling experiments, strongly suggests that memory-related operations are currently not the bottleneck of the system for Bolt65 encoding. This insight directed our optimization efforts more towards enhancing computation kernels using vectorization than focusing on improving memory-related aspects of the application.

However, it's essential to note that more complex scenarios in video encoding/transcoding involve handling higher-resolution video sequences, such as emerging 4k and 8k resolutions. Unfortunately, we were unable to include these resolutions in our experiments due to memory limitations and restrictions of the test cluster. Therefore, optimization for High Bandwidth Memory (HBM) and Bolt65, while not implementable in the current test environment, remains a factor that will be considered in future work.

3.2 Dyablo - Whole Sun

3.2.1 Introduction

Dyablo is a novel adaptive mesh refinement framework for the simulation of astrophysical plasmas developed at CEA. The objective of dyablo is to enable high performance computing for any astrophysical CFD problem from cosmology to exoplanet atmospheres, spanning a very wide range of temporal and spatial scales. One of the key projects of the development of dyablo is the simulation of solar-like magneto-hydrodynamic convection within the ERC project Whole Sun. Being able to

capture the dynamic of the Sun at every important scale require the simulation of a very large number of cells and thus require very high scalability and performance on modern hardware.

To reach exascale performance, the code relies on the portability performance library Kokkos. Effectively, Kokkos transforms computation kernels to dedicated architectures/programming models such as CUDA, HIP, SYCL or OpenMP allowing us to target many different types of hardware. On top of Kokkos managing shared-parallelism within a single node/computing unit, the full domain is decomposed into sub-domains synchronised on multiple nodes through MPI.

In the context EUPEX, the CEA dyablo and Atos' CEPP (Center for Excellence in Performance Programming) teams are collaborating on the analysis and the optimization of the solar part of dyablo. This work started early 2023 and is still ongoing. The simulation of the solar use-case requires the evaluation and the tuning of four main elements: the hydrodynamics, thermal conduction and viscosity computation kernel, and the AMR (adaptive mesh refinement) management cycle. Since all evaluations on EUPEX are done on a single-node setup, load balancing is not evaluated. Similarly, I/O questions are less relevant to EUPEX and are also ignored in this study.

The use-case used in the collaboration is the simulation of a convective cartesian slab. We initialize a box with a stratified fluid in hydrostatic equilibrium. We perturb this equilibrium which leads to the formation of convection cells within the box. Convection is highly turbulent which means the AMR kernels are highly stressed. Two types of run are made, a benchmarking run which is very short to evaluate the performance of the code and a validation run, much longer, allowing us to assess if the use of a specific combination of a compiler/hardware and compilation flags are not invalidating the results of the code.

In the following, we first describe the systems we used to experiment. After a brief description of what flags Kokkos propagates, we present the results of our first experiments. Motivated by unexpected results with one toolchain, we then explore the impact of the Kokkos runtime, and override the default Kokkos flags to investigate the issue and maximize performance. Finally, we show results comparing different distribution schemes, as well as studying the impact of the use of HBM, which will equip the RHEA processor, and then conclude.

3.2.2 Work Done

Test Systems Description

In order to run the benchmarks and verify that produced results are correct, we decided to conduct the experiments on the following ATOS systems:

- Intel Xeon 8358 (Icelake), 2 sockets, 2x32 cores, 2.6GHz, 256GB DDR4, x86_64, AVX2, AVX512
- AMD EPYC 7763 (Milan), 2 sockets, 2x64 cores, 2.45GHz, 256GB DDR4, x86_64, AVX2
- Ampere Altra Q8030 (Neoverse N1), single socket, 80 cores, 3GHz, 256GB DDR4, aarch64, NEON
- Fujitsu FX700 (A64FX), single socket, 48 cores, 1.8GHz, 32GB HBM2, aarch64, NEON, SVE

The AMD and Intel systems are used as reference systems, that is the historical x86_64 instruction set with vector extensions (AVX2, AVX512). They will be referred to as the x86 systems in the following. The Fujitsu system is used to validate that targeting SVE system does not alter compilation and result correctness after execution. The Ampere Altra system is used to test initial performance evaluation on aarch64 without SVE. These 2 systems will be referred to as ARM systems in the following.

Indeed, although the A64FX architecture offers SVE and HBM, its internal micro architecture is that of a SPARC from earlier Fujitsu CPU generations, with an ARM frontend [55]. Moreover, the memory layers do not expose an L3 cache. In short, it is quite far from what the RHEA processor will be. Early performance results comfort this position (see below). It exposes HBM, but from an older generation (HBM2 on A64FX, versus HBM2e on RHEA or Sapphire Rapids) and with limited quantity (32GB only). Finally, its lack of DDR prevents fair comparisons between the two memory technologies.

For the x86 systems, we targeted the following toolchains:

- Intel “Classic” (icc/icpc/ifort): considered the best-in-class when we started the study. Although OneAPI is closing the gap with its predecessor in terms of performance, we could still observe some runtime issues leading to bug reports.
- GCC: Still considered the most stable compiler for all machines.
- AOCC: To verify the behavior of LLVM based compiler on x86, we chose to use the AMD compiler, that is based on LLVM.

For the ARM systems, we targeted the following toolchains:

- Arm Compiler for Linux (ACFL): considered the best-in-class for performance when targeting ARM systems; LLVM based compiler.
- GCC: Also considered the most stable on ARM systems. Behaves similarly on x86 and ARM, can be used for comparison.

Finally, since the ARM CPUs do not provide any Simultaneous MultiThreading (SMT) feature, and since SiPearl did not announce such feature for RHEA, we decided to disable SMT during all our experiments.

Initial Issues with Kokkos

Dyablo is mostly built around the Kokkos C++ Programming Model - Parallel Execution and Memory Abstraction[1]. Kokkos is a set of headers, mainly consisting of routines and templates that are compiled when building the final application or library. Only a small runtime is compiled beforehand, when building the Kokkos library. Following the recommendations, dyablo trusts Kokkos for setting its build flags: *"Kokkos propagates all the necessary flags to your project."*[2].

Kokkos propagates the flags to the projects in three different ways, specified by the user:

- Default: When the user does not specify anything, Kokkos passes the `-O3` flag to the compiler.

[1] Kokkos: `https://kokkos.org/about/`
[2] `https://github.com/kokkos/kokkos/blob/master/BUILD.md`

- Native: If specified via `KOKKOS_ARCH_NATIVE=ON`, Kokkos will append `-march=native` and `-mtune=native` to the `-O3` flag.
- Arch specific: Kokkos allows the user to target a specific architecture using a predefined dictionary of optimized flags for each pair (target CPU, target compiler)[3]. For instance, the flag definition when targeting A64FX is the following:

```
Clang -march=armv8.2-a+sve -msve-vector-bits=512
GNU -march=armv8.2-a+sve -msve-vector-bits=512
MSVC NO-VALUE-SPECIFIED
NVHPC NO-VALUE-SPECIFIED
DEFAULT -march=armv8.2-a+sve
```

The observation of this particular example unveils several issues:

- Recent compilers are aware of the A64FX micro-architecture, and `-march=a64fx` is available in GCC since version 10, and in LLVM since version 11. The `-march=a64fx` specification enables armv8.2, SVE, but also some side extensions available on the CPU. This issue is bound to be true for any recent CPU, where both the compiler and the Kokkos CMake file need to be updated in order to exploit the latest features.
- As stated by ARM, but also valid for all CPU families except x86, the `-march -mtune` scheme is subsumed by `-mpcu`[4]. This ensures both the instruction set and micro architecture features (for instance, presence of two pipelines per core) are properly exploited.

We can observe similar issues when we look at the native example above:

- `-mtune=native` does not work on ARM systems with LLVM based compilers, including ACFL. As per ARM recommendations, `-mcpu=native` is to be used instead. The CMake file has been fixed since our initial experiments[5].
- Some compilers use another set of flags, for instance NVHPC uses `-tp=native`.

From these few observations, we can conclude that trusting the default Kokkos flag propagation is either non-working or suboptimal when targeting performance, especially when targeting recent machines.

Another concern lies behind using aggressive math assumptions (usually referenced as fast math mode) by default. Indeed, they can raise issues around semantic correctness[6], but more importantly it is unfair when comparing the ability of other compilers. Indeed, if the default behavior of the Intel compilers is to apply such optimizations, we should either be applying them by default on other compilers or be disabling them on Intel compilers.

For the sake of maximized performance, we chose to apply the fast math flags on all compilers. These flags can be overridden by setting the usual CMake flags:

```
-DReleaseType="None"
-DCMAKE_CXX_FLAGS="<compile flags>"
```

[3] https://github.com/kokkos/kokkos/blob/master/cmake/kokkos_arch.cmake
[4] https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/compiler-flags-across-architectures-march-mtune-and-mcpu
[5] https://github.com/kokkos/kokkos/issues/5808
[6] https://github.com/spack/spack/discussions/38689

ARM Software Stack Check and Preliminary Performance Assessment

During this step we verify that the application is behaving similarly on the ARM system as it is on the reference systems, that are Intel and AMD x86 systems. The following points are verified:

- Compilation with available toolchains: verify which compilers can build the application, using the custom set of flags we defined.
- Numerical validation: for every version that we could build, verify that the application produces results within acceptable deviation margin.
- Early performance analysis: given the theoretical peak performance of the machine and measured time to solution; we can draw rough conclusions about performance disparities between ARM and X86 systems.

For all these toolchains, we did override the default compiler flags, mainly to:

- Target the machine micro architecture (initially broken on ARM systems, see above)
- Force fast math on all compilers, as the Intel compilers does by default.

The test cases start from an initialised box (as stated in the introduction), set at simulated time of 6. The full test case simulates 44 seconds, until simulated time 50; whereas the shorter case stops after 1000 steps, that is around 0.08 seconds.

Table 19 presents the results we obtained on the shorter case. We can observe that all the configurations led to proper build, execution.

CPU	Compiler	Execution Time (s)
AMD EPYC 7763	AOCC	156.02
	GCC	137.039
	Intel	213.666
Ampere Altra Q8030	ACFL	152.846
	GCC	179.02
Fujitsu FX700	ACFL	981.171
	GCC	1271.63
Intel Xeon 8358	AOCC	261.664
	GCC	217.436
	Intel	347.98

Table 19: Dyablo runtime for the shorter test case using the different configurations

To further validate, we ran the full simulation, until simulated time of 50. The performance results are shown in Table 20. Also, this full run allowed us to observe the numerical deviation over the internal energy. This value is represented in Figure 57.

Intel Suite Results The execution on the Icelake system using the Intel compiler led to a SLURM timeout: it went over the time limit of 2 days we specified for this job. After further investigations, this job stopped at simulated time of 46 seconds (out of 50), with snapshot values within acceptable margin. The unexpected poor performance using this toolchain compared to GCC is discussed in the build flags study section.

CPU	Compiler	Execution Time (s)
AMD EPYC 7763	AOCC	80612.3
	GCC	75343
	Intel	111878
Ampere Altra Q8030	ACFL	83620.5
	GCC	96228
Intel Xeon 8358	AOCC	136324
	GCC	112628
	Intel	SLURM Timeout (2 days)

Table 20: Dyablo runtime for the full test case using the different configurations

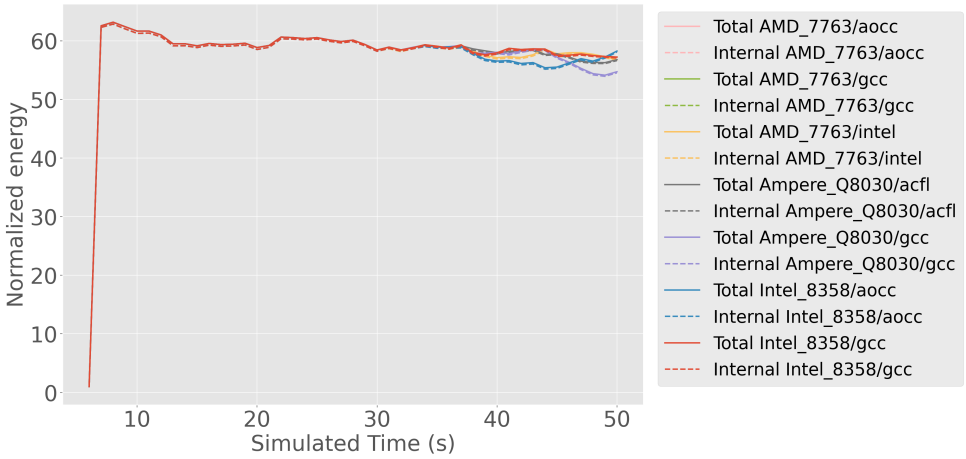


Figure 57: Value of the internal energy during the simulation. We can observe a slight numerical deviation occurring at simulated time of 35, for all tested configurations.

ARM Support We identified a few issues when building on ARM systems. They are mentioned briefly in the section above and fixed in the related upstream projects at the time of writing (Kokkos, CMake).

A64FX Results Given the long time it takes to run this full simulation (20h on our fastest configuration), and poor performance of the A64FX (7x slower when running the shorter test case), we decided to not run this full simulation on A64FX. We trust the numerical stability we observed on the short test will remain within acceptable margin when running on SVE. Although not presented in this document, we have validated that it does not deviate up to step 10000. Validating the full run on SVE is one of the first experiments we will do when we have access to Grativon3, or RHEA in the longer term.

Numerical Validation As shown in figure 57, we can observe a slight deviation in the energy value starting around simulated time 35. This is, however, within an acceptable margin.

Performance The first observation we can make is that, although not fitting perfectly, the time to solution difference between configurations (CPU and compiler) remains similar when we change the simulation duration. For instance, in the 1000 steps case, the run with GCC on the Icelake is 18%

slower than the run with GCC on Neoverse-N1 (Table 19), while the difference is 15% in the full run (Table 20). This allows us to use faster benchmarking in the following studies while keeping results representative enough for a first glance.

Second, we can observe from both Tables 19 and 20 (respectively short and full run) that the time to solution of the single socket Neoverse-N1 configuration is in between the performance of the dual socket Milan and of the dual socket Icelake systems. During these preliminary experiments, we followed developer recommendations by starting one process per socket. According to internal timers, the MPI overhead due to running on two-socket machines is around 15%. Further experiments (see section 3.2.2) have shown that this still brings best performance in most configurations, even compared to running with a single MPI process on the whole bi-socket machines. However, and even considering this overhead, the performance of the Neoverse-N1 ranks at a similar efficiency (time-to-solution / theoretical RPEAK).

Kokkos Runtime Flags Impact

The study of the Kokkos runtime flags impact was quite straightforward: we always built dyablo with the same flags but linking against a Kokkos runtime built with different flag sets. Along with the three different sets of flags Kokkos uses, described in the previous section, we also added a 4th way forcing the `-O0` flag. As shown in table 21, the variation is negligible, even when comparing Kokkos built with `-O0` against Kokkos build with more aggressive `-O3` plus native flags. Only one configuration (line 10) led to significant degradation, that we could reproduce. However, we did not spend time investigating this particular case.

From these results we can assume that Kokkos runtime indeed has neglectable to no impact on the performance of dyablo.

Build Flags Overriding

In the previous section, we decided to override the default flags, propagated by Kokkos, to force fast math and properly target the native host, on top of applying level 3 optimizations. We refer to this configuration as "custom" in the following, in contrast with the "default" set of flags set by Kokkos. This led the Intel suite to produce binaries running unexpectedly slower than GCC. Our investigation into these poor results started with falling back to the default flags.

As shown in lines 6 and 20 of Table 22, overriding the flags with native target degrades performance by 5 to 11% when compiling with the Intel suite. This has been confirmed when checking the build log files: the only difference is the `-xhost` flag (equivalent to `-march/-mtune=native` when using the Intel suite). We also applied the recommendations from AMD (`-march=core-avx2`) [7] and Intel (`-xCOMMON-AVX512`) [8], leading to the same results as when using `-xhost`. Using the default flags, but hinting Kokkos with `KOKKOS_ARCH_NATIVE=ON` and `KOKKOS_ARCH_ICX=ON` led also to similar degradation.

[7] https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/programmer-references/compiler-options-quick-ref-guide-epyc-7xx3-series-processors.pdf

[8] https://www.intel.com/content/www/us/en/developer/articles/guide/relion-3-1-tuning-guide-on-xeon-based-platfo html

#	CPU	Compiler	Kokkos Runtime Build Flags	Dyablo Execution Time (s)	vs -O0 (%)
1	AMD EPYC 7763	AOCC	-O0	159.256	
2			Native	160.751	0.94 %
3			Default	160.873	1.02 %
4			ARCH=ZEN3	158.796	-0.29 %
5		GCC	-O0	132.682	
6			Native	132.025	-0.50 %
7			Default	133.262	0.44 %
8			ARCH=ZEN3	132.949	0.20 %
9		Intel	-O0	207.643	
10			Native	248.866	19.85 %
11			Default	209.291	0.79 %
12			ARCH=ZEN3	206.249	-0.67 %
13	Ampere Altra Q8030	ACFL	-O0	153.153	
14			Native	153.336	0.12 %
15			Default	152.658	-0.32 %
16			ARCH=ARMV8.1	152.977	-0.11 %
17		GCC	-O0	171.197	
18			Native	172.035	0.49 %
19			Default	173.113	1.12 %
20			ARCH=ARMV8.1	171.881	0.40 %
21	Fujitsu FX700	ACFL	-O0	975.215	
22			Native	956.436	-1.93 %
23			Default	957.039	-1.86 %
24			ARCH=A64FX	985.533	1.06 %
25		GCC	-O0	1234.20	
26			Native	1227.97	-0.50 %
27			Default	1224.19	-0.81 %
28			ARCH=A64FX	1224.66	-0.77 %
29	Intel Xeon 8358	AOCC	-O0	223.921	
30			Native	223.481	-0.20 %
31			Default	223.484	-0.20 %
32			ARCH=ICX	223.362	-0.25 %
33		GCC	-O0	204.711	
34			Native	204.262	-0.22 %
35			Default	211.744	3.44 %
36			ARCH=ICX	203.509	-0.59 %
37		Intel	-O0	307.045	
38			Native	305.829	-0.40 %
39			Default	306.682	-0.12 %
40			ARCH=ICX	306.198	-0.28 %

Table 21: Dyablo runtime for the shorter test case, depending on Kokkos runtime library compiler flags. We can observe negligible to no impact.

The same experiment was also applied to other CPUs and compilers. As presented in the remaining of Table 22, gains vary between negligible up to around 25% on the Fujitsu FX700 (A64FX).

This study allows us to confirm the importance of flags when dealing with Kokkos and recent micro architectures. We could report the unexpected results to upstream technology providers. Finally, our next studies will be driven by these results.

#	CPU	Compiler	Application Build Flags	Execution Time (s)	vs default (%)
1	AMD EPYC 7763	AOCC	default	147.682	
2			custom	149.226	1.05 %
3		GCC	default	135.929	
4			custom	120.616	-11.27 %
5		Intel	default	183.488	
6			custom	193.224	5.31 %
7	Ampere Altra Q8030	ACFL	default	153.276	
8			custom	150.359	-1.90 %
9		GCC	default	176.48	
10			custom	170.104	-3.61 %
11	Fujitsu FX700	AOCC	default	1160.76	
12			custom	925.105	-20.30 %
13		GCC	default	1584.64	
14			custom	1194.96	-24.59 %
15	Intel Xeon 8358	AOCC	default	232.718	
16			custom	211.189	-9.25 %
17		GCC	default	211.744	
18			custom	190.701	-9.94 %
19		Intel	default	275.143	
20			custom	307.554	11.78 %

Table 22: Dyablo runtime for the shorter test case, with different build flags on the different configurations.We can observe significant degradation when targeting the host machine ("custom") and using the Intel suite. With other configurations, targeting the host machine leads to gains up to 24%.

Distribution Schemes and HBM Evaluation

During this study, we evaluated the impact on performance of several process and thread distribution schemes, as well as forcing the allocation of buffers within the HBM versus DDR5. This involved the design of a portable methodology for binding to compute and memory resources.

In the remaining, we distinguish 1) the distribution scheme (also referred to as mapping in OpenMPI `--map-by`), that is how we choose to distribute processes and threads; from 2) the binding methodology, that is how we implement this distribution scheme.

Binding Methodology For the binding methodology to be portable, the underlying tools must not depend on vendor specific technologies. For instance, the Intel MPI variables like `I_MPI_DOMAIN=socket` will not be interpreted by OpenMPI, and Intel MPI will most likely not be available on ARM systems. The general idea behind the methodology we used is to rely the portable tool `hwloc` to bind in a similar way on every system, and simply ask the launcher (MPI, SLURM, etc.) not to bind anything.

> *The Portable Hardware Locality (hwloc) software package provides a portable abstraction (across OS, versions, architectures, ...) of the hierarchical topology of modern architectures, including NUMA memory nodes (DRAM, HBM, etc.), sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such*

as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or GPUs. [9]

`hwloc` is developed by Inria Bordeaux, and will be ported to RHEA as part of WP5 of EUPEX. The way of numbering the resources in a logical way, versus a physical way when using other tools, enforces that the binding will always be the same, resilient to BIOS updates, OS change, etc.

Figure 58 illustrates an example of binding execution to NUMA 0 of socket 0 but allocating memory to NUMA 3 of the same socket 0, using the following command:

```
hwloc-bind --cpubind socket:0.numa:0 --membind socket:0.numa:3 $APP
```

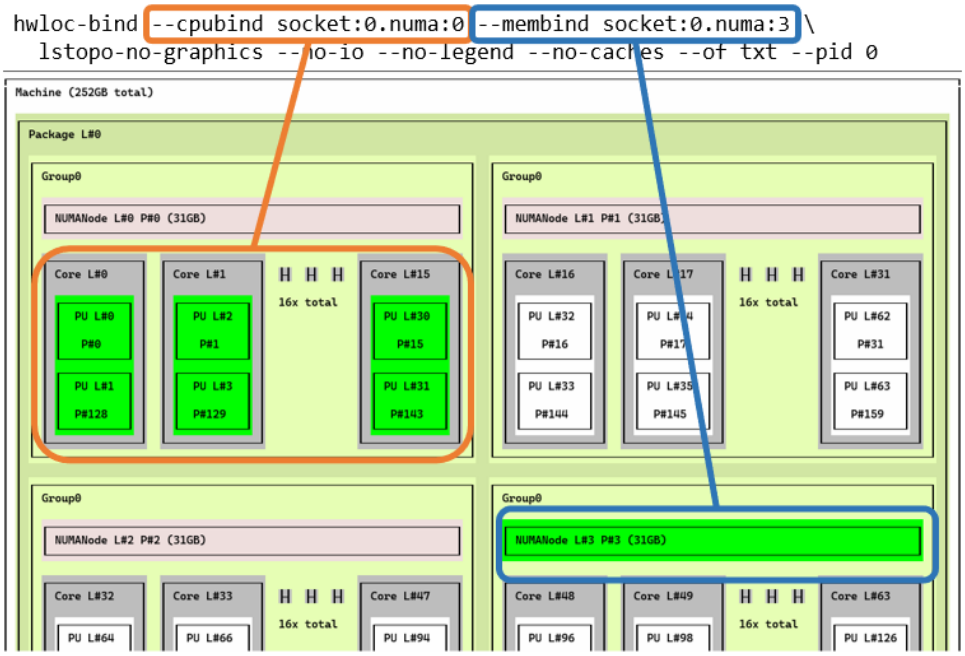


Figure 58: Illustration of `hwloc-bind` usage for binding. The output is produced by `lstopo`, another tool bundled with the `hwloc` distribution.

More complex distribution schemes can be constructed in combination with MPI launchers. For instance, the following command[10] will start two MPI processes, asking the MPI launcher to not bind anything, then assigning 1 socket per rank, and binding to the memory of the NUMAs attached to the corresponding socket:

```
mpiexec -n 2 --map-by none --bind-to none \
    hwloc --cpubind socket:$MPIRANK \
        --membind $(hwloc-calc socket:$MPIRANK --local-memory) \
            $DYABLO_EXE $DYABLO_ARGS
```

As of today, the thread binding must be done using OpenMP environment variables. `hwloc` provides a tool (`hwloc-calc`) to convert a logical resource identifier to its physical identifier. Using this tool, it is then possible to generate the proper `OMP_PLACES`, in the following form:

[9] `https://www.open-mpi.org/projects/hwloc/`
[10] The command is simplified for the sake of readability; actual implementation requires the use of wrapper scripts.

```
# Rank 0
export OMP_NUM_THREADS=56
export OMP_PROC_BIND=true
export OMP_PLACES="{0},{1},{2},{3},{4},{5},{6},{7},{8},{9},{10},{11},{12},{13}, \
    {14},{15},{16},{17},{18},{19},{20},{21},{22},{23},{24},{25},{26},{27},{28}, \
    {29},{30},{31},{32},{33},{34},{35},{36},{37},{38},{39},{40},{41},{42},{43}, \
    {44},{45},{46},{47},{48},{49},{50},{51},{52},{53},{54},{55}"
hwloc-bind core:0-55.pu:0 --membind numa:0 numa:2 numa:4 numa:6 --strict \
    $DYABLO_EXE $DYABLO_ARGS

# Rank 1
export OMP_NUM_THREADS=56
export OMP_PROC_BIND=true
export OMP_PLACES="{56},{57},{58},{59},{60},{61},{62},{63},{64},{65},{66},{67}, \
    {68},{69},{70},{71},{72},{73},{74},{75},{76},{77},{78},{79},{80},{81},{82}, \
    {83},{84},{85},{86},{87},{88},{89},{90},{91},{92},{93},{94},{95},{96},{97}, \
    {98},{99},{100},{101},{102},{103},{104},{105},{106},{107},{108},{109},{110}, \
    {111}"
hwloc-bind core:56-111.pu:0 --membind numa:8 numa:10 numa:12 numa:14 --strict \
    $DYABLO_EXE $DYABLO_ARGS
```

On top of such binding tools, `hwloc` comes with tools to observe and verify the actual bindings (`hwloc-ps`, or `lstopo` as illustrated by Figure 58). Usual `htop` or other tools can also be used to double check. Using such tools, we could confirm that this binding methodology leads to similar behavior when using different MPI implementations and on all the systems described earlier.

Distribution Schemes Using the binding methodology described above, we could easily explore the various process and thread distribution schemes. We defined three distribution schemes:

- CCD (L3): one process is started per group of cores attached to the same L3 cache, then one thread per core within each CCD. This distribution scheme is only available on the AMD EPYC 7763 system: the other systems share an L3 cache for a whole socket (Ampere Altra Q8030, Intel Xeon 8358) or have no L3 caches (Fujitsu FX700).
- NUMA: one process is started per NUMA available on the machine, then one thread per core within each NUMA.
- Socket: one process is started per socket available, then one thread per core within each socket.
- Machine: one process is started for the whole machine, with one thread per core.

Also, please note that the Ampere Altra Q8030 has one NUMA domain only, and that both ARM systems have one socket only. Consequently, the only distribution scheme available on Ampere is the Machine one, hence the absence of exploration for this system.

The various configurations shown in Table 23 clearly show the distribution scheme has an impact on dyablo's performance of up to 15%. Moreover, and although the socket distribution scheme seems to stand out, we can see that a different configuration (CPU and compiler) implies a different distribution scheme to achieve best performance. This calls for extra caution when testing on RHEA.

#	CPU	Compiler	Distribution	Dyablo Execution Time (s)	vs worse (%)
1	AMD EPYC 7763	AOCC	CCD (L3)	167.246	
2			NUMA	167.389	
3			Socket	158.502	-9.94 %
4			Machine	176.004	←
5		GCC	CCD (L3)	150.879	←
6			NUMA	148.486	
7			Socket	134.780	-10.67 %
8			Machine	142.685	
9		Intel	CCD (L3)	212.225	
10			NUMA	215.604	
11			Socket	207.600	-8.67 %
12			Machine	227.301	←
13	Fujitsu FX700	ACFL	NUMA	957.426	←
14			Machine	926.034	-3.28 %
15		GCC	NUMA	1222.24	-1.75 %
16			Machine	1244.07	←
17	Intel Xeon 8358	AOCC	NUMA	258.844	←
18			Socket	224.028	-13.47 %
19			Machine	241.839	
20		GCC	NUMA	238.999	←
21			Socket	203.496	-14.85 %
22			Machine	213.866	
23		Intel	NUMA	341.684	←
24			Socket	306.050	-10.43 %
25			Machine	316.560	

Table 23: Dyablo runtime for the shorter test case, depending on the distribution scheme. The arrow points to the worse execution time, against which the performance gain is calculated.

Starting one process per core is not supported by dyablo on all the configurations: the domain decomposition for the test case supports up to 64 processes. Although supported on some systems (Fujitsu with 48 cores, or Intel Xeon 8358 with 2x32 cores), some experiments have shown performance degradation compared to the other distribution schemes.

HBM Impact In order to fairly compare the impact of using HBM versus DDR5, we targeted a dedicated system: Intel Xeon Max 9480 (Sapphire Rapids with on-package HBM): a bi-socket system with 56 cores per socket, 512GB of DDR5, 64GB of HBM2e per socket, with x86_64 instruction set along with AVX2 and AVX512 vector extensions.

Using the same binding methodology described above, we can fine tune the memory binding of an application. The Intel Xeon Max 9480 offers several configurations to access its HBM, summarized on Figure 59 (source: Intel[11]). We used the SNC4 configuration, with the flat HBM setting.

Using this configuration, we can address every core group and memory bank independently. We validated the approach by running the stream benchmark on both the DDR5 and HBM, reproducing results from the literature [56]. Binding to the HBM instead of the DDR5 brings around 3$\times$ performance gain to the stream benchmark.

[11] https://www.intel.com/content/www/us/en/content-details/769060/intel-xeon-cpu-max-series-configuration-and-tu
html

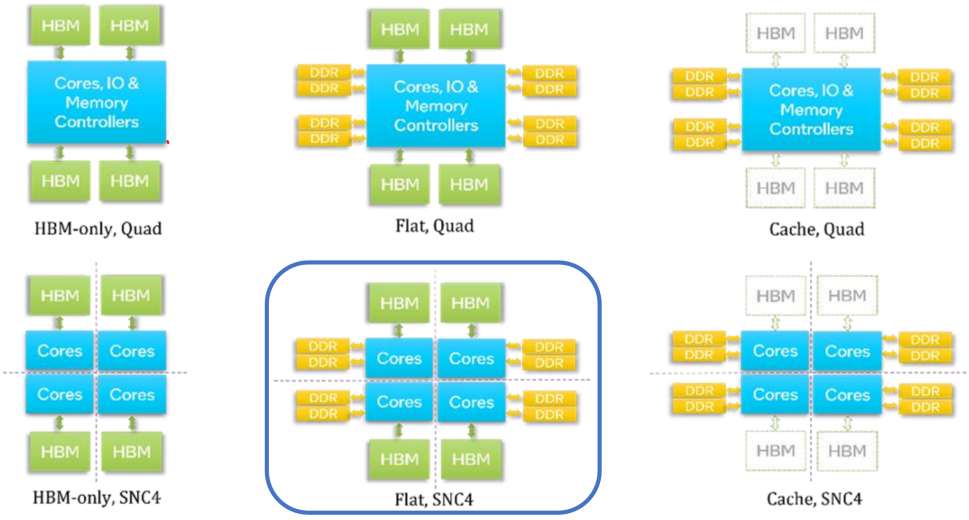


Figure 59: Illustration of the different configurations for the Intel Xeon Max 9480

We describe here two memory allocation schemes, as we did for the distribution schemes above:

- HBM: allocate everything in the HBM associated to the compute resources where the processes are bound.
- DDR5: allocate everything in the DDR5 associated to the compute resources where the processes are bound.

Again, and following the same binding methodology as earlier, `hwloc` provides the necessary tools to calculate the logical NUMA identifiers that are close to a given compute resource and to generate the proper binding command.

Running dyablo on the whole machine when binding to either the HBM or the DDR5 produced the results presented in Table 24. Dyablo benefits from HBM, but the gains are not that significant compared to the bandwidth increase offered by this new memory technology. Some distributions schemes even bring performance degradation when allocating to the HBM. We can also note that the socket distribution scheme leads to the best performance on the Intel Xeon 9480.

3.2.3 Conclusion

The early experiments have triggered some bug reports to upstream tools and dependencies (Kokkos, CMake, hwloc). Thanks to our work and reporting, some fixes are now or will be available in later versions. Especially, hwloc already features improvements in the handling of heterogeneous memories [12]. Additionally, the first performance results we could observe showed similar machine efficiency on all the systems we have on hand. This allows us to conclude that dyablo is ready for ARM, regarding both the software stack support and performance wise.

However, the experiments raised few questions. Indeed, the HBM gains are weak and should investigated. Moreover, the flag study showed that forcing vectorization could bring performance

[12] `https://raw.githubusercontent.com/open-mpi/hwloc/v2.10/NEWS`

#	CPU	Compiler	Distribution	Memory	Dyablo Execution Time (s)	vs DDR5 (%)
1	Intel Xeon 9480	AOCC	NUMA	DDR5	219.088	
2				HBM	203.324	-7.20 %
3			Socket	DDR5	185.979	
4				HBM	174.022	-6.43 %
5			Machine	DDR5	203.079	
6				HBM	208.382	2.61 %
7		GCC	NUMA	DDR5	197.795	
8				HBM	181.472	-8.25 %
9			Socket	DDR5	163.588	
10				HBM	153.025	-6.46 %
11			Machine	DDR5	177.604	
12				HBM	203.866	14.79 %
13		Intel	NUMA	DDR5	276.754	
14				HBM	261.302	-5.58 %
15			Socket	DDR5	242.985	
16				HBM	230.104	-5.30 %
17			Machine	DDR5	258.818	
18				HBM	287.721	11.17 %

Table 24: Dyablo runtime for the shorter test case, depending on the memory allocation scheme, on the Intel Xeon Max 9480 system.

degradation, especially with the best-in-class compiler for such vectorization optimizations as the Intel one. This could indicate that the vector instructions are properly generated by the compilers, but that vectors are not fully populated during execution. This could denote low arithmetic intensity, or simply control bound algorithms. Properly filling the vectors would lead to not only higher arithmetic intensity, but also higher memory pressure. Optimizing in such a way would benefit both vectorization and HBM exploitation. The next step will be to investigate these aspects.

Finally, we could test and validate a binding methodology based on hwloc tools used in EUPEX. Having this tool ported to RHEA, as planned in WP5, will ensure the approach we developed will be suitable for the EUPEX pilot system.

4 Summary

This document has presented the work undertaken to optimise the EUPEX application code for the use of SVE and HBM, on the Fujitsu A64FX early software development vehicle hosted by the CEA. It also includes an iteration of progress on the work done on GPUs, from months 1-24 as part of deliverable D3.1 [10] - which demonstrates the portability of GPU kernels. Performance assessments of the different applications have also been documented. A particular focus has been placed in the document on focused on concerns regarding portability, scalability, and accuracy.

The different porting approaches described in the introduction have been covered by the EUPEX pilot applications.

The use of platform-optimised libraries is critical on any HPC platform, and the SVE-enabled A64FX platform is no exception. Work undertaken in particular by the ESPRESO developers found the Fujitsu BLAS library to be up to eighteen times faster than other BLAS implementations on the platform. The IFS application also observed the crucial performance implications of platform-optimised BLAS and FFTW libraries.

The importance of the compiler has been highlighted by a number of applications, regarding the ability to make efficient use of the SVE instruction set. The OpenGADGET application developers performed a detailed study of this question, finding the GNU compiler to vectorise significantly less well than the Fujitsu compiler, leading to a 30% performance deficit for the GNU compiler.

In a majority of applications, code modifications were needed to make effective use of SVE. Exposing SIMD characteristics at the algorithmic implementation level was found to be highly beneficial. Manual use of SVE intrinsics was also investigated by certain applications, with varying degrees of benefit. While for the IFS-CloudSC kernel, manual vectorisation was not found to yield better results than compiler-vectorised code, the ability to use C-language intrinsics in Fortran code was established. On the other hand work carried out on the Bolt65 mini-app highlighted that there are situations where manual SVE usage results in significantly improved performance as compared relying on auto-vectorisation. Possibly due to a lack of compiler auto-vectorisation, as seen by ESPRESO.

The detailed investigation into SVE undertaken in this deliverable is key to making efficient use of the upcoming EPI CPU. Nevertheless, vendor-specific, and therefore non-portable, approaches have had to be deployed.
The work undertaken by BigDFT developers to re-implement GPU kernels in SYCL has demonstrated a highly promising approach to achieve platform and performance portability - even if it was not possible to use an ARM based platform at this stage.
Cybeletech also demonstrated the use of containerisation as an approach to portability. Allowing them to deploy code with minimal effort as it had already been compiled and included necessary dependencies. They were able to do this with minimal memory overhead from the virtualisation layer, making maximum use of the limited HBM available.

Some of the EUPEX pilot applications have also demonstrated the effects of High Bandwidth Memory on application runtime. For large, heterogeneous applications such as the IFS, overall performance may benefit significantly, even if some application components (the CloudSC kernel, in the case of the IFS) do not see a substantial performance boost. The use of Intel hardware was crucial for this

point, as the HBM-equipped Xeon Max systems are the only platforms on which performance can be fairly compared with and without HBM.

This document concludes the reporting on the second phase of WP3, which is focused on the architecture-specific features of the EUPEX hardware. The next phase of the project will aim to broaden the optimisation focus out to the workflow level in heterogeneous and modular HPC architectures.

List of Acronyms and Abbreviations

C

CMG Core Memory Group

CPU Central Processing Unit

CUDA The Compute Unified Device Architecture is a parallel computing platform as well as an API that allows for the communication with certain types of graphics-processing units

CEA French Alternative Energies and Atomic Energy Commission, France

E

EC European Commission

EU European Union

Exascale Computer systems or Applications, which are able to run with a performance above 10^{18} Floating point operations per second

ECMWF European Centre for Medium-range Weather Forecasts, headquartered in Reading, UK

F

FFT Fast Fourier Transform

FLOP/s FLoating-point OPeration per Second

G

GPU Graphics Processing Unit

H

H2020 Horizon 2020

HBM High Bandwidth Memory, see 1.3

HEVC High Efficiency Video Coding

HPC High Performance Computing

HPCG The High Performance Conjugate Gradient benchmark is a benchmark based on a conjugate-gradient kernel

HPL The High Performance LINPACK (HPL) is a performant software package for solving linear system.

I

IB verbs The API for communication using InfiniBand (IB), a communication hardware

IFS Integrated Forecasting System

I/O Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

IPC Instructions Per Cycle

ISA Instruction Set Architecture

L

LINPACK LINPACK is a software package for solving linear systems

M

MPI Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

MPI-I/O MPI – Input/Output is an extentsion to MPI for I/O

MPMD Multiple-Program-Multiple-Data

MSA Modular Supercomputer Architecture

ML Machine Learning

MKL The Math Kernel Library is a mathematics library provided by Intel®

N

NUMA Non-Uniform Memory Access

NVLink NVLink describes the suite of tools for communicating between NVIDIA GPUs. It includes an API that requires physical NVLink bridges between GPUs to use

O

OpenMP Open Multi-Processing, Application programming interface that support multi-platform shared memory multiprocessing

R

RAM Random-Access Memory

S

SIMD Single Instruction Multiple Data

SVE Scalable Vector Extension : see 1.2

SW Software

T

TCP The Transmission Control Protocol (TCP) is one of the main communication protocols of the internet protocol

U

UCP The Unified Communication Protocol (UCP) is an API aimed at unifying different communication APIs, similar in that sense to MPI

W

WP Work package

WAM WAve Model, used in IFS forecasts to predict the ocean-atmosphere interface

X

x86 Family of instruction set architectures based on the Intel® 8086 CPU

Bibliography

[1] F. Limited. “Datasheet FUJITSU Processor A64FX”. In: (). URL: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf.

[2] T. G. C. de Calcul du CEA. *TGCC documentation*. URL: https://www-hpc.cea.fr/tgcc-public/en/latex/tgcc-public.pdf.

[3] F. Limited. “Development Studio Profiler User’s Guide”. In: (Mar. 2023). URL: https://software.fujitsu.com/jp/manual/manualfiles/m230003/j2ul2483/02enz012/j2ul-2483-02enz0.pdf.

[4] L. Limited. “Linaro Forge”. In: (). URL: https://www.linaroforge.com.

[5] A. Knüpfer et al. “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir”. In: Jan. 2012, pp. 79–91. ISBN: 9783642314759. DOI: 10.1007/978-3-642-31476-6_7.

[6] N. Stephens et al. “The ARM Scalable Vector Extension”. In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: 10.1109/MM.2017.35.

[7] F. Limited. “Fortran User’s Guide”. In: (Oct. 2023), pp. 262–306. URL: https://software.fujitsu.com/jp/manual/manualfiles/m230008/j2ul2580/03enz004/j2ul-2580-03enz0.pdf.

[8] F. Limited. “C User’s Guide”. In: (Oct. 2023), pp. 64–102. URL: https://software.fujitsu.com/jp/manual/manualfiles/m230008/j2ul2582/03enz004/j2ul-2582-03enz0.pdf.

[9] S. hynix. *History*. URL: https://www.skhynix.com/company/UI-FR-CP05/.

[10] D. Cesarini. *EUPEX Deliverable 3.1: Application Analysis Report*. Tech. rep. Dec. 2022.

[11] “mpi4py”. In: (). URL: https://mpi4py.readthedocs.io/en/stable/.

[12] “Apptainer”. In: (). URL: https://apptainer.org/.

[13] P.-H. Cournède et al. “Development and evaluation of plant growth models: Methodology and implementation in the pygmalion platform”. In: *Mathematical modelling of natural phenomena* 8.4 (2013), pp. 112–130.

[14] J. Aschbacher. “ESA’s Earth Observation Strategy and Copernicus,” in Satellite Earth Observations and Their Impact on Society and Policy”. In: *Singapore: Springer Singapore* (2017). URL: https://doi.org/10.1007/978-981-10-3713-9%5C_5.

[15] M. Götz, C. Bodenstein, and M. Riedel. “HPDBSCAN: highly parallel DBSCAN”. In: Nov. 2015, pp. 1–10. DOI: 10.1145/2834892.2834894.

[16] Z. Liu et al. “Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’23. ACM, Nov. 2023. DOI: 10.1145/3581784.3607073. URL: http://dx.doi.org/10.1145/3581784.3607073.

[17] S. Li and T. Hoefler. “Chimera: efficiently training large-scale neural networks with bidirectional pipelines”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. ACM, Nov. 2021. DOI: 10.1145/3458817.3476145. URL: http://dx.doi.org/10.1145/3458817.3476145.

[18] D. Narayanan et al. “PipeDream: Generalized Pipeline Parallelism for DNN Training”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP ’19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 1–15. ISBN: 9781450368735. DOI: 10.1145/3341301.3359646. URL: https://doi.org/10.1145/3341301.3359646.

[19] O. Beaumont, L. Eyraud-Dubois, and A. Shilova. “Pipelined Model Parallelism: Complexity Results and Memory Considerations”. In: *Euro-Par 2021: Parallel Processing*. Ed. by L. Sousa, N. Roma, and P. Tomás. Cham: Springer International Publishing, 2021, pp. 183–198. ISBN: 978-3-030-85665-6.

[20] X. Zhao et al. *Rockmate: an Efficient, Fast, Automatic and Generic Tool for Re-materialization in PyTorch*. 2023. arXiv: 2307.01236 [cs.LG].

[21] J. Rasley et al. “DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3505–3506. ISBN: 9781450379984. DOI: 10.1145/3394486.3406703. URL: https://doi.org/10.1145/3394486.3406703.

[22] M. Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2020. arXiv: 1909.08053 [cs.CL].

[23] J. Gusak et al. *Survey on Large Scale Neural Network Training*. 2022. arXiv: 2202.10435 [cs.LG].

[24] A. Folch et al. “The EU Center of Excellence for Exascale in Solid Earth (ChEESE): Implementation, results, and roadmap for the second phase”. In: *Future Generation Computer Systems* 146 (2023), pp. 47–61. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.04.006. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23001401.

[25] K.-J. Bathe. *Finite Element Procedures*. 2006. ISBN: 9780979004902.

[26] T. Sun et al. “A study of vectorization for matrix-free finite element methods”. In: *The International Journal of High Performance Computing Applications* 34.6 (2020), pp. 629–644. DOI: 10.1177/1094342020945005.

[27] Intel corporation. *Intel® 64 and IA-32 ArchitecturesOptimization Reference Manual*. Apr. 2012. URL: https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf.

[28] I. Reguly and M. Giles. “Finite Element Algorithms and Data Structures on Graphical Processing Units”. In: *International Journal of Parallel Programming* 43 (Apr. 2013). DOI: 10.1007/s10766-013-0301-6.

[29] L. Říha et al. “Efficient Implementation of Total FETI Solver for Graphic Processing Units Using Schur Complement”. In: *High Performance Computing in Science and Engineering*. Cham: Springer International Publishing, 2016, pp. 85–100. ISBN: 978-3-319-40361-8.

[30] C. Farhat and F.-X. Roux. “A method of finite element tearing and interconnecting and its parallel solution algorithm”. In: *International Journal for Numerical Methods in Engineering* 32.6 (Oct. 1991), pp. 1205–1227. DOI: 10.1002/nme.1620320604. URL: http://dx.doi.org/10.1002/nme.1620320604.

[31] D. Gadioli et al. “EXSCALATE: An Extreme-Scale Virtual Screening Platform for Drug Discovery Targeting Polypharmacology to Fight SARS-CoV-2”. In: *IEEE Transactions on Emerging Topics in Computing* (2022), pp. 1–12. ISSN: 2168-6750, 2376-4562. DOI: 10.1109/TETC.2022.3187134. URL: https://ieeexplore.ieee.org/document/9817028/ (visited on 09/02/2022).

[32] G. Vistoli et al. “MEDIATE - Molecular DockIng at homE: Turning collaborative simulations into therapeutic solutions”. en. In: *Expert Opinion on Drug Discovery* (July 2023), pp. 1–13. ISSN: 1746-0441, 1746-045X. DOI: 10.1080/17460441.2023.2221025. URL: https://www.tandfonline.com/doi/full/10.1080/17460441.2023.2221025 (visited on 07/11/2023).

[33] L. Genovese et al. “Efficient solution of Poisson’s equation with free boundary conditions”. In: *The Journal of chemical physics* 125.7 (2006), p. 074105.

[34] L. Genovese et al. “Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures”. In: *The Journal of Chemical Physics* 131.3 (July 2009), p. 034103. ISSN: 0021-9606. DOI: 10.1063/1.3166140. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.3166140/15673325/034103_1_online.pdf. URL: https://doi.org/10.1063/1.3166140.

[35] NVIDIA Corporation. *CUDA toolkit*. 2023. URL: https://developer.nvidia.com/cuda-toolkit. Accessed: 2023-08-03.

[36] NVIDIA Corporation. *cuFFT*. https://docs.nvidia.com/cuda/cufft/index.html. Accessed: 2023-08-03. 2023.

[37] L. E. Ratcliff et al. “Affordable and accurate large-scale hybrid-functional calculations on GPU-accelerated supercomputers”. In: *Journal of Physics: Condensed Matter* 30.9 (Feb. 2018), p. 095901. DOI: 10.1088/1361-648X/aaa8c9. URL: https://dx.doi.org/10.1088/1361-648X/aaa8c9.

[38] Advanced Micro Devices, Inc. *AMD Instinct GPUs*. https://www.amd.com/en/graphics/instinct-server-accelerators. Accessed: 2023-08-03. 2023.

[39] Intel Corporation. *Intel Max Series GPUs*. 2023. URL: https://www.intel.com/content/www/us/en/products/details/discrete-gpus/data-center-gpu/max-series.html. Accessed: 2023-08-03.

[40] Oak Ridge National Laboratory. *Frontier Supercomputer*. https://www.olcf.ornl.gov/frontier/. Accessed: 2023-08-03. 2023.

[41] Argonne Leadership Computing Facility. *Aurora Supercomputer*. https://www.alcf.anl.gov/aurora. Accessed: 2023-08-03. 2023.

[42] OpenMP. *OpenMP Compilers*. https://www.openmp.org/resources/openmp-compilers-tools/. Accessed: 2023-08-03. 2023.

[43] OpenACC-standard.org. *OpenACC*. https://www.openacc.org. Accessed: 2023-08-03. 2023.

[44] Khronos Group. *OpenCL*. https://www.khronos.org/opencl/. Accessed: 2023-08-03. 2023.

[45] Khronos Group. *SYCL*. https://www.khronos.org/sycl/. Accessed: 2023-08-03. 2023.

[46] Codeplay Software Ltd. *Codeplay ComputeCpp*. https://developer.codeplay.com/. Accessed: 2023-08-03. 2023.

[47] Intel Corporation. *Intel oneAPI DPC++*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/data-parallel-c-plus-plus.html#gs.0k3wn8. Accessed: 2023-08-03. 2023.

[48] A. Alpay et al. "Exploring the Possibility of a HipSYCL-Based Implementation of OneAPI". In: *International Workshop on OpenCL*. IWOCL'22. Bristol, United Kingdom, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450396585. DOI: 10.1145/3529538.3530005. URL: https://doi.org/10.1145/3529538.3530005.

[49] *OpenSYCL Github*. https://github.com/OpenSYCL/OpenSYCL. Accessed: 2023-08-03. 2023.

[50] V. Springel, N. Yoshida, and S. White. "GADGET: A Code for Collisionless and Gasdynamical Cosmological Simulations". In: *New Astronomy* 6 (Mar. 2000), pp. 79–117. DOI: 10.1016/S1384-1076(01)00042-2.

[51] V. Springel. "The cosmological simulation code gadget-2". In: *Monthly Notices of the Royal Astronomical Society* 364.4 (Dec. 2005), pp. 1105–1134. ISSN: 0035-8711. DOI: 10.1111/j.1365-2966.2005.09655.x. eprint: https://academic.oup.com/mnras/article-pdf/364/4/1105/18657201/364-4-1105.pdf. URL: https://doi.org/10.1111/j.1365-2966.2005.09655.x.

[52] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[53] C. Yang et al. "An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability". In: Nov. 2018, pp. 14–23. DOI: 10.1109/P3HPC.2018.00005.

[54] G. J. Sullivan et al. "Overview of the High Efficiency Video Coding (HEVC) Standard". In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (2012), pp. 1649–1668. DOI: 10.1109/TCSVT.2012.2221191.

[55] M. Sato. *The post-K project and Fujitsu ARM-SVE enabled A64FX processor for energy-efficiency and sustained application performance*. CEA-RIKEN Summer School, 13th June 2019, MDLS, Paris. 2019. URL: https://indico.math.cnrs.fr/event/4705/attachments/2362/2942/CEA-RIKEN-school-19013.pdf.

[56] J. McCalpin. "Bandwidth Limits in the Intel Xeon Max (Sapphire Rapids with HBM) Processors". In: Aug. 2023, pp. 403–413. ISBN: 978-3-031-40842-7. DOI: 10.1007/978-3-031-40843-4_30.

5 Appendix

5.1 Appendix: IFS

5.1.1 A handwritten SVE intrinsics kernel for the hottest loop in CloudSC

```
#include <stdint.h>
#include <stdbool.h>

#ifdef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */

#define MAX(x, y) (((x) > (y)) ? (x) : (y))

void hot_loop_(const int32_t JK, const int32_t KIDIA, const int32_t KFDIA, const
    int32_t KLEV, const int32_t KLON, const int32_t NCLV, const float ZEPSEC,
    int32_t LLINDEX3[KLON][NCLV][NCLV], float ZSOLQA[KLON][NCLV][NCLV], float
    ZSINKSUM[KLON][NCLV], float ZQX[KLON][KLEV][NCLV], float ZRATIO[KLON][NCLV]) {
#ifdef __ARM_FEATURE_SVE
    svint32_t ones_vec = svdup_n_s32(1);
    uint32_t lane_width = svcntw();
    svbool_t all_32 = svptrue_b32();
    svbool_t pg;

#endif
    for (int32_t JM = 0; JM < NCLV; ++JM) {
        for (int32_t JO = 0; JO < NCLV; ++JO) {
#ifndef __ARM_FEATURE_SVE
            for (int32_t JL = KIDIA - 1; JL < KFDIA; ++JL) {
                LLINDEX3[JM][JO][JL] = ZSOLQA[JM][JO][JL] < 0.0;
                ZSINKSUM[JM][JL] = ZSINKSUM[JM][JL] - ZSOLQA[JL][JO][JL];
            }
#else
            for (uint32_t JL = KIDIA - 1; svptest_first(all_32,
                pg=svwhilelt_b32(JL, KFDIA)); JL += lane_width)
            {
                svfloat32_t ZSOLQA_vec = svld1(pg, &ZSOLQA[JM][JO][JL]);
                svfloat32_t ZSINKSUM_vec = svld1(pg, &ZSINKSUM[JM][JL]);

                svbool_t pg2 = svaclt_n_f32(pg, ZSOLQA_vec, 0.f);
                svst1(pg2, &LLINDEX3[JM][JO][JL], ones_vec);
                svst1_f32(pg, &ZSINKSUM[JM][JL], svsub_f32_x(pg, ZSINKSUM_vec,
                    ZSOLQA_vec));
            }
#endif
        }

// !---------------------------
```

```
39 // ! recalculate scaling factor
40 // !--------------------------
41 #ifndef __ARM_FEATURE_SVE
42        for (int32_t JL = KIDIA - 1; JL < KFDIA; ++JL) {
43            float ZMM = MAX(ZQX[JM][JK][JL], ZEPSEC);
44            float ZRR = MAX(ZSINKSUM[JM][JL], ZMM);
45            ZRATIO[JM][JL] = ZMM / ZRR;
46        }
47 #else
48        svfloat32_t ZEPSEC_vec = svdup_n_f32(ZEPSEC);
49
50        for (uint32_t JL = KIDIA - 1; svptest_first(all_32, pg=svwhilelt_b32(JL,
              KFDIA)); JL += lane_width)
51        {
52            svfloat32_t ZQX_vec = svld1(pg, &ZQX[JM][JK][JL]);
53            svfloat32_t ZSINKSUM_vec = svld1(pg, &ZSINKSUM[JM][JL]);
54
55            svfloat32_t ZMM_vec = svmax_f32_x(pg, ZQX_vec, ZEPSEC_vec);
56            svfloat32_t ZRR_vec = svmax_f32_x(pg, ZSINKSUM_vec, ZMM_vec);
57
58            svst1_f32(pg, &ZRATIO[JM][JL], svdiv_f32_x(pg, ZMM_vec, ZRR_vec));
59        }
60 #endif
61
62 // !------
63 // ! scale
64 // !------
65        for (int32_t JO = 0; JO < NCLV; ++JO) {
66 #ifndef __ARM_FEATURE_SVE
67            for (int32_t JL = KIDIA - 1; JL < KFDIA; ++JL) {
68                if (LLINDEX3[JM][JO][JL]) {
69                    ZSOLQA[JM][JO][JL] = ZSOLQA[JM][JO][JL] * ZRATIO[JO][JL];
70                    ZSOLQA[JO][JM][JL] = ZSOLQA[JO][JM][JL] * ZRATIO[JO][JL];
71                }
72            }
73 #else
74            for (uint32_t JL = KIDIA - 1; svptest_first(all_32,
                  pg=svwhilelt_b32(JL, KFDIA)); JL += lane_width)
75            {
76                svint32_t LLINDEX3_vec = svld1(pg, &LLINDEX3[JM][JO][JL]);
77
78                svbool_t pg2 = svcmpeq_n_s32(pg, LLINDEX3_vec, 1);
79
80                svfloat32_t ZSOLQA_vec = svld1(pg2, &ZSOLQA[JM][JO][JL]);
81                svfloat32_t ZRATIO_vec = svld1(pg2, &ZRATIO[JO][JL]);
82                svfloat32_t ZSOLQA_flipped_vec = svld1(pg2, &ZSOLQA[JO][JM][JL]);
83
84
85                svst1_f32(pg2, &ZSOLQA[JM][JO][JL], svmul_f32_x(pg2, ZSOLQA_vec,
                      ZRATIO_vec));
86                svst1_f32(pg2, &ZSOLQA[JO][JM][JL], svmul_f32_x(pg2,
                      ZSOLQA_flipped_vec, ZRATIO_vec));
```

```
87
88              }
89  #endif
90          }
91      }
92  }
```