D4.3

First Co-design report

Augustin Degomme, Lubomir Riha, Marc Sergent, Loris Lucido, Erwan Raffin, Filip Vaverka, Daniel Wortmann, Andrea Ferretti, Luigi Genovese, Alberto Garcia, and Elisabetta Boella

Due date of deliverable: 30/06/2025 (month 30)
Actual submission date: 30/06/2025
Final version: 30/06/2025

Lead beneficiary: SIPEARL (participant number 12)
Dissemination level: PU - Public

Document information

Project acronym:	MAX
Project full title:	Materials Design at the Exascale
Research Action Project type:	Centres of Excellence for HPC applications
EC Grant agreement no.:	101093374
Project starting / end date:	01/01/2023 (month 1) / 31/12/2026 (month 48)
Website:	www.max-centre.eu
Deliverable No.:	D 4.3

Authors: Augustin Degomme, Lubomir Riha, Marc Sergent, Loris Lucido, Erwan Raffin, Filip Vaverka, Daniel Wortmann, Andrea Ferretti, Luigi Genovese, Alberto Garcia, and Elisabetta Boella

To be cited as: A. Degomme et al., (2025): First Co-design report. Deliverable D4.3 of the HORIZON-EUROHPC-JU-2021-COE-01 project MAX (final version as of 30/06/2025). EC grant agreement no: 101093374, SIPEARL, SiPearl SE.

Disclaimer:

This document's contents are not intended to replace consultation of any applicable legal sources or the necessary advice of a legal expert, where appropriate. All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user, therefore, uses the information at its sole risk and liability. For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Versioning and contribution history:

Version	Date	Author	Note
0.1	16/05/2025	A. Degomme	Preliminary draft
0.2	23/06/2025	A. Degomme	Siesta HBM part added
0.3	25/06/2025	E. Raffin	Internal review

D4.3 First Co-design report

Content

1. Executive Summary 5
2. Introduction 5
3. Analysis methods and platforms 7
    3.1. Targets description 7
        3.1.1. Rhea 7
        3.1.2. Alternative platforms 11
    3.2. Tools 13
    3.3. Analysis methodology 16
4. Status of Co-design in Max 18
    4.1. Mini Apps and extracted kernels 18
        4.1.1. Quantum ESPRESSO 18
        4.1.2. YAMBO 19
        4.1.3. BigDFT 19
        4.1.4. Siesta 20
        4.1.5. FLEUR 20
    4.2. ARM port and compiler optimization 21
        4.2.1. Target compilers and toolchains 21
        4.2.2. SiPearl compiler 23
        4.2.3. Statuses of the codes and miniapps 24
    4.3. Co-design and optimisation 30
        4.3.1. Quantum Espresso 30
        4.3.2. YAMBO 45
        4.3.3. BigDFT 51
        4.3.4. Siesta 58
        4.3.5. FLEUR 64
    4.4. Recommandations 68
        4.4.1. Quantum Espresso 68
        4.4.2. YAMBO 68
        4.4.3. BigDFT 69
        4.4.4. Siesta 69
        4.4.5. Fleur 69
5. Conclusion 70

1. Executive Summary

This deliverable is the first of task T4.1 of WP4. Its purpose is the exploration of the co-design opportunities for applications involved in the MaX project. The methodology exposed in this document will help to analyze the status of the MaX codes in terms of porting and targeting the best performance on the European processor Rhea from the EPI project, developed by SiPearl and that will equip the European Pilot for Exascale "EUPEX". Focus is the exploitation of Rhea characteristics such as SVE (Scalable Vector Extension) vectorization and high-speed memories, such as HBM, and the development of mini-apps to assess performance of the MaX applications on future platforms.

2. Introduction

To achieve best performance for MaX applications on future HPC hardware, collaboration with platform developers is essential at both the hardware and software level.

Code optimization for a specific platform is important but ensuring that the development of the platforms themselves benefits the codes in the long term is also critical. Each layer, whether hardware, software, compilers, or system libraries, may require adjustments to enhance overall performance and usability. European R&D initiatives such as the MaX project provide an excellent opportunity for such co-design activities, as all technology developers for future European platforms are members of the consortium and can provide useful insight to the other partners. This collaborative effort can follow an iterative approach, where the performance results of applications, technological advancements, and trade-offs in system design are continuously exchanged. Such a process enables informed decisions to shape the next generation of hardware and software effectively.

During the course of the project, T4.1 of WP4 is in charge of ensuring that MaX codes will take advantage of the characteristics of the Rhea CPU that will be at the heart of the EUPEX

platform, such as ARM's Scalable Vector Extensions (SVE) instructions, HBM memory or many-core design with up to 160 core per dual socket node.

For this task and pending Rhea's availability, full application performance will be analyzed on representative hardware, already providing some of these technologies. ARM SVE 256-bit vectorization is already available on AWS "Graviton 3" processors, and HBM memory is available on some of Intel's "Sapphire Rapids" class processors.

To dig deeper into the application performance needs, and to allow running on specialized development platforms for the hardware, full application may not be the most adapted target. MaX codes can have a lot of dependencies and require a specific set of inputs and some tuning to be efficiently used, increasing the risk of mistakes and the time spent in the co-design process. To allow running at smaller scales but also to improve reliability of the analysis, representative subsets of each MaX application have been extracted as Mini-Apps or single computing kernels.

Compiler compatibility and performance will also be studied for all codes and mini-apps, as the compiler is an extremely important tool to ensure performant use of a platform without the need to change the applications.

In this report we will present the methodology adopted in MaX for this analysis, as well as the current status of the extraction of the mini-apps of MaX codes. The first results of performance assessments of Mini-Apps and Max applications and the first feedback to developers will also be discussed.

3. Analysis methods and platforms

3.1. Targets description

3.1.1. Rhea

SiPearl's Rhea will be the first processor from the EPI initiative. Based on 80 power-efficient and powerful Arm Neoverse V1 cores, each including 256-bit wide Arm Scalable Vector Extension (SVE) vector support and providing in-package High Bandwidth Memory (HBM2e), Rhea will deliver extraordinary compute performance and efficiency with an unmatched Bytes/Flops ratio.

Rhea's goal is to address the full range of HPC workloads, including AI and Machine Learning. The high number of cores in modern processors presents a challenge for memory systems, as their size and bandwidth do not scale at the same speed. HBM provides a very high bandwidth and will help unlock performance of this many-core system for most of the existing codes, which are often limited by the memory system performance.

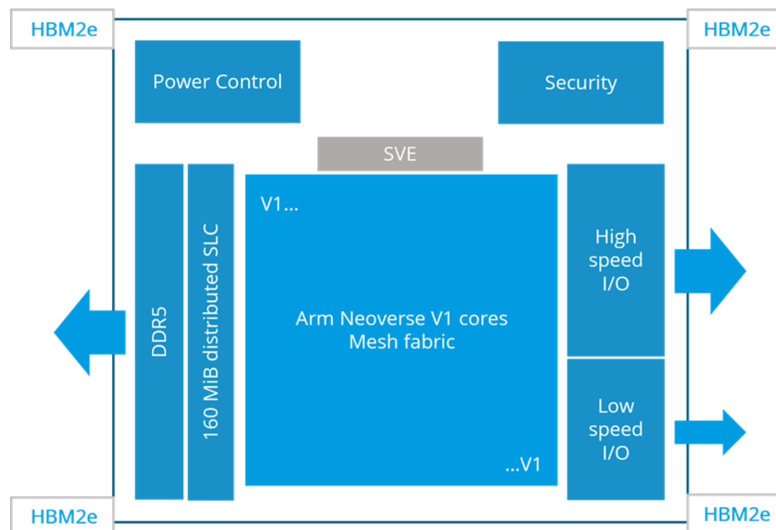

Fig. 1. Functional Diagram of Rhea.

As one can see in Fig. 1, all the Arm Neoverse V1 cores, including the SVE extensions, are connected through a Coherent on-chip network (Mesh fabric). This network includes PCIe and CXL to ensure smooth connection with accelerator and high-speed low latency networks, which

is mandatory for efficient HPC applications. This network also provides access to Last Level Cache (SLC, Arm terminology for Level 3 cache), DDR and HBM stacks to all the compute cores.

Rhea Features

- Core
  - 80 Arm Neoverse™ V1 CPU cores as compute unit building block.
  - SVE with 256-bit length per core supporting 64/32/BF16 and Int8.
  - Arm Virtualization Extensions.
- SoC:
  - Memory-coherent on-chip network provided by the Arm Mesh fabric.
  - Topology-aware design for scalability and flexibility.
  - Distributed last-level cache to memories with re-configurable NUMA domains.
  - Isolation between computing units.
  - Coherent SMP between chip domains.
- Memory:
  - HBM2e and DDR5 for memory bandwidth and capacity.
- Latest PCIe gen5/CXL/CCIX links for interconnect and accelerators.
- Low-power - Low latency links for die-to-die or chip-to-chip connections

SVE

Scalable Vector Extension (SVE) is an advanced vector instruction set architecture designed by Arm to enhance the performance and flexibility of SIMD (Single Instruction Multiple Data) processing. Unlike fixed-width SIMD architectures, SVE is unique in its scalability, supporting vector lengths that can range from 128 to 2048 bits in multiples of 128. This scalability enables SVE to adapt to various workloads and hardware implementations, optimizing performance across different applications, such as machine learning, scientific computing, and high-performance computing (HPC).

In the case of the Arm Neoverse V1 cores, a 256-bit variant of SVE is implemented, striking a balance between performance and power efficiency. By leveraging the 256-bit vector length, these cores deliver enhanced parallelism and computational density, crucial for workloads like data analysis and large-scale simulations. V1 core furthermore provides two SVE units per core, doubling the total instruction throughput also enables better utilization of modern memory

hierarchies and wider data paths, boosting overall throughput without significantly increasing energy consumption.

<table>
<tr><th>Neon vector loop example</th><th>SVE vector loop example</th></tr>
<tr><td>

```
#include <arm_neon.h>

void neon_vector_add(double *a, double
*b, double *c, int n) {
 for (int i = 0; i < n; i += 2) {
  float64x2_t vec_a = vld1q_f64(&a[i]);
  float64x2_t vec_b = vld1q_f64(&b[i]);
  float64x2_t vec_c = vaddq_f64(vec_a,
vec_b);
  vst1q_f64(&c[i], vec_c);
  }
}
```

</td><td>

```
#include <arm_sve.h>

void sve_vector_add(double *a, double *b,
double *c, int n) {
 svbool_t pg = svptrue_b64();
 for (int i = 0; i < n; i += svcntd()) {
  svfloat64_t vec_a = svld1(pg, &a[i]);
  svfloat64_t vec_b = svld1(pg, &b[i]);
  svfloat64_t vec_c = svadd(vec_a,
vec_b);
  svst1(pg, &c[i], vec_c);
 }
}
```

</td></tr>
</table>

Table 1. NEON to SVE example.

In example Table 1., we can see that portability effort from Neon code to SVE code can be relatively low, while providing future-proof performance portability with future CPUs. Using predicates (pg) instead of fixed vector size is the main difference, and allows the same code and the same executable to run efficiently on processors supporting from 128-bit to 2048-bit vector sizes.

In the context of MaX, ensuring good SVE vectorization of codes and miniapps is key, as the efficient use of vectors can provide an important advantage in terms of performance for current and future ARM-based processors. In this regard a lot of effort will be dedicated to analyze code behaviour, compiler tuning and results in terms of performance.

Memory system

A memory system combining DDR5 (Double Data Rate) and HBM2e (High-Bandwidth Memory 2nd generation enhanced) offers a powerful solution for high-performance computing (HPC) applications. DDR memory provides large capacity and cost-effective scalability, making it ideal for general-purpose workloads. On the other hand, HBM2e delivers exceptional bandwidth and energy efficiency, thanks to its 3D-stacked architecture and proximity to the processor. By integrating both types of memory, systems can leverage the strengths of each, optimizing performance for diverse HPC tasks.

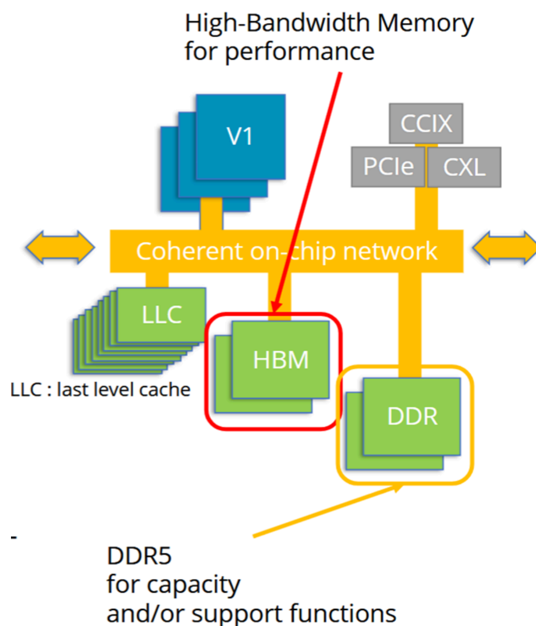

Fig. 2. Rhea building blocks.

Providing several memory systems with different characteristics brings new questions. Indeed Non Uniform Memory Access (NUMA) systems have been around for decades and efficient use of their characteristics have been a major issue on HPC systems. Placement of the data in each NUMA node to maximize performance and efficiency is critical. With HBM and DDR available, several modes of use can be envisioned, such as HBM-as-cache, or NUMA aware tools can be provided to analyze or move data at runtime to ensure efficient use of the memory system. In the context of MaX, studies about each code behaviour with HBM will be performed, and IT4I is developing a tool to efficiently help users make use of heterogeneous memory access systems.

HBM/DDR usage Modes:

- HBM as cache: HBM is not directly visible for users of the system, and operating system or vendor-provided modules handle data movement from and to HBM, which is seen as a last level of cache. As in other cache levels, last accessed data is kept and can be accessed much faster. Advantages are versatility, possibility to use full DDR size, and low adaptation cost for end users, at the cost of a small fraction of performance compared to direct HBM usage in some cases, and losing ability to use HBM to expand total memory size. All codes may not be able to benefit from performance boost as well, depending on the access patterns and sizes. This feature will not be available in the first version of Rhea, but its pertinence can be assessed.

- Only HBM or DDR: in other modes, all HBM nodes are exposed to the application level, adding to the DDR. The application can run on HBM directly or DDR only, allowing selection. In the case of full HBM usage, performance can be optimal, at the cost of size. Indeed HBM is smaller than DDR usually, so using only HBM limits the amount of memory accessible and the size of the systems codes are computing.
- NUMA-aware placement: with both HBM and DDR available, toolchains and users can perform placement in order to preserve both optimal performance and maximum size of the systems codes can run. With either manual or automated analysis, offline or at runtime, identifying the data that can be safely and efficiently placed in HBM and/or moved from it can yield significant performance improvements. With some tools, after analysis, code can be modified under supervision of developers, to add hints for compilers and toolchains to perform adequate placement on such systems. Other tools may rely only on dynamic analysis of behaviour at runtime.

3.1.2. Alternative platforms

As Rhea platform is still under development, it won't be available to test MaX codes and Mini-apps until later in the project. Alternative platforms have been selected for WP4 to test portability and performance of the application with a good level of similarity.

Portability to ARM platforms can be assessed using one of the numerous ARM-based processors already in the market nowadays, such as Ampere Altra (80 core Neoverse-N1 with NEON but no SVE support), Fujitsu A64FX, or Nvidia Grace.

Fujitsu's A64FX, also based on Arm architecture, provides 512-bit wide SVE support and HBM, and has been available for several years in some computing centers, being at the heart of Fugaku's supercomputer. It features a very specific core, and a complex memory system to exploit fully. Its proprietary toolchains are also not a target for the codes in the context of MaX, and their availability is not ensured for the duration of the project, as they are being phased out. EUPEX used A64FX as a SDV platform for software stack development and first experiments but now focus on Grace based platforms (Nvidia).

The Amazon AWS Graviton3 is the third generation of Arm-based processors designed by Amazon Web Services (AWS). These processors are tailored for cloud workloads but also for HPC usages, as AWS provides HPC cloud-based clusters. They are based on the same Neoverse-V1

core that will be used in Rhea, making it a prime target for portability and performance evaluation, mainly for the SVE vectorization effort.

Nvidia Grace is a 72 core Neoverse V2 CPU, providing 128-bit wide SVE support (compared to 256 in V1). Its original purpose is to be coupled to Nvidia's own GPUs, but its intrinsic performance is still very high, and Nvidia releases dual-socket systems without GPU. This highly efficient and powerful ARM CPU shows that many vendors are now considering ARM-based CPUs for their performance needs, validation choices made for Rhea. Several of these CPUs are available for use for MaX members thanks to E4.

To assess HBM performance gains, Intel Sapphire Rapids were retained as the prime target. Indeed the Intel Max line of CPUs integrates up to 56 core, with HBM2e into the package, much like Rhea, and benefits from these bandwidth gains. They can be configured to use HBM as cache or as separate NUMA nodes (SNC4 mode), allowing study placement methods and tools to automatize efficient use of HBM. MaX codes have been proved to be working on Intel platform and with Intel toolchains for years, so portability is not an issue.

System	Ampere Altra	AWS Graviton 3	SiPearl Rhea 1	Nvidia Grace	Intel Xeon MAX 9460/9468/9480
Core type	Neoverse-N1	Neoverse-V1		Neoverse-v2	Sapphire Rapids
Architecture	ARM v8	ARM v8	ARM v8	ARM v9	x86_64
Memory Type	DDR-4	DDR-5	DDR5, HBM2e	LpDDR5	DDR5, HBM2e
SIMD/core	2 x 128	2 x 256 or 4 x 128		4 x 128	1 x 512
L1	10 MB	10 MB	10 MB	9 MB	80KB/core
L2	80 MB	64 MB	80 MB	72 MB	2MB/core
SLC	32 MB	32 MB	80 MB	124 MB	98MB
Number of cores	80	64	80	72	40/48/56 with Hyper Threading
Extension	NEON	NEON, SVE	NEON, SVE	NEON, SVE, SVE2	AVX2, AVX512

Table 2. T4.1 main characteristics of the alternative platforms,, compared with Rhea.

3.2. Tools

Maqao

MAQAO[1] (Modular Assembly Quality Analyzer and Optimizer), developed by UVSQ (University of Versailles St Quentin), is a performance analysis and optimization framework designed to operate at the binary level. Its primary goal is to assist developers in improving the performance of their applications by providing detailed insights and recommendations. MAQAO combines both static and dynamic analyses, enabling it to reconstruct high-level structures such as functions and loops from application binaries. This makes it agnostic to the programming language used in the source code, as it does not require recompilation for analysis.

The tool includes several modules tailored for specific tasks. For example, the LProf module is a lightweight profiler that identifies performance hotspots at the function and loop levels, while the CQA module evaluates the quality of the code generated by the compiler. Additionally, the ONE View module aggregates results from other modules to provide a comprehensive overview of application performance. MAQAO also supports advanced features like value profiling (VProf) and decremental analysis (DECAN), making it a versatile tool for performance tuning.

IT4I HBM

The IT4I HBM tool[2] complements tools such as HWLOC by enabling fine grained control over memory placement of individual objects (allocations) created by the application. This level of control allows us to analyze and optimize placement of each individual object so that a good match between object usage patterns and memory type it resides in can be achieved (e.g. latency sensitive objects can be kept in DDR memory pool, while bandwidth sensitive ones may be placed in HBM memory pool).

[1] https://maqao.org
[2] Filip Vaverka, Ondrej Vysocky, Lubomir Riha, *Heterogeneous Memory Pool Tuning*, https://arxiv.org/abs/2505.14294

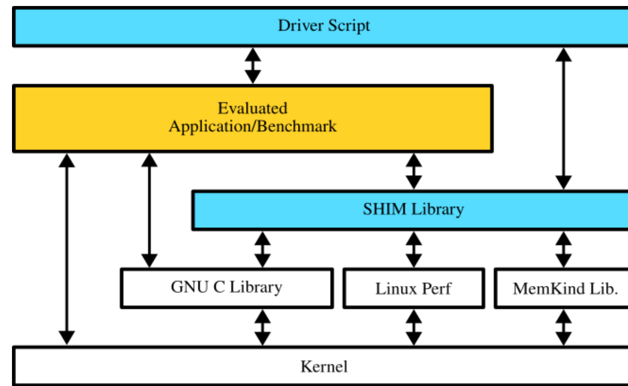

Fig. 3. IT4I HBM tool architecture overview.

The tool is lightweight and (generally) does not require modifications or recompilation of the analyzed application. It consists of “driver script” and “shim library”, where the former controls the object placement, while the latter intercepts object allocation calls in the application and places the allocated objects into requested memory pools. The library integrates Intel PEBS hardware monitoring (Linux perf) to collect statistics (such as memory access latency) for individual objects.

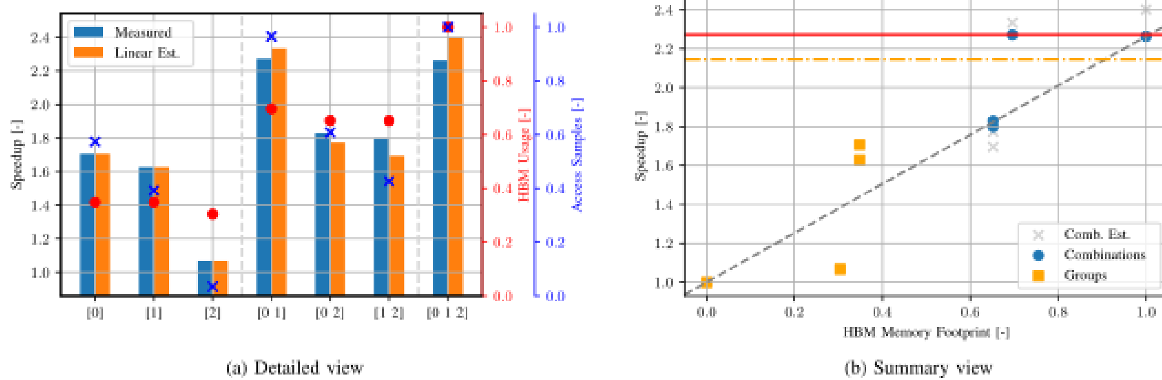

Fig. 4. IT4I HBM tool analysis output.

The IT4I HBM tool analysis output figure illustrates capabilities of the tool on an analysis of the Multi-Grid benchmark of the NAS Parallel Benchmark Suite (NPB) for all possible placements of significant allocations. The detailed view shows 3 allocations in 7 DDR/HBM placement configurations (DDR-only placement is used as a reference for speedup). The bars denote measured (blue) and expected (orange) speedup of each configuration, while red dots and blue crosses denote fraction of the data in HBM and fraction of memory accesses to the data in HBM (IBS/PEBS sampled) respectively.

The summary view focuses exclusively on the relationship between speedup and fraction of the application data in HBM. The results are shown as a scatter plot with yellow squares denoting results for individual allocations in HBM plus the DDR-only case, while blue dots denote combinations of two or more allocations in HBM (corresponding to blue bars). The gray crosses show expected speedup (corresponding to orange bars). Finally, the horizontal lines denote the maximum (solid red line) and 90 % of the maximum (dash-dotted orange line) speedup.

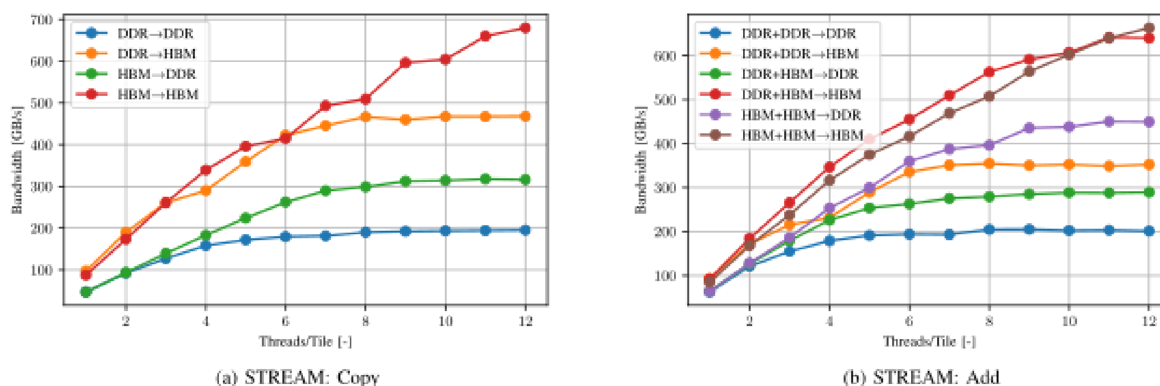

Fig. 5. Memory bandwidth of Copy and Add sub-tests of STREAM benchmark in relation to placement (DDR or HBM) of each work array (16 GB per array).

The STREAM benchmark is a good way to further illustrate potential intricacies of dual-memory pool platforms such as Intel Sapphire Rapids with HBM. The figure shows the bandwidth scaling of STREAM's *Copy* and *Add* sub-tests for each data placement configuration. While the cases where all the data are placed in DDR or HBM memory pools reflect the peak bandwidth of respective memory pools, the situation is more interesting for mixed cases. The copy kernel performs considerably worse when copying from HBM to DDR memory than the other way around, achieving only about 65% of expected bandwidth. The behavior is similar for the Add kernel, where reading two arrays from HBM memory and writing the result to DDR memory (HBM + HBM→DDR) performs similarly to its complementary configuration (DDR + DDR→HBM). On the other hand, we can achieve HBM-only performance while storing one of the input arrays in DDR memory (saving a third of the limited HBM memory capacity).

These results suggest that similar situations may arise in full applications, allowing to maintain high performance for workloads that may not fit in the limited HBM memory pool.

3.3. Analysis methodology

HBM vs DDR

For the HBM work presented in this report, the main platform used is the Intel SapphireRapids HBM presented in Alternative platforms. This processor has several configuration modes for both the cores and memory, as shown below.

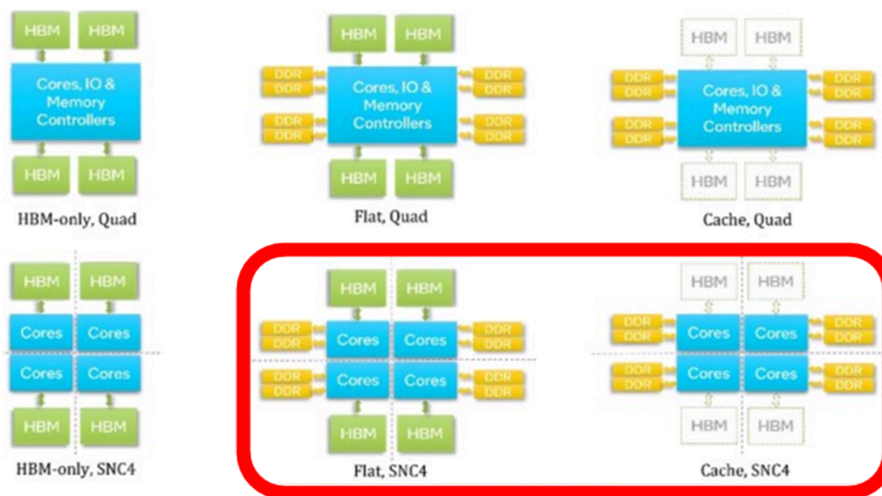

Fig. 6. Intel SapphireRapids available core and memory configuration modes.

We discuss first the core subdivision mode used. The two available for Intel SapphireRapids are called Quad and SNC4. In Quad mode, all the cores are seen by the operating system as a single monolithic entity, and all memories are attached to this single entity. In SNC mode (that can be either 2 or 4), the cores are subdivided into multiple entities and shown to the operating system. This mode is often preferred as it exposes the NUMA nature of the processor and allows proper NUMA binding. We thus used the SNC4 mode for all the studies presented in this report.

Moreover, three configuration modes of the HBM memory are available for Intel SapphireRapids. The HBM-only mode only exposes HBM memory and not the DDR memory, which is not interesting for the study we aim to perform. The Flat mode exposes the HBM as a dedicated NUMA node next to the DDR NUMA node, and tools that perform NUMA memory binding can directly address them. In Cache mode, the HBM is seen by the operating system as a new top-level cache in the cache hierarchy. Thus, for the experiments presented in this report,

we used both Flat and Cache mode when appropriate to investigate performance of MaX codes and mini-apps under both configurations.

We perform the memory binding with the Hwloc tool (Hardware locality), that exposes DDR and HBM NUMA nodes separately for binding, as shown below in Fig. 7. Moreover, it can automatically compute which memory to use through the `hwloc-calc` command line tool depending on a required binding criteria, such as latency and bandwidth. With this, we can easily produce a command line that will either bind to HBM or DDR depending on latency/bandwidth criteria. To validate the approach, we assessed the performance of DDR and HBM NUMA nodes with the STREAM TRIAD benchmark. With the proposed binding method, we obtained 61GB/s for DDR5 and around 208GB/s for HBM, which is coherent with the values presented by the vendor for this setup.

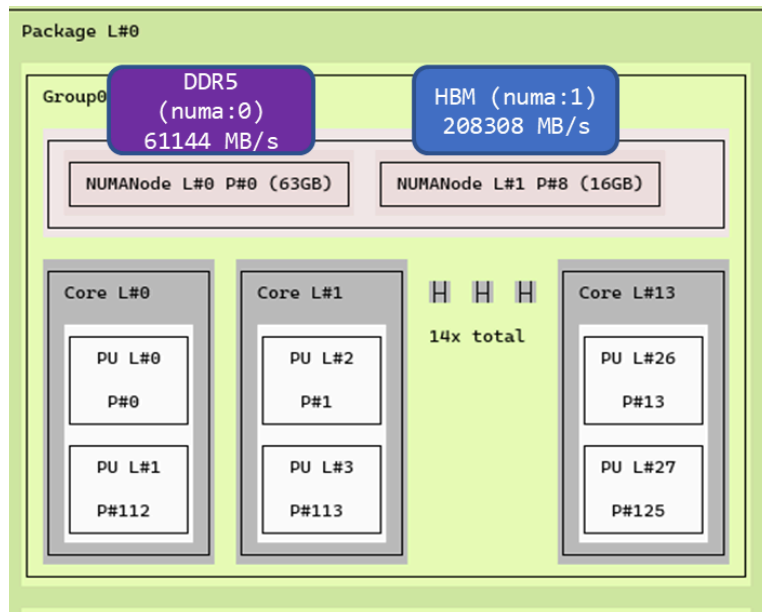

Fig. 7. Hwloc graphical output of a single SNC4 module of the Intel SapphireRapids processor.

With this in mind, to compare HBM and DDR performance, we perform a study on either the full mini-application (if available) or the original code by applying both HBM and DDR memory bindings and comparing them. Then we explore a more localized approach with the IT4I HBM tool introduced in Section 3.2 (if applicable).

Vectorisation

For the vectorisation study, we use the MAQAO tool introduced in Section 3.2 to explore vectorisation metrics. The goal is to compare between the x86 best compiler (often Intel) with GNU to improve the vectorisation of GNU for the x86 world and bring it as close as possible to the best compiler performance. From this point, we can compare GNU between microarchitectures (x86 and aarch64) to assess that the identified vectorisation is similar and that vectorised instructions for ARM (especially SVE) are properly generated and efficient. Finally, we want to compare GNU with LLVM-based compilers (such as ACFL) on ARM ecosystem to assess vectorisation on such compilers, which will be used for Rhea compiler developed by SiPearl, and eventually suggest how to improve vectorisation of mini-app for these compilers.

4. Status of Co-design in Max

4.1. Mini Apps and extracted kernels

During the first phase of the project, the main output of T4.1 is the extraction of mini-apps or kernels from MaX codes, to simplify analysis of representative portions of each code and enable focused feedback for developers. These mini-apps, build and running instructions and guidelines are gathered in the repository max-centre / Components / Mini-apps · GitLab . Recipes for building mini-apps on the various platforms used for analysis are also gathered there.

4.1.1. Quantum ESPRESSO

Quantum Espresso provided two mini-apps, with a third one in progress. The first miniapp tests the functionality of FFTXLib in the so called 'Rho' modality that is used for performing distributed 3D FFTs for the charge densities, and potentials within QE application.
It generates a distributed 3D data set in G space and transforms it back and forth performing a 3D FFTD. This is a lightweight mini-app suited for detailed profiling and analysis with emulators and tools with significant overhead. Together with the library FFTXlib which must be recompiled and linked it also tests the Distribution of 3D FFT Grids.
The second miniapp generates many 3D data sets in G space, distributed in the Wave function grid of QE, and transforms all of them back and forth performing 3D FFTs (in wave modality).
It reproduces the operation of FFTXlib for many wave functions and allows to explore trade-off between R&G, band distribution, and the speedup for the execution with many functions modality inside QE with many FFTs in batched mode.

Both of these mini-apps have already been integrated in SiPearl's internal testing environment for running on Rhea emulation hardware. Due to scheduling constraints, at the time of this writing these experiments have not yet been run.

4.1.2. YAMBO

One of the key kernels of the many-body perturbation theory methods implemented in the Yambo code is the calculation of the independent particle (IP) density-density response function (Xo, in the Yambo sources). For each transferred momentum q, the computation of Xo(q) is implemented as a sum over valence-conduction transitions, and is represented as a matrix depending on G,G' plane waves (which can be in the order of few hundreds to thousands and tens of thousands). Multiple frequencies (w) can be handled jointly. In practical simulations, the number of transitions (controlled by the number of bands included in the calculation) and the number of plane waves, which can be regarded as convergence parameters, control the time- and memory-to-solution of the kernel. Since the evaluation of Xo(q,w) is the leading contribution in the GW timing of Yambo, as well as also very relevant for the Bethe Salpeter equation, it was decided that Xo was the first mini-app to be extracted from the Yambo code. At present, the mini-app initializes wavefunctions and dipoles with random data, and then goes through the calculation of Xo for multiple q and frequencies with the dimensions taken from realistic datasets. Current capabilities covers many-core with openmp support as well as Nvidia GPUs via the CUDA-Fortran programing model. OpenACC and OpenMP-GPU support is ongoing.

4.1.3. BigDFT

Within the BigDFT electronic structure code, a Fock mini-app has been extracted to represent one of the most computationally demanding components of hybrid functional calculations, especially under the PBE0 approximation. This kernel evaluates the Fock exchange operator, which involves repeated applications of a Poisson solver and is highly representative of the memory and computational behavior of the full application. As such, it serves as an ideal proxy for architecture-specific performance tuning and benchmarking.

In the context of co-design for future platforms, this mini-app has been ported to SYCL to support execution on Intel GPUs, allowing exploration of heterogeneous memory hierarchies (e.g., HBM) and high-throughput compute nodes. The SYCL version leverages advanced memory handling techniques, including efficient zero-padding for free boundary conditions via callbacks,

and makes extensive use of optimized FFT libraries. Performance comparisons show that the SYCL port is both portable and competitive, often outperforming the legacy CPU and CUDA implementations on relevant workloads.

This kernel is being actively used to assess the performance impact of HBM-bound workloads, cache usage, and communication bottlenecks in multi-node deployments. Its profiling reveals a range of memory intensities and arithmetic intensities depending on the workload size, with larger systems being more bandwidth-bound — a key insight for guiding architectural choices and compiler strategies on future ARM-based many-core nodes like Rhea. The modularity and relative independence of the Fock mini-app make it a valuable tool for both performance analysis and co-design feedback loops within the MaX project.

4.1.4. Siesta

The lion's share of the computing time in a Siesta execution is taken by the solver stage. The most used solver technique is diagonalization, a generalized eigenvalue problem typically involving Cholesky decomposition, conversion to a tri-diagonal form, solving, and appropriate back transformation. The diagon mini-app supports various methods: divide and conquer and MRRR from (Sca)LAPACK, and those in the ELPA library (which can also be executed on GPUs). Hamiltonian and Overlap matrices for test cases are provided in files, and the mini-app supports the direct setting of parameters such as the block size and the number of eigenvectors to compute, as well as method selection. Additionally, a kernel for the computation of the density matrix from the eigenvectors can optionally be exercised. It is expected that the mini-app will also be useful as a tool in the preparation of a performance model able to predict approximately the computing time for a given problem size and architecture.

4.1.5. FLEUR

FLEUR uses a multitude of rather complex Fortran data types which are input to most of the computational kernels. This makes the generation of mini-apps challenging, since these data types have to be initialized properly and most of them can not be easily stored. Hence, we follow different approaches to generate mini-apps. On the one hand, we have a few computational relevant parts that can be called without the need of extensive initialization of these data structures. Here we can make rather small stand-alone mini-apps that are close enough to the production code to be useful. Most notable in this respect is a mini-app which encapsulates the solution of the generalized eigenvalue problem which is central to the runtime

of the code. This mini-app only requires a few of the FLEUR functionality and thus a very simple build is possible. More complex in this respect is a second mini-app that focuses on the matrix redistribution from the layout used in the matrix setup to that needed in a distributed matrix diagonalization. This routine, while not very relevant in smaller setups, can be the source of a bottleneck when it comes to scalability to large simulations. This second mini-app is also equipped with a build mechanism that is designed to resemble the main features of the main build. In addition, we have the possibility to run the full FLEUR code only up to the point in which the matrix is diagonalized. This therefore means that we have a “mini-app” that is not “mini” as the full code is used, but tests basically the matrix setup only and thus only a small part of the full code.

4.2. ARM port and compiler optimization

4.2.1. Target compilers and toolchains

ARM platforms are capable of running the majority of software available on Linux systems, thanks to extensive efforts over the past decade to enhance application portability. This includes compilers and toolchains, which play a crucial role in ensuring both compatibility and optimal performance for applications like the MaX codes.

GNU compilers have long been supported on ARM platforms, and LLVM toolchains have also been available, though they initially lacked a functional Fortran compiler—essential for many HPC codes like the MaX ones. In addition, HPC vendors provide comprehensive toolchains alongside their processors, including C/C++/Fortran compilers and specialized libraries for domains such as linear algebra (BLAS/LAPACK), FFT, and MPI communication.

ARM provides the Arm Compiler for Linux toolchain, featuring an LLVM-based C/C++ compiler and a PGI-based Fortran compiler, complemented by ARM performance libraries that provide vectorized implementations of many common HPC operations. Similarly, with the launch of their Grace processor, Nvidia greatly enhanced the work on their HPC-focused toolchains for ARM processors, including NVHPC compilers—LLVM-based for C/C++ and PGI-based for Fortran. Nvidia's performance libraries (NVPL) have also been optimized to boost performance on Grace processors, making them appealing for Neoverse-V1-based processors like Rhea.

In recent years, vendor-specific compilers have increasingly aligned with LLVM-based solutions, with vendors contributing modifications to the upstream LLVM compiler while retaining some

optimizations in their proprietary versions. However, the absence of a modern Fortran compiler in LLVM prompted Nvidia to initiate an effort to integrate one into the upstream project. Originally named F18, this compiler is now fully incorporated into LLVM as Flang. As a result, the older PGI Fortran compiler previously used by ARM and Nvidia is being phased out, with future versions of their toolchains transitioning to Flang as the foundation for Fortran compilation. A preliminary version of the future ARM flang-based fortran compiler was made available (named ARM toolchain for linux), and was tested with some version of MaX codes.

Building a compiler from scratch for a language as established as Fortran—while ensuring support for the various iterations of Fortran specification and high-performance executable generation for HPC applications—is an immense undertaking. Flang remains under active development, with ongoing efforts to resolve issues and improve both portability and efficiency. SiPearl is contributing to this initiative, as the toolchain accompanying Rhea will be LLVM-based, with Flang serving as the designated Fortran compiler.

Within the scope of WP4 and T4.1, ensuring that MaX codes and their mini-apps compile, execute, and perform optimally on ARM-based platforms—both with existing and upcoming toolchains—is a key objective. Compilers play a crucial role in optimization, including vectorizing code to leverage available hardware efficiently without requiring direct code modifications. Verifying that the compiler performs these tasks effectively can significantly reduce development time for application developers.

At the moment of writing this document, recipes have been written to allow compilation of MaX codes with several toolchains and sets of HPC libraries. As the compilers and toolchains have evolved since the beginning of the project, multiple versions have been tested, only the latest will be reported.

Tested compilers for ARM targets:

- GNU compilers (gcc/g++/gfortran), versions ranging from 13 to 15;
- ARM compilers (armclang/armclang++/armflang), from 22.04 to 24.10, and ATfL (ARM toolchain for linux, prerelease with flang based on LLVM 19);
- Nvidia compilers (nvc/nvc++/nvfortran), 23.11 to 25.3;
- SiPearl LLVM compilers (clang/clang++/flang), 17 to 21.

For Intel targets used to test HBM performance gains within the scope of this project, Intel compilers have also been used. Since 2023 these have been also LLVM-based for the C/C++

compilers. Intel Fortran compiler, ifx, is a competing effort to provide a modern proprietary Fortran compiler, based on LLVM like flang, while retaining decades of work on previous ifort compilers from Intel. Tested versions ranged from 2023.1 to 2025.1.0.

Several optimized HPC set of libraries have also been tested:

- ARM performance libraries ArmPl (22.04 to 25.04);
- Nvidia Performance libraries nvpl (23.11 to 25.1.1);
- OpenBLAS (0.3.25 to 0.3.29)/BLIS (development version)/FFTW (development version with SVE optimization) libraries,which are open source versions of linear algebra or FFT libraries, and provide optimizations for ARM, have also been tested. In some cases they have been shown to provide equivalent/better performance than proprietary solutions.

For Intel targets, GNU compilers and Intel's oneAPI toolkit have been used (it includes the former MKL set of optimized libraries).

In order to assess communication patterns and evaluate scaling, MPI libraries can also be a critical factor.Multi node behaviour is out of the scope of this WP, but even on single node with many cores, hybrid computing using both MPI and OpenMP is a widespread way of using the CPU at its best, and all Max codes support it. OpenMPI (4 or 5) was the main tested version until now, with Intel's proprietary MPI being used also on Intel platforms.

4.2.2. SiPearl compiler

LLVM is today the state of the art compiler tailored for ARM platforms, paving the way for the most efficient use of Rhea-based systems. Nonetheless, in the context of HPC applications, some additional optimisation could be integrated to exploit the best of AArch64 and SVE. In this context, SiPearl contributed to strengthen this open source toolchain by the adjunction of several features, that are mostly in the process of being released to the Community.

C and C++ compilers from LLVM are already mature targets and work perfectly with most codes, while the Fortran compiler is new and still under stabilization. SiPearl is contributing to this effort by reporting and fixing as many issues as possible to ensure that most HPC codes of the community are supported, both functionally and performance-wise, including MaX codes.

Among the specificities of SiPearl compilers (upstreamed or yet to be upstreamed) are:

- Full scheduling information for V1 cores;

- Better complex number support;
- Several optimization directives for guided unrolling/vectorization in fortran;
- Support for many intrinsics (erfc_scaled, derfc, ierrno, chdir, ...);
- Vectorized intrinsics for fortran math functions;
- Outer-loop vectorization support with tail-folding, reductions, and first-order recurrences;
- Many vectorization optimization to ensure best use of SVE.

4.2.3. codes and miniapps status

In this section, compilation and execution statuses of all codes will be reported for various ARM platforms. When errors are encountered at compilation time or run time, and when large adaptations (not just a compilation flag) are necessary, this will be reported. All tests were performed using various OS, mainly RHEL or Ubuntu systems, and with MPI and OpenMP both activated. Vectorization options for each platform were activated.

Color code:
OK means code compiles and executes without modification.
OK means code compiles and executes with minor modification (which has been/will be reported to developers or compiler teams).
OK means code compiles and executes only with major modification.
NOK means code won't compile or run, and explanation will be given on the observed issues.

Platforms used are Ampere Altra for Neoverse-N1 (no SVE, only Neon), Graviton 3 for Neoverse-V1, and Nvidia Grace for Neoverse-V2.

Quantum Espresso

- **Application**

Quantum Espresso was found to be generally compatible with most compilers and made some changes recently to improve support of Nvidia compilers or flang.
Version tested: devel from https://gitlab.com/QEF/q-e , commit 0b9715eb from 10/05/25.
Inputs: ausurf with pw.x executable.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK	OK	OK	OK	OK
Neoverse-V1	OK	OK	OK*	OK	OK
Neoverse-V2	OK	NOK**	OK*	OK	OK

* since latest LLVM21-based release ("Instruction does not dominate all uses" OpenMP issue fixed).
** "VPlan cost model and legacy cost model disagreed" error when vectorizing, only with V2 target for Arm's preview flang compiler.

- **MiniApps**

Version tested: commit 52b356e9 from https://gitlab.com/max-centre/components/mini-apps
Inputs: inputw_large/input_large for miniapp1 and 2.
Same result for both QE miniapps:

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK	NOK*	OK	OK	OK
Neoverse-V1	OK	NOK*	OK	OK	OK
Neoverse-V2	OK	NOK*	OK	OK	OK

* Arm experimental compiler compiles the miniapps but there are crashes at runtime

YAMBO

- Application

Tested version: 5.2.4 from https://github.com/yambo-code/yambo/.
Only compilation was tested currently for the main app, as test cases necessitate QE preprocessed data that was not available at the moment of the experiment.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK	OK	OK	OK	OK
Neoverse-V1	OK	OK	OK	OK	OK
Neoverse-V2	OK	OK	OK	OK	OK

- MiniApp

Version tested: 52b356e9 from https://gitlab.com/max-centre/components/mini-apps.
Inputs: dataset_1.dat -ng 3000 -nb 200 -nq 10 -nw 2 only with openMP.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK	OK	OK	OK	NOK
Neoverse-V1	OK	OK	OK	OK	NOK
Neoverse-V2	OK	OK	OK	OK	NOK

Miniapp compiles and executes, but there are still reliability issues, as output is not stable. Ongoing work is done to stabilize the mini-app, in coordination with the developers. The latest release should fix these issues.

Nvidia build but version triggers segmentation fault when deallocation data during the run, localized at line X_irredux.f90:588 in “pgf90_dealloc_mbr03a_i8”.

BigDFT

- Application

BigDFT devel commit a1854369 from https://gitlab.com/l_sim/bigdft-suite.
Inputs H20-32 from https://gitlab.com/max-centre/benchmarks.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK*	NOK	OK**	OK	OK
Neoverse-V1	OK*	NOK	OK**	OK	OK
Neoverse-V2	OK*	NOK	OK**	OK	OK

* Arm Compiler compiles BigDFT but triggers crashes when deallocating some structure, at the end of iterations or after the end of execution. Removing these deallocations allows to compile and get correct results, at the cost of memory usage.
** A heavily modified version of BigDFT was able to compile and run correctly with LLVM Sipearl's compiler, but many modifications had to be performed in the process. These are being investigated by Sipearl's team and BigDFT developers.

- MiniApp

Fock miniapp is based on some of BigDFT libraries and shares its build system and utilities. The repository version is the same. Inputs: -n 64.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	OK	OK*	OK*	OK	OK
Neoverse-V1	OK	OK*	OK*	OK	OK
Neoverse-V2	OK	OK*	OK*	OK	OK

* Both ARM ATfL compiler and LLVM reports deallocation errors at the end of execution, but only after reporting correct results. This hints at a deallocation issue within the runtime of these compilers.

Siesta

- **Application**

Siesta MaX build relies on external libraries such as ELSI, ELPA, Pexsi or Superlu. These dependencies cannot all be built with each compiler. For example on neoverse-V2, LLVM based fortran compilers (sipearl and ATfL) will fail to compile ELPA libraries. In these cases these dependencies were built with GNU compilers. This is not reported in the table as these dependencies were built externally (they can be built internally with siesta, but it would fail in these cases).
Version: devel commit 5d891ece3 from https://gitlab.com/siesta-project/siesta.
Inputs: water-box.fdf from https://gitlab.com/max-centre/benchmarks.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	NOK*	NOK**	OK	OK	OK
Neoverse-V1	NOK*	NOK**	OK	OK	OK
Neoverse-V2	NOK*	NOK**	OK	OK	OK

* Arm compiler builds but fails at runtime.
** AMR LLVM compiler on V2 triggers “VPlan cost model and legacy cost model disagreed” error. Recent LLVM SiPearl based on LLVM 21 works, while previous builds triggered “instruction does not dominate all uses” errors.

- **MiniApp**

Version commit 52b356e9 from https://gitlab.com/max-centre/components/mini-apps.
Inputs: nanoscroll-dzp and covid-8k.

	ARM 24.10	ARM ATfL	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	NOK*	OK**	OK	OK	OK
Neoverse-V1	NOK*	OK**	OK	OK	OK
Neoverse-V2	NOK*	OK**	OK	OK	OK

* Arm compiler builds but fails at runtime.
** MPI version fails, but sequential version OK.

FLEUR

- **Application**

Develop version, commit c5d564c1 from https://iffgit.fz-juelich.de/fleur/fleur.
Input:
https://gitlab.com/max-centre/benchmarks/-/tree/master/FLEUR/inp_0.33/fleur_tiny_NaCl.

	ARM 24.10	ARM ATfl	LLVM Sipearl dev	GNU 15	Nvidia 25.3
Neoverse-N1	NOK*	NOK*	OK**	OK	OK
Neoverse-V1	NOK*	NOK*	OK**	OK	OK
Neoverse-V2	NOK*	NOK*	OK**	OK	OK

* ARM compilers : builds, but « Invalid address alignment » immediately at runtime.
** Between versions 20 and 21 of the SiPearl compiler, a regression was detected which caused “Invalid address alignment“ errors when building FLEUR. The issue was reported and fixed in the latest version.
** Volatile keyword is still not supported in procedure interfaces with flang. SiPearl team is notified, but code compiles when removing it.

4.3. Co-design and optimisation

4.3.1. Quantum Espresso

All the work presented in this section was performed on the second mini-app of Quantum Espresso as introduced in Section 3.1.1. We used the commit 86f0dd38 (2024-04-30) of the mini-app repository. All runs were executed with the *input_very_large* input file to have enough execution time to properly explore performance statistics.

HBM vs DDR

All tests were performed on an Eviden in-house cluster equipped with Intel Sapphire Rapids processors with HBM as described in Section 2.1. The following Table 3 shows the system and software configuration used for these runs.

OS	Compiler	BLAS	FFT	MPI	Tools
RHEL 8.8	Intel OneAPI 2023.2.0	Intel MKL 2023.2.0	Intel FFTW 2023.2.0	Open MPI 4.1.6 UCX 1.16.0	Hwloc 2.10

Table 3. System and software stack used for HBM vs DDR experiments.

We specified the following options to the CMake build system of the miniapp:

```
-DCMAKE_C_COMPILER=icx
-DCMAKE_Fortran_COMPILER=ifx
-DQE_ENABLE_OPENMP=1
-DQE_FFTW_VENDOR=Intel_FFTW3
```

To properly compare HBM with DDR, we chose to use only 14 cores out of the 56 available to saturate a single SNC NUMA node. By doing so, we want to ensure that only the same NUMA nodes close to the cores chosen are used to properly compare only the characteristics of the memory nodes.

To ensure a proper binding on NUMA nodes, we used the Hwloc tool to perform the core and memory binding. This tool has the advantage to include a companion tool called hwloc-calc that allows computing automatically the NUMA node associated with a core binding that delivers the best performance for a specific attribute, such as bandwidth or latency. For our

experiments, the best bandwidth will be the HBM NUMA node, and the best latency will be the DDR NUMA node. By exploiting this property, we can ensure with Hwloc that a proper memory binding is made for our experiments.

From there we used two types of launch, either a full MPI with 14 MPI process with 1 OpenMP thread per process, and a full OpenMP one with 1 MPI process and 14 OpenMP threads. The goal here is to assess if the two parallelization methods have different requirements in terms of memory and if the HBM behaves differently with one of them.

The results we obtained are shown below in Fig. 8. For the full MPI setup we observe an execution time reduction of 33,2%, and of 17.5% for OpenMP with HBM. We can then argue that the HBM is very beneficial for this mini-app, but the OpenMP parallelization scheme seems to take less advantage of the HBM than the MPI one.

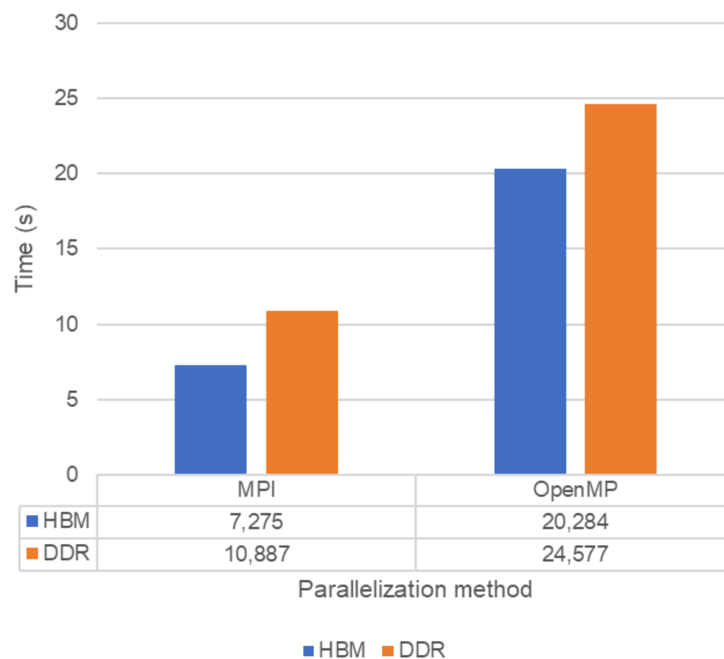

Fig. 8. Performance of QE Wave miniapp between HBM and DDR.

HBM vs DDR - Extended study

The improved version of IT4I HBM tool was used to perform full allocation-level memory pool study of QE Wave miniapp. The miniapp was compiled with Intel 2021.6 toolchain and executed with the *input_very_large* input file on a single socket (1 MPI process with 48 OpenMP threads).

The miniapp achieved maximum speedup of 1.35x with all of the data (approx. 1.25 GB visible to the tool) in the HBM memory pool. The analysis (see Fig. 9) showed three significant allocation groups (G1, G2, G3) in the miniapp, which lead to formation of clearly separated speedup clusters. The first allocation group (G1) with negligible memory footprint forms a cluster with ~10% performance improvement. The situation is more interesting when it comes to allocation groups G2 and G3 each of which encompasses about 45% of the application data. However, while moving the G2 to HBM can achieve nearly 15% speedup the G3 exhibits no performance improvement (for the same amount of HBM memory used). Finally, the combination of groups G1 and G2 can achieve over 90% of the maximum speedup at 50% of HBM usage, while the combination of G2 and G3 offers almost no speedup over G2 alone and doubles the amount of used HBM memory.

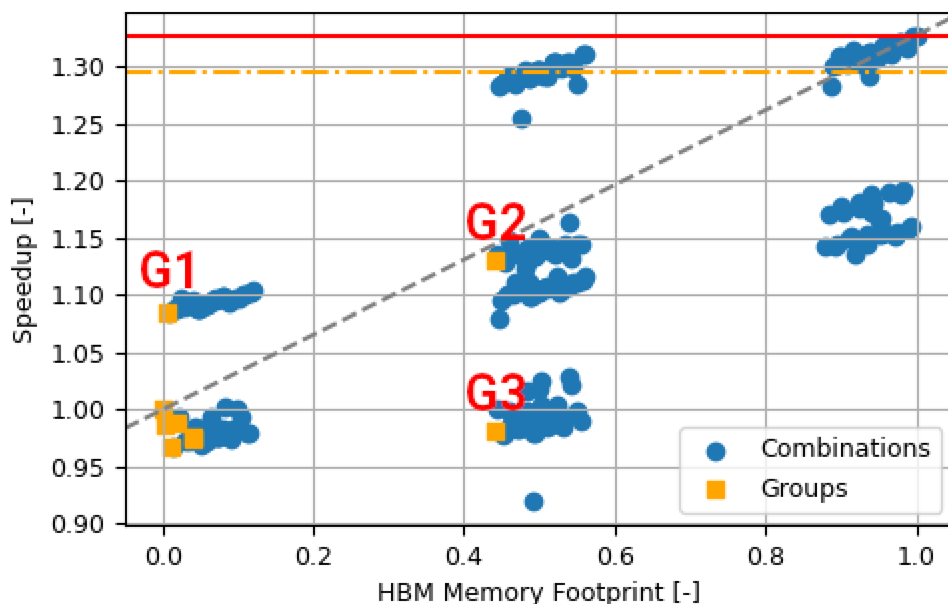

Fig. 9. Speed-up for combinations of individual significant allocations in QE Wave miniapp.

Vectorisation

In this section, we first explore the status of the vectorisation for each microarchitecture. The goal is to build a clear view of the vectorisation status with each compiler, their differences and optimisation possibilities, and the potential differences between microarchitectures. Second, we explore the SVE vectorisation on aarch64-based processors to identify if the SVE vectorisation is beneficial for this mini-app.

Current status

All tests were performed on an Eviden in-house cluster equipped with Intel Sapphire Rapids processors with HBM for x86 microarchitecture and Ampere Altra Max for aarch64 architecture, as described in Section 2.1. The following Table 4 shows the system and software configuration used for these runs.

	OS	Compiler	BLAS	FFT	MPI	Tools
x86	RHEL 8.8	Intel OneAPI 2023.2.0 GNU 13.2.0	Intel MKL 2023.2.0	Intel FFTW 2023.2.0	Open MPI 4.1.6 UCX 1.16.0	MAQAO 2.20.0
aarch64	RHEL 8.8	GNU 13.2.0 ACFL 24.04	ARMPL 24.04	ARMPL FFTW 24.04	Open MPI 4.1.6 UCX 1.16.0	MAQAO 2.20.0

Table 4. System and software stack used for vectorisation status experiments.

We specified the following options to the CMake build system of the miniapp:

```
-DCMAKE_C_COMPILER=<icx|gcc|clang>
-DCMAKE_Fortran_COMPILER=<ifx|gfortran|flang>
-DQE_ENABLE_OPENMP=1
-DQE_FFTW_VENDOR=<Intel_FFTW3|ARMPL>
```

To focus on analysing vectorisation metrics without considering the impact of other considerations, we chose to launch the execution with 1 MPI process and 1 OpenMP thread on a single core. Then we use the OneView mode of the MAQAO tool introduced in Section 2.2 to create a report on vectorisation of the run. To create such report, we launch the following command line:

```
maqao oneview --create-report=one -dbg=1 --replace \
              --envv_OMP_NUM_THREADS=1 --number-processes=1 \
              --uarch=<SAPPHIRE_RAPIDS|ARM_NEOVERSE_N1> \
              -xp=/path/to/logdir --show-program-output=on -- \
              /path/to/builddir/bin/miniapp2 < input_very_large
```

The following Table 5 shows the execution time we obtained for each compiler and microarchitecture used. We observe that on x86, OneAPI is way better than GNU (~20%), and ACFL is slightly slower than GNU on aarch64 (~5%). We can also observe that GNU is slightly faster on AARCH64 than on x86, but this comparison is not meaningful since both processors do

not have the same base frequency, number of ALUs, etc. We thus focus only on compiler comparison for the same microarchitecture, and compare GNU compiler between architecture for vectorisation, i.e. ensure that GNU compiler vectorisation metrics are the same for both microarchitectures.

	X86 (SPR HBM)	AARCH64 (Ampere Altra)
Intel OneAPI	45.55	
GNU	56.99	51.72
ACFL		54.80

Table 5. Execution time (in seconds) depending on compiler and microarchitecture used.

To better understand where the difference comes from between compilers, we present below the vectorisation statistics shown by MAQAO on x86 with Intel OneAPI and GNU. We can observe, as highlighted by red boxes, that the main differences between compilers comes from two locations: in `grid_set` routine in `fft_types.f90` file, and `impl_xy` routine in `fft_scatter.f90` file, where GNU vectorisation is very inefficient compared to OneAPI. These locations are the one to look at to search for performance optimisation, and will be discussed in the next section.

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
1261	miniapp2 - fft_scatter.f90:209-210	impl_xy_.DIR.OMP.PARALLEL.7	7.08	34.78	13.04
643	miniapp2 - fft_types.f90:1235-1251	grid_set	5.74	96.3	90.97
1264	miniapp2 - fft_scatter.f90:195-199	impl_xy_.DIR.OMP.PARALLEL.7	5.73	91.67	92.71
1016	miniapp2 - fft_scalar.FFTW3.f90:135-157 [...]	cft_1z	5.15	100	100
1253	miniapp2 - fft_scatter.f90:228-232	impl_xy_.DIR.OMP.PARALLEL.LOOP.25.split770	3.64	92.86	83.04
1313	miniapp2 - fft_scatter.f90:771-774	impl_yz_.DIR.OMP.PARALLEL.LOOP.25.split1177	3.12	92.86	83.04
401	miniapp2 - stick_base.f90:676-694	hpsort	2.88	0	10
1322	miniapp2 - fft_scatter.f90:746-749	impl_yz_.DIR.OMP.PARALLEL.7	2.21	91.67	92.71
753	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	1.99	100	25

X86 Intel OneAPI

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
172	miniapp2 - fft_types.f90:1233-1248	fft_type_allocate	27.18	10	11.88
516	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	5.64	100	37.5
609	miniapp2 - fft_scatter.f90:197-197	impl_xy.1._omp_fn.1	4.21	100	25
612	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	2.55	100	33.33
552	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	2.5	100	33.33
104	miniapp2 - stick_base.f90:676-697	hpsort	2.18	0	10.58
548	miniapp2 - fft_scatter.f90:747-747	impl_yz.0._omp_fn.1	1.78	100	25
265	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	1.61	100	25
607	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	1.29	28.57	12.95

X86 GNU

When looking at GNU compiler metrics between microarchitectures, as shown below, the main differences comes from the `impl_xy` routine in `fft_scatter.f90` file as previously mentioned,

and the `cft_1z` routine from `fft_scalar.FFTW3.f90` file. These locations should be checked to ensure that vectorisation is properly done on AARCH64.

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
172	miniapp2 - fft_types.f90:1233-1248	fft_type_allocate	27.18	10	11.88
516	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	5.64	100	37.5
609	miniapp2 - fft_scatter.f90:197-197	impl_xy.1._omp_fn.1	4.21	100	25
612	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	2.55	100	33.33
552	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	2.5	100	33.33
104	miniapp2 - stick_base.f90:676-697	hpsort	2.18	0	10.58
548	miniapp2 - fft_scatter.f90:747-747	impl_yz.0._omp_fn.1	1.78	100	25
265	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	1.61	100	25
607	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	1.29	28.57	12.95

X86 GNU

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
595	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	5.63	14.29	57.14
692	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	4.05	100	100
688	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	3.84	0	25
686	miniapp2 - fft_scatter.f90:195-197	impl_xy.1._omp_fn.1	1.84	100	100
632	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	1.8	100	100
118	miniapp2 - stick_base.f90:646-709 [...]	hpsort	1.45	0	36.76
627	miniapp2 - fft_scatter.f90:746-747	impl_yz.0._omp_fn.1	1.21	100	100

AARCH64 GNU

Finally, when comparing ACFL and GNU compilers on AARCH64, the main difference is located in the `cft_1z` routine from `fft_scalar.FFTW3.f90` file, where ACFL successfully vectorises it where GNU fails to do so. Since our target for this work is LLVM-based compilers, we argue that this location should be explored but with less priority than the two others to ensure proper vectorisation on AARCH64 LLVM-based compilers.

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
595	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	5.63	14.29	57.14
692	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	4.05	100	100
688	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	3.84	0	25
686	miniapp2 - fft_scatter.f90:195-197	impl_xy.1._omp_fn.1	1.84	100	100
632	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	1.8	100	100
118	miniapp2 - stick_base.f90:646-709 [...]	hpsort	1.45	0	36.76
627	miniapp2 - fft_scatter.f90:746-747	impl_yz.0._omp_fn.1	1.21	100	100

AARCH64 GNU

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
247	miniapp2 - fft_types.f90:1233-1248	fft_types_grid_set	27.71	90.57	75.24
545	miniapp2 - fft_scatter.f90:230-232	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L219_3_	6.09	66.67	83.33
500	miniapp2 - fft_scalar.FFTW3.f90:135-135	fft_scalar_fftw3_cft_1z	5.81	100	100
616	miniapp2 - fft_scatter.f90:772-774	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L760_13_	2.34	66.67	83.33
542	miniapp2 - fft_scatter.f90:197-199	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L185_2_	2.28	50	75
612	miniapp2 - fft_scatter.f90:747-749	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L727_12_	2.04	50	75

AARCH64 ACFL

To conclude this section, the vectorisation can be improved for GNU compiler, and we identified possible locations for optimisation:

- `grid_set` routine in `fft_types.f90`;
- `impl_xy` routine in `fft_scatter.f90`;
- `cft_1z` routine from `fft_scalar.FFTW3.f90`.

Since these files are located in the fftxlib7 library used by both the mini-app and Quantum Espresso, these optimisations could also easily benefit Quantum Espresso.

ARM NEON vs SVE

For these experiments we used the NVIDIA Grace CPU (72 cores) from the E4 cluster introduced in Section 2.1. The following Table 6 shows the system and software configuration used for these runs.

OS	Compiler	BLAS	FFT	MPI	Tools
Ubuntu 22.04 LTS	GNU 13.2.0 ACFL 24.04	ARMPL 24.04	ARMPL FFTW 24.04	Open MPI 4.1.6 UCX 1.16.0	MAQAO 2.20.7

Table 6. System and software stack used for vectorisation status experiments.

The build options given to the CMake build system of the miniapp are the same as the previous section. For these runs, we edited the Makefile to build with specific compiler options either for NEON or SVE specifically, as follows:

- SVE: `-mtune=neoverse-v2 \`
  `-march=armv9.1-a+simd`**`+sve2`** `\`
  `-msve-vector-bits=128`
- NEON: `-mtune=neoverse-v2 \`
  `-march=armv9.1-a+simd`**`+nosve`**

The command line used to obtain the OneView from MAQAO is also similar to the one shown in the previous section. However, the microarchitecture parameter `uarch` has been changed to `ARM_NEOVERSE_V1` to identify SVE instructions. At the time the experiments were made, MAQAO did not support ARM Neoverse V2 profiling yet. This is a work in progress in the context of the EMOPASS European Project. This implies that some metrics, especially the vector length use, might be incorrect as ARM Neoverse V1 exhibits SVE vectors of 256 bits while the Nvidia Grace CPU (Neoverse V2) has SVE vectors of 128 bits. These results may need to be consolidated when the support of ARM Neoverse V2 microarchitecture is available in MAQAO, but are still meaningful to ensure that compilers properly generate SVE instructions and investigate performance differences and compiler choices between using NEON or SVE.

The following Table 7 shows the performance results we obtained. We observe that the miniapp is either at the same performance or slightly slower when using SVE compared to NEON.

	NEON (NVIDIA Grace)	SVE (NVIDIA Grace)

GNU	23.46	23.89
ACFL	24.75	24.78

Table 7. Execution time (in seconds) depending on compiler and ARM vectorisation used.

When looking at MAQAO vectorisation reports shown below between the two builds for ACFL, we observe that only a single location (`cft_1z` in `fft_scalar.FFTW3.f90` file) has different vectorisation statistics between the two builds, suggesting that SVE instructions are only used in this location. However, the biggest contributor in terms of execution time (`grid_set` in `fft_types.f90` file) has the same vectorisation metrics, suggesting that NEON instructions are used in both builds.

Loop id	Source Location	Source Function	Exclusive coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
236	miniapp2 - fft_types.f90:1249-1262	fft_types_grid_set	21.47	97.14	47.5
481	miniapp2 - fft_scalar.FFTW3.f90:135-135	fft_scalar_fftw3_cft_1z	5.13	100	75
517	miniapp2 - fft_scatter.f90:198-200	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L186_2_	3.78	66.67	41.67
515	miniapp2 - fft_scatter.f90:214-215	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L186_2_	3.43	0	25
571	miniapp2 - fft_scatter.f90:762-764	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L742_12_	2.87	66.67	41.67
519	miniapp2 - fft_scatter.f90:245-247	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L234_3_	2.67	66.67	41.67
123	miniapp2 - stick_base.f90:676-694	stick_base_hpsort	2.52	0	19.64
574	miniapp2 - fft_scatter.f90:787-789	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L775_13_	2.00	66.67	41.67
287	miniapp2 - fft_helper_subroutines.f90:1035-1036	fft_helper_subroutines_fftx_psi2c_k	1.05	100	50
572	miniapp2 - fft_scatter.f90:747-748	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L742_12_	1.01	0	12.5

ACFL NEON

Loop id	Source Location	Source Function	Exclusive coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
250	miniapp2 - fft_types.f90:1249-1262	fft_types_grid_set	21.55	97.14	47.5
497	miniapp2 - fft_scalar.FFTW3.f90:135-135	fft_scalar_fftw3_cft_1z	5.01	92.31	94.23
534	miniapp2 - fft_scatter.f90:198-200	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L186_2_	3.96	66.67	41.67
532	miniapp2 - fft_scatter.f90:214-215	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L186_2_	3.17	0	25
585	miniapp2 - fft_scatter.f90:762-764	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L742_12_	2.99	66.67	41.67
536	miniapp2 - fft_scatter.f90:245-247	__nv_fft_scatter_fft_scatter_xy_impl_xy_F1L234_3_	2.58	66.67	41.67
136	miniapp2 - stick_base.f90:676-694	stick_base_hpsort	2.54	0	19.64
588	miniapp2 - fft_scatter.f90:787-789	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L775_13_	2.18	66.67	41.67
303	miniapp2 - fft_helper_subroutines.f90:1035-1036	fft_helper_subroutines_fftx_psi2c_k	1.05	100	50
586	miniapp2 - fft_scatter.f90:747-748	__nv_fft_scatter_fft_scatter_yz_impl_yz_F1L742_12_	1.01	0	12.5

ACFL SVE

We decided to explore further and we present below the assembly code for the two locations. For the `cft_1z` routine, we can clearly observe that the NEON build (left) uses V registers that are associated with the NEON ISA, while the SVE build (right) uses Z registers that are associated with the SVE ISA. We can thus ensure that the GNU compiler is able to generate SVE instructions for this miniapp.

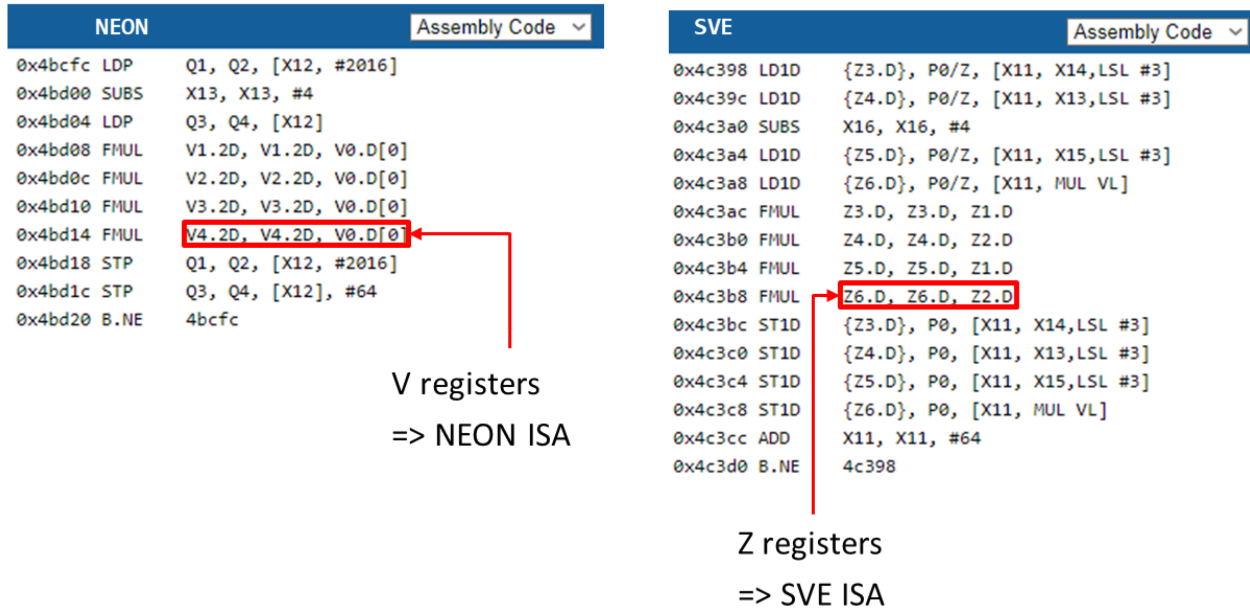

Fig. 10. Assembly code of `cft_1z` routine in `fft_scalar.FFTW3.f90: 135-135.`

For the `grid_set` routine, we can clearly see that both builds use V registers that are associated with the NEON ISA. We thus expect that the compiler identified through its performance models that this section would have better performance if vectorised with NEON instructions.

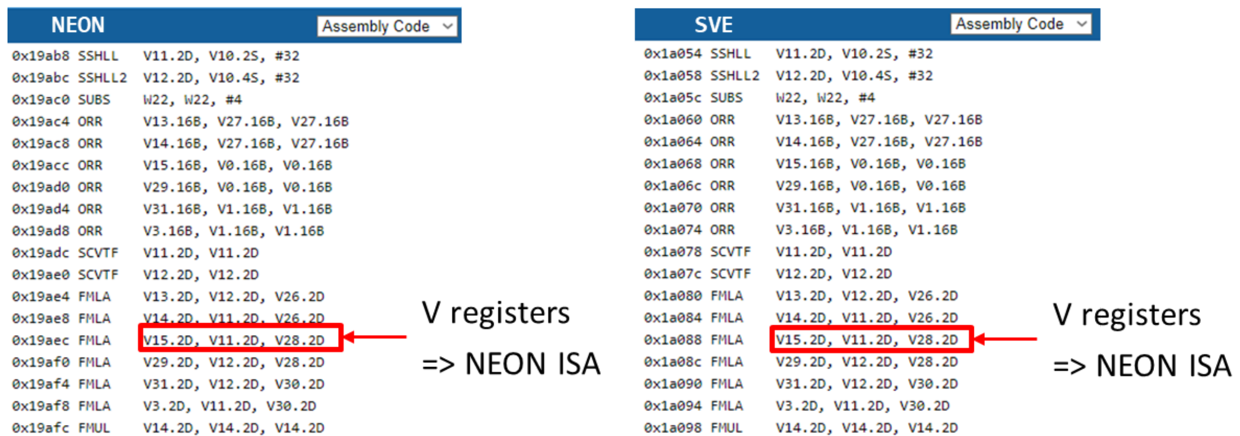

Fig. 11. Assembly code of `grid_set` routine in `fft_types.f90: 1249-1262.`

We can look at why the compiler took this decision by adding to the Makefile the following options (for LLVM-based compilers): `-Rpass=loop -Rpass-analysis=loop`. The output for both builds and locations for the ACFL compiler is shown below. Vectorisation is not considered

by the compiler for the `grid_set` routine, telling us that the cost model of the compiler finds that NEON would be better in performance compared to SVE, which is the case in our experiments.

fft_scalar.FFTW3.f90:135

- NEON

```
/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_scalar.FFTW3.f90:135: remark: vectorized
loop (vectorization width: 2, interleaved count: 2) [-Rpass=loop-vectorize]
  135 |         cout( 1 : ldz * nsl ) = cout( 1 : ldz * nsl ) * tscale
```

- SVE

```
/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_scalar.FFTW3.f90:135: remark:
Instruction with invalid costs prevented vectorization at VF=(vscale x 1): load
[-Rpass-analysis=loop-vectorize]
  135 |         cout( 1 : ldz * nsl ) = cout( 1 : ldz * nsl ) * tscale

/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_scalar.FFTW3.f90:135: remark:
Instruction with invalid costs prevented vectorization at VF=(vscale x 1):
store [-Rpass-analysis=loop-vectorize]
/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_scalar.FFTW3.f90:135: remark: vectorized
loop (vectorization width: vscale x 2, interleaved count: 2) [-Rpass=loop-
vectorize]
```

fft_types.f90:1249-1262

- NEON

```
/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_types.f90:1247: remark: the cost-model
indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]
 1247 |             DO i = -nr1, nr1

/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_types.f90:1247: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

- SVE

```
/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_types.f90:1247: remark: the cost-model
indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]
 1247 |             DO i = -nr1, nr1

/home/msergent/workspace/projects/2024/max/mini-
apps/Quantum_Espresso/fftxlib7/src/fft_types.f90:1247: remark: vectorized loop
(vectorization width: 4, interleaved count: 1) [-Rpass=loop-vectorize]
```

Optimisation

To improve the performance of the Quantum Espresso wave miniapp, we looked at the two parts that were most identified in the vectorisation study: `grid_set` in `fft_types.f90`, and `impl_xy` in `fft_scatter.f90`.

Vectorisation on grid_set

In this routine, the targeted loop is shown below. The red square shows the part of this code that is responsible for bad vectorisation from compilers. This `IF` construct cannot be vectorised by compilers due to the presence of the `MAX` operator: compilers do not know the reduction operator for this construct. Consequently, the compiler cannot vectorise the whole inner loop, resulting in bad vectorisation for this loop nest.

```
1227       DO k = -nr3, nr3
1228         !
1229         ! ... me_image = processor number, starting from 0
1230         !
1231         IF( MOD( k + nr3, dfft%nproc ) == dfft%mype ) THEN
1232           DO j = -nr2, nr2
1233             DO i = -nr1, nr1
1234
1235               g( 1 ) = DBLE(i)*bg(1,1) + DBLE(j)*bg(1,2) + DBLE(k)*bg(1,3)
1236               g( 2 ) = DBLE(i)*bg(2,1) + DBLE(j)*bg(2,2) + DBLE(k)*bg(2,3)
1237               g( 3 ) = DBLE(i)*bg(3,1) + DBLE(j)*bg(3,2) + DBLE(k)*bg(3,3)
1238
1239 ! ...          calculate modulus
1240
1241               gsq =  g( 1 )**2 + g( 2 )**2 + g( 3 )**2
1242
1243               IF( gsq < gcut ) THEN
1244
1245 ! ...            calculate maximum index
1246                 nb(1) = MAX( nb(1), ABS( i ) )
1247                 nb(2) = MAX( nb(2), ABS( j ) )
1248                 nb(3) = MAX( nb(3), ABS( k ) )
1249               END IF
1250
1251             END DO
1252           END DO
1253         END IF
1254       END DO
```

Fig 12. grid_set code before optimisation.

We show the optimisation applied for this loop nest below. First, we changed computations of intermediate variables, so that they are computed at the associated loop nest level to reduce the number of duplicated computations of the same variable. Second, we changed the `IF(…) MAX` construct with equivalent `MAX(…, MERGE(…))` construct. With the `MERGE` construct, the compiler is able to identify how to vectorise the inner loop. We checked the numerical correctness to ensure that there is no change in computed output values for this loop nest.

```
1228        max_nr = MAX(nr1, MAX(nr2, nr3))
1229
1230        DO k = -nr3, nr3
1231          !
1232          ! ... me_image = processor number, starting from 0
1233          !
1234          kg(1) = DBLE(k)*bg(1,3)
1235          kg(2) = DBLE(k)*bg(2,3)
1236          kg(3) = DBLE(k)*bg(3,3)
1237          absk = ABS( k )
1238
1239          IF( MOD( k + nr3, dfft%nproc ) == dfft%mype ) THEN
1240            DO j = -nr2, nr2
1241
1242              jg(1) = kg(1) + DBLE(j)*bg(1,2)
1243              jg(2) = kg(2) + DBLE(j)*bg(2,2)
1244              jg(3) = kg(3) + DBLE(j)*bg(3,2)
1245              absj = ABS( j )
1246
1247              DO i = -nr1, nr1
1248
1249                g( 1 ) = jg(1) + DBLE(i)*bg(1,1)
1250                g( 2 ) = jg(2) + DBLE(i)*bg(2,1)
1251                g( 3 ) = jg(3) + DBLE(i)*bg(3,1)
1252
1253 ! ...          calculate modulus
1254
1255                gsq =  g ( 1 )*g( 1 ) + g( 2 )*g( 2 ) + g( 3 )*g( 3 )
1256
1257                gsq_gcut = gsq < gcut
1258
1259 ! ...          !   calculate maximum index
1260                nb(1) = MAX( nb(1), ABS( i ) + MERGE( 0, -max_nr, gsq_gcut))
1261                nb(2) = MAX( nb(2), absj + MERGE( 0, -max_nr, gsq_gcut))
1262                nb(3) = MAX( nb(3), absk + MERGE( 0, -max_nr, gsq_gcut))
1263              END DO
1264            END DO
1265          END IF
1266        END DO
```

Fig 13. grid_set code after proposed optimisation.

We present in the following MAQAO reports with this optimisation, that shows that vectorisation is now properly done by both OneAPI and GNU compilers on x86 for this routine.

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
1277	miniapp2 - fft_scatter.f90:209-210	impl_xy_.DIR.OMP.PARALLEL.7	7.53	34.78	13.04
1280	miniapp2 - fft_scatter.f90:195-199	impl_xy_.DIR.OMP.PARALLEL.7	6.33	91.67	92.71
659	miniapp2 - fft_types.f90:1251-1265	grid_set	5.69	99.53	98.86
1032	miniapp2 - fft_scalar.FFTW3.f90:135-157 [...]	cft_1z	5.44	100	100
1269	miniapp2 - fft_scatter.f90:228-232	impl_xy_.DIR.OMP.PARALLEL.LOOP.25.split770	3.78	92.86	83.04
1329	miniapp2 - fft_scatter.f90:771-774	impl_yz_.DIR.OMP.PARALLEL.LOOP.25.split1177	3.41	92.86	83.04
414	miniapp2 - stick_base.f90:676-694	hpsort	3	0	10
1338	miniapp2 - fft_scatter.f90:746-749	impl_yz_.DIR.OMP.PARALLEL.7	2.39	91.67	92.71
769	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	2	100	25

X86 Intel OneAPI

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
166	miniapp2 - fft_types.f90:1251-1264	grid_set.isra.0	9.46	100	47.46
517	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	7.12	100	37.5
610	miniapp2 - fft_scatter.f90:197-197	impl_xy.1._omp_fn.1	5.28	100	25
553	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	3.14	100	33.33
613	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	3.14	100	33.33
106	miniapp2 - stick_base.f90:676-697	hpsort	2.81	0	10.58
549	miniapp2 - fft_scatter.f90:747-747	impl_yz.0._omp_fn.1	2.27	100	25
266	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	1.98	100	25
608	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	1.72	28.57	12.95

X86 GNU

We show in the following table the performance results we obtained with this optimisation. The left value (red) is the execution time with the original code, while the right value (green) is the one obtained with the `grid_set` optimisation. Depending on compilers and microarchitecture used, we observe a reduced execution time from 6.3 to 21% on this mini-app with the proposed optimisation.

	X86 (SPR HBM)	AARCH64 (Ampere Altra)
Intel OneAPI	45.55 => **42.70 (-6.3%)**	
GNU	56.99 => **45.16 (-21%)**	51.72 => **47.24 (-8.7%)**
ACFL		54.80 => **49.00 (-10.6%)**

Table 8. Execution time (in seconds) depending on compiler and microarchitecture used with `grid_set` optimisation.

Vectorisation on impl_xy

In this routine, the targeted loop is shown below. The red square shows the part of this code that is responsible for bad vectorisation from compilers. This `IF` construct cannot be vectorised by compilers since they do not know the reduction operator for this construct. This is probably due to the pointer arithmetic to determine the location to write in the `f_aux` table. Consequently, the compiler cannot vectorise the whole inner loop, resulting in bad vectorisation for this loop nest.

```
203       !
204       ! clean extra array elements in each stick
205       !
206 !$omp do
207       DO k = 1, my_nr2p*desc%my_nr3p
208          DO i1 = 1, desc%nr1x
209             IF(iplx(i1)==0) f_aux(desc%nr1x*(k-1)+i1) = (0.0_DP, 0.0_DP)
210          ENDDO
211       ENDDO
212 !$omp end do nowait
```

Fig 14. Impl_xy code before optimisation

We show the optimisation applied for this loop nest below. First, we changed computations of intermediate variables, so that they are computed at the associated loop nest level to reduce the number of duplicated computations of the same variable. To ensure proper variable isolation between OpenMP threads, we added the `private(f_aux_i, f_aux_k)` clause to the OpenMP construct. Second, we changed the `IF(…)` construct with an equivalent `MERGE(…)` construct. With the `MERGE` construct, the compiler is able to identify how to vectorise the inner loop. We checked the numerical correctness to ensure that there is no change in computed output values for this loop nest.

```
204       !
205       ! clean extra array elements in each stick
206       !
207 !$omp do private(f_aux_i, f_aux_k)
208       DO k = 1, my_nr2p*desc%my_nr3p
209          nr1x = desc%nr1x
210          f_aux_k = nr1x*(k-1)
211          DO i1 = 1, nr1x
212             f_aux_i = f_aux_k+i1
213             f_aux(f_aux_i) = MERGE((0.0_DP, 0.0_DP), f_aux(f_aux_i), iplx(i1) == 0)
214          ENDDO
215       ENDDO
```

Fig 15. Impl_xy code after proposed optimisation

We show in the following table the performance results we obtained with this optimisation. The left value (red) is the execution time with the `grid_set` optimisation , while the right value (green) is the one obtained with both `grid_set` and `impl_xy` optimisations. Depending on compilers and microarchitecture used, we observe a reduced execution time from 2.3 to 4% on this mini-app with the `impl_xy` optimisation proposed. We do not have results for ACFL due to a preliminary version of the optimisation that was not working on LLVM-based compilers. This version of the code has been reworked thanks to the input of partners of EMOPASS European Project, but Ampere Altra CPU was not available anymore for testing. However, results for Intel OneAPI and GNU for X86 / AARCH64 are enough to show that this optimisation is beneficial.

	X86 (SPR HBM)	AARCH64 (Ampere Altra)
Intel OneAPI	42.70 => **41.11 (-3.7%)**	
GNU	45.16 => **44.11 (-2.3%)**	47.24 => **45.35 (-4%)**

Table 9. Execution time (in seconds) depending on compiler and microarchitecture used with `impl_xy` optimisation.

MAQAO reports shows that for OneAPI the `impl_xy` part is less vectorised than before, but the time spent in the routine dropped from 7.53% to 1.39% of the total execution time. For the GNU compiler, this section is not even identified at a hotspot anymore by MAQAO.

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
666	miniapp2 - fft_types.f90:1249-1263	grid_set	7.04	99.58	98.98
1291	miniapp2 - fft_scatter.f90:196-200	impl_xy_.DIR.OMP.PARALLEL.7.split841	6.52	91.67	92.71
1043	miniapp2 - fft_scalar.FFTW3.f90:135-157 [...]	cft_1z	5.62	100	100
1280	miniapp2 - fft_scatter.f90:241-245	impl_xy_.DIR.OMP.PARALLEL.LOOP.25.split780	4.04	92.86	83.04
1340	miniapp2 - fft_scatter.f90:784-787	impl_yz_.DIR.OMP.PARALLEL.LOOP.25.split1177	3.53	92.86	83.04
421	miniapp2 - stick_base.f90:676-694	hpsort	3.14	0	10
1349	miniapp2 - fft_scatter.f90:759-762	impl_yz_.DIR.OMP.PARALLEL.7	2.51	91.67	92.71
776	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	2.15	100	25
1288	miniapp2 - fft_scatter.f90:213-214	impl_xy_.DIR.OMP.PARALLEL.7.split841	1.39	0	9.38

X86 Intel OneAPI

Loop id	Source Location	Source Function	Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
166	miniapp2 - fft_types.f90:1251-1264	grid_set.isra.0	9.46	100	47.46
517	miniapp2 - fft_scalar.FFTW3.f90:135-135	cft_1z	7.12	100	37.5
610	miniapp2 - fft_scatter.f90:197-197	impl_xy.1._omp_fn.1	5.28	100	25
553	miniapp2 - fft_scatter.f90:772-772	impl_yz.0._omp_fn.2	3.14	100	33.33
613	miniapp2 - fft_scatter.f90:230-230	impl_xy.1._omp_fn.2	3.14	100	33.33
106	miniapp2 - stick_base.f90:676-697	hpsort	2.81	0	10.58
549	miniapp2 - fft_scatter.f90:747-747	impl_yz.0._omp_fn.1	2.27	100	25
266	miniapp2 - fft_helper_subroutines.f90:559-559	fftx_c2psi_k	1.98	100	25
608	miniapp2 - fft_scatter.f90:208-209	impl_xy.1._omp_fn.1	1.72	28.57	12.95

X86 GNU

4.3.2. YAMBO

Miniapp - HBM study

The results presented here have been obtained from the commit 9997bbca (dated from 05/03/2024) of the mini-app repository. This mini-app was extracted from the version 5.3-beta of the Yambo application. The mini-app is a Fortran only full OpenMP application. We always use a number of threads equal to the number of physical CPU cores available.

In all testing, we used the following input arguments and test case : `-ng 3000 -nb 200 -nq 10 -nw 2 -f dataset_1.dat` where `-ng` is the number of plane-waves to represent the X matrix, -nb is the number of number of bands included in the sum-over-states, -nq the number of q-points where X is computed and -nw the number of frequencies for which X is computed.

The HBM study has been performed on an dual-socket Intel 9480 node with 56 cores per socket. HBM is configured as SNC4 with either Flat mode (16GB on all 8 NUMA nodes) or Cache mode (128GB of L4) as described in section 3.3. We only use the first socket so `OMP_NUM_THREADS` is set to 56.

We have tested a GNU and oneAPI toolchain as shown in the following Table 10:

Compiler	BLAS
GNU 14.2.0	MKL 2025.0
oneAPI 2025.0	MKL 2025.0

Table 10. Toolchains used for HBM study.

In the following figure (Fig. 16) we compare a GNU toolchain using the gfortran compiler to a oneAPI toolchain using the ifx compiler. Both use the same MKL version. We use the following compilation flags to take advantage of the AVX512 instructions set:

- `-O2 -funroll-loops -march=native -mtune=native`, for GNU toolchain;
- `-O2 -xHost`, for oneAPI toolchain.

Without using HBM the GNU version is about 7% faster than oneAPI with the previous standard set of optimizations and about 11% faster when enabling the most aggressive optimizations using `-Ofast` instead of `-O2`.

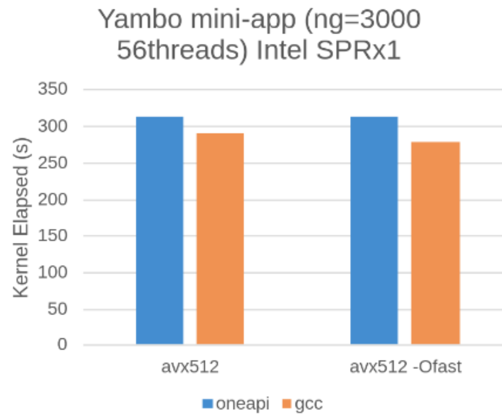

Fig. 16. Elapsed time, GNU vs. oneAPI, no HBM used.

The HBM study has been performed with the GNU compiler. With Flat mode, binding on HBM nodes has been manually performed with `hwloc-bind --strict --membind N` where `N` is the node number closest to the process. The Yambo miniapp requires about 10GB of memory to run (for the 3000 ng test case). Since we have 128GB of HBM we can fit everything on the HBM and we don't expect Cache mode to behave better than Flat mode.

On the following figure (Fig. 17) we observe a 57% performance gain when running on HBM memory with Flat mode. In Cache mode, enabling Transparent Huge Pages (THP) improves performance by about 15%, enabling us to achieve the same performance improvement as Flat mode. THP has no performance impact in Flat mode for the Yambo mini-app. From our testing we can see that the Yambo mini-app is very memory bound despite not being very sensitive to vectorization (discussed in next sections). Improving vectorization could further increase sensitivity to memory bandwidth and increase performance gain from HBM.

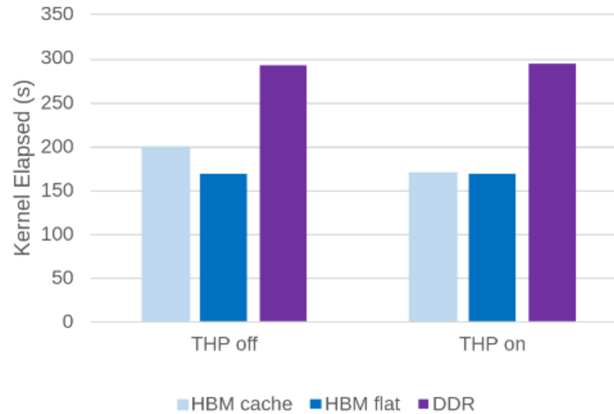

Fig. 17. HBM elapsed time study with GNU toolchain.

Miniapp - Extended HBM study for individual allocations using IT4I HBM tool

For this extended study we used the miniapp version from commit 3c4ff63a (06/05/2025). It was compiled with Intel 2021.6 toolchain and the test case was configured as: `-X_ng 5000 -X_nq 3 -nbnd 25 -f tests/dataset_1.dat`.

The detailed analysis of individual allocations in the miniapp (Fig. 18) showed that there are two allocations with significant performance impact. The first of these two allocations allows to improve performance by 30%, while using up only about 380MBs of HBM memory pool. However, the largest performance uplift (about 60%) can be achieved by moving the largest allocation (about 27GBs or 95% of the total memory used for the 5000 ng test case) to the HBM memory pool. Using HBM for all four significant allocations allows to achieve up to 80% performance uplift compared to using only DDR memory.
The results of detailed analysis somewhat differ from process level analysis due to slight changes in both machine and test case. While the platform used was the same the CPUs are the 48 core variant (single socket was used for testing).

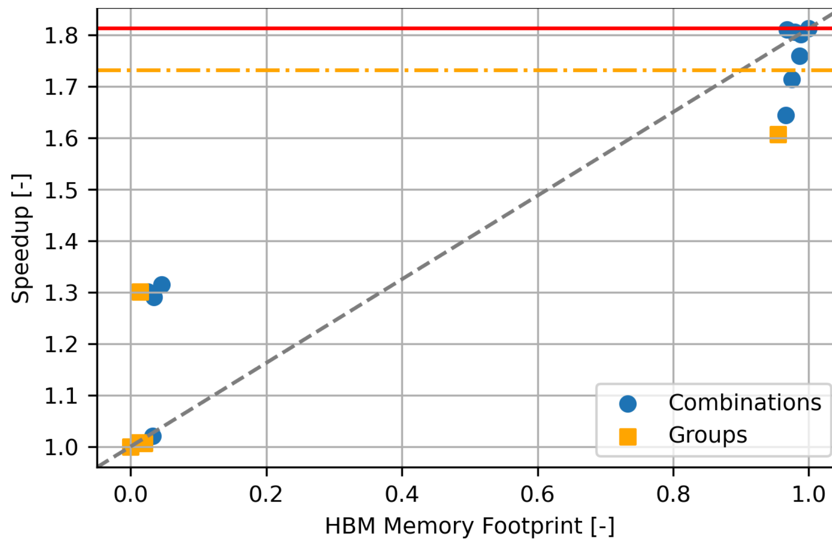

Fig. 18. Speed-up for combinations of individual significant allocations in Yambo miniapp.

Miniapp - SVE2 Vectorization

We use the same miniapp version (9997bbca) and test case (`-ng 3000 -nb 200`) than presented in the previous sections.

The vectorization study has been performed on a quad-socket Nvidia Grace node with 72 cores per socket. The CPU is a ARM Neoverse-V2 architecture, with 4x128b SVE2. We only use the first socket so `OMP_NUM_THREADS` is set to 72.

The ARM toolchains tested are shown in the following table:

Compiler	BLAS
GNU 14.2.0	ARMPL 24.10
ACFL 24.10	ARMPL 24.10
NVHPC 25.1	ARMPL 24.10
MAQAO 2.21.3	

Table 11. Toolchains and tools used.

We also tested a LLVM 20.1 toolchain but we were not able to compile the mini-app due to an error during the compilation link. We used the following compilation flags to compare SVE2 vectorization and NEON vectorization:

- `-O2 -funroll-loops -mcpu=neoverse-v2 -msve-vector-bits=128`, for SVE2 vectorization;
- `-O2 -funroll-loops -mcpu=neoverse-v2+nosve`, for NEON vectorization.

As we can see on the following figure (Fig. 19), the best result is obtained with the NVHPC compiler. With -O2 NVHPC is about 43% and 44% faster than respectively ACFL and GNU. Enabling the most aggressive optimization flag with -Ofast improves GNU performance by about 42%, closely matching NVHPC performance. This has no impact on ACFL performance however. Investigating the different optimizations enabled with -Ofast we were able to deduce that it comes from the -fstack-arrays optimization, which puts all unknown size and array temporaries onto stack memory. Finally, there is no noticeable performance benefit from activating SVE2 vectorization compared to NEON vectorization. We can also note that NVHPC isn't able to produce a NEON only binary. The compiler would crash during compilation.

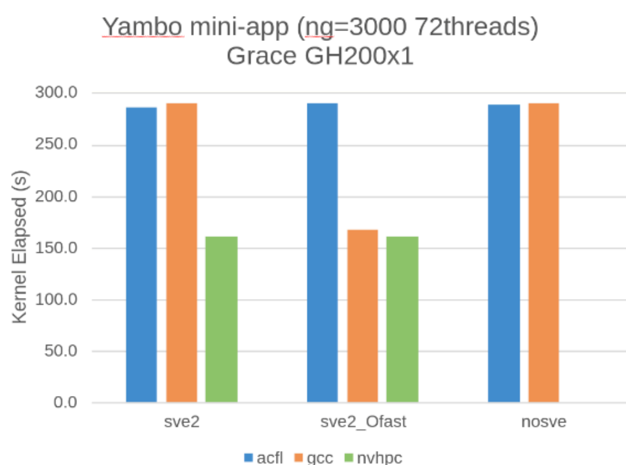

Fig. 19. Elapsed time with several ARM toolchains and optimization flags.

To get a good idea of the parallel efficiency of the Yambo mini-app, we performed a single node strong scalability analysis on up to 4 sockets (288 threads). The overall parallel efficiency is quite poor across all toolchains with 53% efficiency obtained on 9 cores with the NVHPC toolchain and 15% on a full socket. When we use more than one socket, we even observe a performance degradation. We could not clearly identify the source of this poor scalability but we postulate that NUMA effect may be one of the causes.

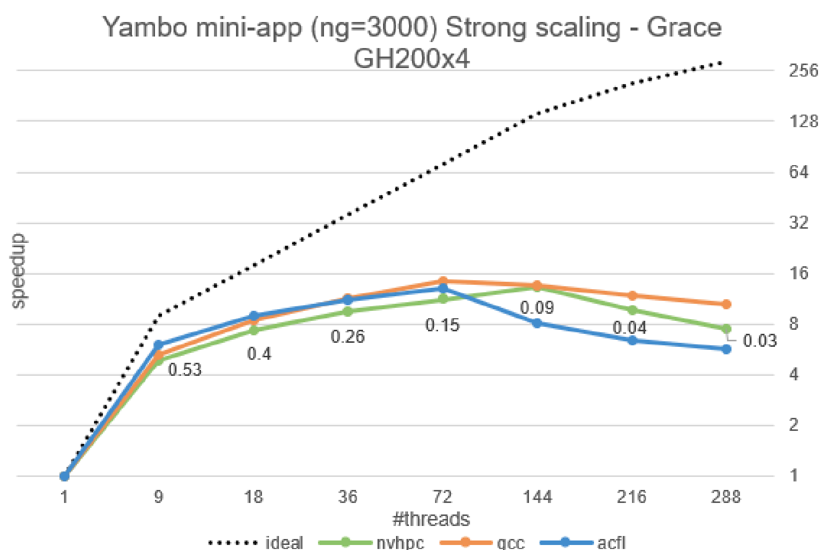

Fig. 19. Strong scalability, efficiency for the NVHPC toolchain is shown on the plot for each data point.

We use MAQAO performance summary report to get an overview of current bottlenecks. We found out that a significant amount of threads are idle with 35% observed idle threads on average. When threads aren't idle they spend most of their time in the BLAS1 caxpy , with 55% of the overall runtime. As a consequence, the Yambo mini-app has a fairly flat profile and less than 20% of the time is spent in loops, making it harder to find optimizations opportunities.

In the following figure (Fig. 20) we have summarized performance of different toolchains with SVE2 vectorization compared to with vectorization entirely disabled. We used the following compilation flags to disable vectorization:

- `-O2 -funroll-loops -fno-vectorize -fno-tree-vectorize -fno-slp-vectorize`, for GNU;
- `-O2 -funroll-loops -fno-tree-vectorize -fno-tree-loop-vectorize -fno-tree-slp-vectorize` for ACFL;
- `-O2 -Munroll -mcpu=neoverse-v2 -Mnovect`, for NVHPC.

We observe no noticeable performance degradation when vectorization is entirely disabled. We can conclude that the Yambo mini-app does not benefit from vectorization (excluding any vectorization used in external library routines such as caxpy).

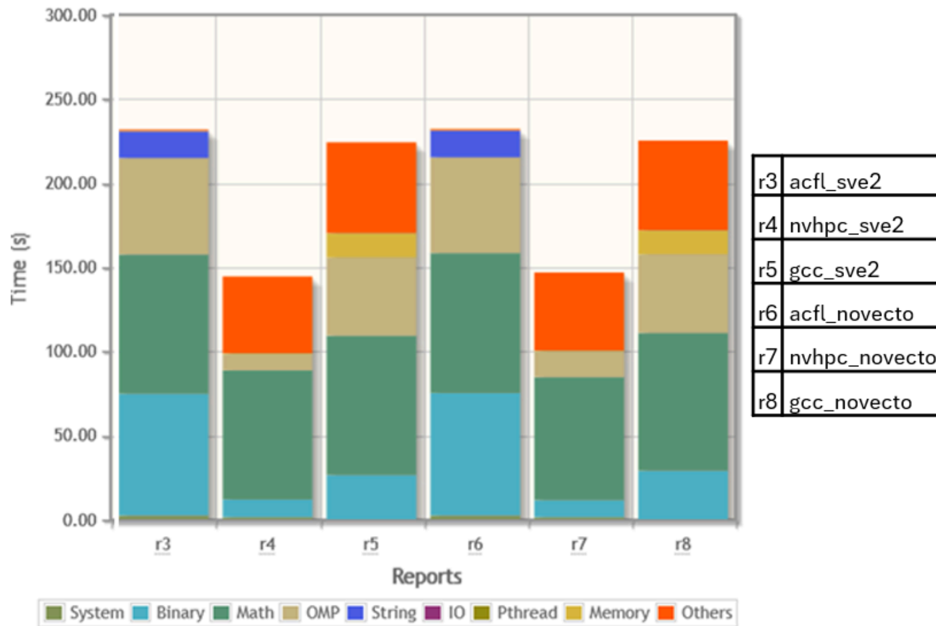

Fig. 20. MAQAO application summarized performance: SVE2 vs. No Vectorization.

4.3.3. BigDFT

Full application - HBM study

The study has been done on the full BigDFT application using the commit 0e33f8503 from the release 1.9.5 (2024-06-17). We use an Intel toolchain with the ifort 2023.2 compiler and version 4.1.5.3 of OpenMPI. While the Fortran compiler is quite old, we could not get a working ifx binary using more recent oneAPI toolchain.

We use the following compilation flags for the Intel toolchain:

- `-O2 -xcommon-avx512 -qopt-zmm-usage=high`, for the optimization options;
- `-lmkl_scalapack_lp64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lmkl_blacs_openmpi_lp64`, for the Intel MKL flags.

Two variations of the H2O-32 (CPU) test case are used: PBE and PBE0. Both have very different profiles, as we will see, and we will mostly discuss PBE0. The test case has been adjusted with the parameter itermax increased to 50.

BigDFT is a hybrid MPI and OpenMP application written in Fortran. We have tested different combinations of the number of OpenMP threads per MPI rank. We found the optimal solution was 16 MPI ranks per node, with 2 MPI ranks per NUMA and 7 OpenMP threads per rank. Turbo and Transparent Huge Pages are deactivated for all runs.

The HBM study has been performed on an dual-socket Intel 9480 node with 56 cores per socket. HBM is configured as SNC4 with either Flat mode (16GB on all 8 NUMA nodes) or Cache mode (128GB of L4) as described in section 3.3.

We first present results obtained with the PBE0 test case as the PBE test case has a different profile and is too short to do any meaningful analysis.

H20-32 PBE0 test case

Before evaluating HBM performance, we evaluated the memory sensitivity of BigDFT by doing a Scatter / Compact study on several nodes. Our normal run uses 32 ranks with 7 OpenMP threads per rank across two nodes (16 ranks per node) for a total of 224 active threads, one for each physical core available (2 nodes, 2x56cores per node). Scatter and Compact use the same number of active threads, but on twice as many nodes (8 ranks per node). In Compact the first

socket of each node is used fully and the second socket is left unused. Since we have twice as many nodes, the communication bandwidth per thread is virtually doubled. In Scatter we use the two sockets but one every two cores is used. In this configuration the memory bandwidth available per thread is virtually doubled (Table 12).

Affinity	Ranks	Nodes	Used NUMA	Ranks per NUMA	Mem. BW	Comm. BW
normal	32	2	2x8	2	α	β
compact	32	4	4x4	2	1xα	2xβ
scatter	32	4	4x8	1	2xα	2xβ

Table 12. Scatter / Compact study test summary.

The Compact run has a less than 2% performance gain while the Scatter improves performance by around 11% (see Fig. 21). We can conclude that BigDFT is not very sensitive to memory bandwidth, as we will see in our HBM study. By manipulating the CPU clock frequency we were also able to observe that BigDFT is quite sensitive to it, which indicates that BigDFT is more compute bound than memory bound.

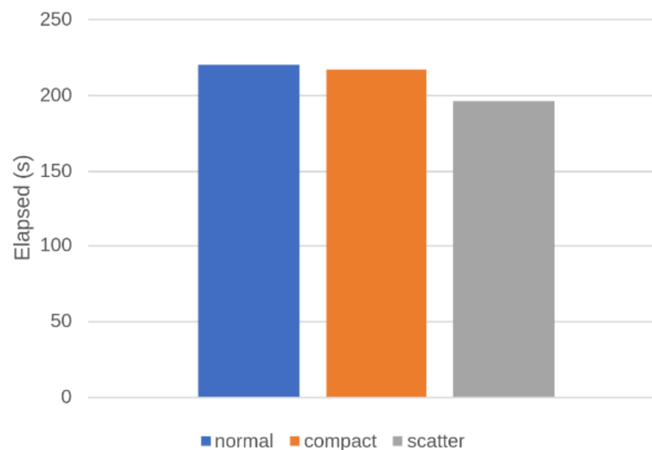

Fig. 21. Scatter / Compact study test results.

With Flat mode, binding on HBM nodes has been manually performed with `hwloc-bind --strict --membind N` where `N` is the node number closest to each MPI rank. We need about 24GB of memory in total so we can allocate all data on HBM nodes and we don't expect Cache mode to behave better than Flat mode.

Below is shown the HBM result for Flat and Cache mode (Fig. 22). We obtained a 14% gain in Flat mode and a 13% gain in Cache mode. It's a fairly low performance gain, but it is not surprising considering the previously discussed memory bandwidth sensitivity study.

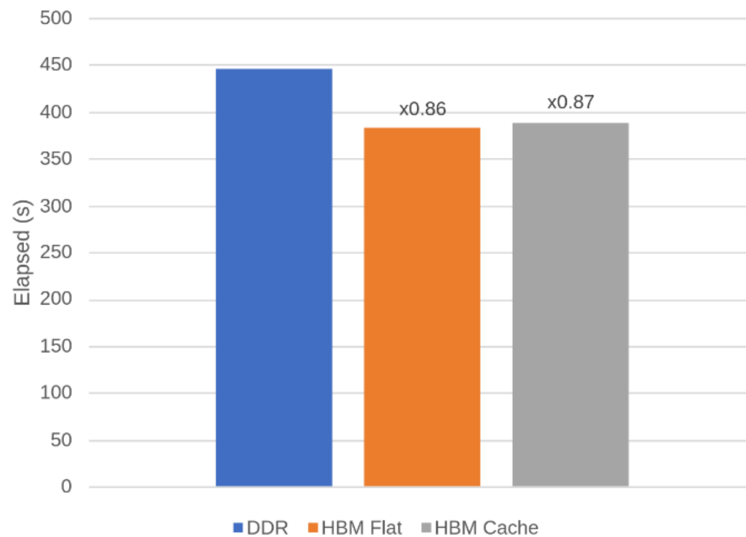

Fig. 22. BigDFT HBM sensitivity.

Using Intel VTune APS we got an overall vectorization ratio of 36%, showing that BigDFT is not very well vectorized. As shown on the figure below (Fig. 23), the main hotspot is the custom FFT baked in BigDFT. With the Intel compiler option -qopt-report=3 we could generate and analyze a vectorization optimization report. This tool reports that the main hotspot loop has been vectorized with strides (which limits performance) and we could obtain a 3.3 speedup on this loop in an ideal scenario. This shows that the application could benefit from better vectorization, it would improve data access efficiency and increase sensitivity to memory bandwidth and gain from HBM.

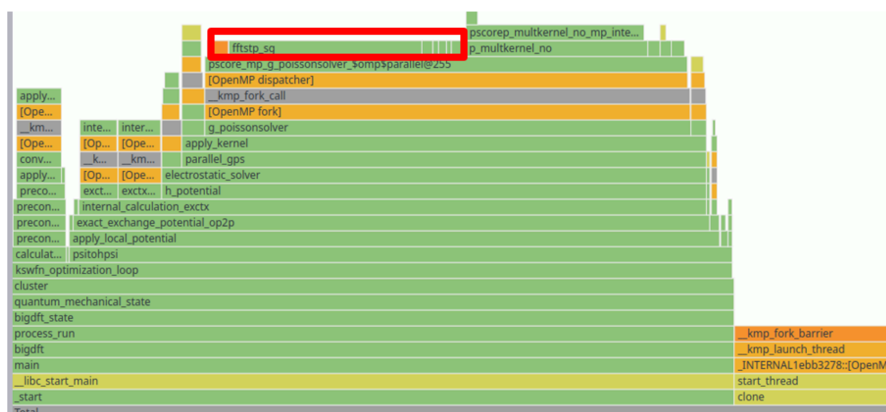

Fig. 23. Stack flamegraph showing custom FFT hotspot (circled in red), PBE0 test case.

H20-32 PBE test case

The PBE test case is short in total runtime (less than a minute). Its main hotspot is the convolution routine and has a very high synchronization overhead, as illustrated in the following figure (Fig. 24). We found no noticeable gain during our HBM study. Since the test case is very small, we believe it isn't a very representative run configuration and studied the test case PBE0 instead.

Fig. 24. Stack flamegraph showing load imbalance and convolution hotspot, PBE test case.

Fock Miniapp - SVE2 Vectorization

The miniapp Fock has been extracted from the version 1.9.5 of BigDFT. We have tested three different compilers, GCC, ACFL and NVHPC. The version details and dependencies are shown in the following Table 13:

<table>
<tr><th>Compiler</th><th>BLAS</th><th>UCX</th><th>OpenMPI</th></tr>
<tr><td>GNU 14.2.0</td><td>ARMPL 24.10.0</td><td rowspan="3">UCX 1.17.0</td><td rowspan="2">Open MPI 4.1.6</td></tr>
<tr><td>ACFL 24.04</td><td>ARMPL 24.04.0</td></tr>
<tr><td>NVHPC 24.9</td><td>ARMPL 24.10.0</td><td>Open MPI 4.1.7</td></tr>
<tr><td colspan="4">MAQAO 2.20.11</td></tr>
</table>

Table 13. Compiler, dependencies and tools version.

The study has been performed on a quad-socket Nvidia Grace node with 72 cores per socket. The CPU is a ARM Neoverse-V2 architecture, with 4x128b SVE2. We used the following compilation flags to compare SVE2 vectorization and NEON vectorization:

- `-mcpu=neoverse-v2 -msve-vector-bits=128`, for SVE vectorization;
- `-mcpu=neoverse-v2+nosve`, for NEON vectorization.

Our Fock miniapp uses 144 orbitals and a domain size of 216, and is called with the following parameters: `./Fock -g P -n 216 -o 144 -a No`. We used this configuration on a single Grace with 72 cores. We tested different combinations of MPI ranks and OpenMP threads of obtained the following result:

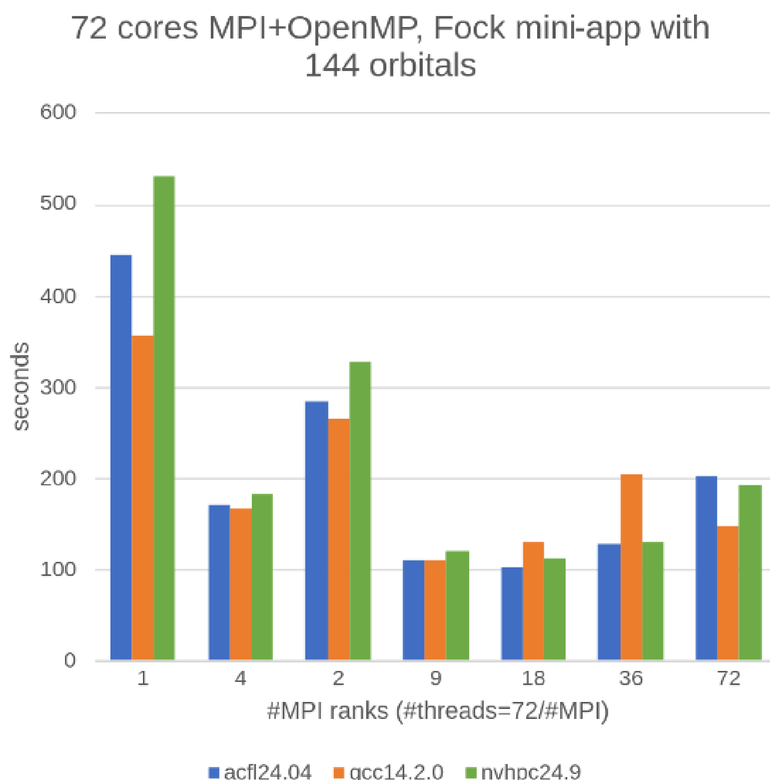

Fig. 25. Fock Mini-app scaling on Grace system

As we can see the best results per compiler can be summarized as follow, with the best result being with the ACFL compiler:

- 9 MPI ranks with 8 threads for GCC;
- 18 MPI ranks with 4 threads for NVHPC and ACFL.

We used MAQAO to get a first overview comparison between NEON vectorization (r0 in the following figure Fig. 26) and SVE vectorization (r1). We can conclude that SVE does not seem to give a significant speedup on NVIDIA Grace, most metrics are quite similar between the two runs.

► Compared Reports

Global Metrics

Metric		r0	r1
Total Time (s)		117.81	153.73
Max (Thread Active Time) (s)		95.37	96.09
Average Active Time (s)		65.99	66.87
Time in analyzed loops (%)		77.2	77.7
Time in analyzed innermost loops (%)		38.6	37.8
Time in user code (%)		96.8	96.8
Compilation Options Score (%)		100	100
Array Access Efficiency (%)		67.0	66.5
Potential Speedups			
Perfect Flow Complexity		1.00	1.00
Perfect OpenMP + MPI + Pthread		1.01	1.02
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.46	1.46
No Scalar Integer	Potential Speedup	1.45	1.45
	Nb Loops to get 80%	5	5
FP Vectorised	Potential Speedup	1.14	1.13
	Nb Loops to get 80%	4	4
Fully Vectorised	Potential Speedup	1.35	1.35
	Nb Loops to get 80%	9	8
Only FP Arithmetic	Potential Speedup	1.75	1.78
	Nb Loops to get 80%	10	10

► Compared Reports

Global Metrics

Metric		r0	r1
Total Time (s)		163.05	165.71
Max (Thread Active Time) (s)		83.92	83.85
Average Active Time (s)		79.64	79.50
Time in analyzed loops (%)		62.6	62.7
Time in analyzed innermost loops (%)		49.3	48.1
Time in user code (%)		80.7	80.8
Compilation Options Score (%)		99.8	99.8
Array Access Efficiency (%)		68.1	56.7
Potential Speedups			
Perfect Flow Complexity		1.00	1.00
Perfect OpenMP + MPI + Pthread		1.02	1.02
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.28	1.28
No Scalar Integer	Potential Speedup	1.30	1.33
	Nb Loops to get 80%	6	7
FP Vectorised	Potential Speedup	1.12	1.12
	Nb Loops to get 80%	5	5
Fully Vectorised	Potential Speedup	1.26	1.28
	Nb Loops to get 80%	9	9
Only FP Arithmetic	Potential Speedup	1.32	1.34
	Nb Loops to get 80%	9	10

Fig. 26. MAQAO performance report for NEON (r0) vs. SVE(r1) for GNU (left) and ACFL (right) compilers.

Unfortunately, at the time of the study, the MAQAO version could not profile SVE vectorization properly as it was introduced at a later date, so no in depth analysis of SVE vectorization has been performed yet for the Fock miniapp.

4.3.4. Siesta

The Siesta mini-app was tested in the two recommended configurations, using ELPA or MRRR algorithms and real diagonalization route. ELPA (https://gitlab.mpcdf.mpg.de/elpa/elpa) is an optimized library providing mathematical kernels optimized for HPC usages. ELPA provides vectorized kernels for many compute-intensive operations such as the diagonalization performed in Siesta and other MaX codes, making it a common dependency for MaX codes. ARM platform support is comprehensive and intrinsics-based NEON/SVE kernels for 128/256/512 bits-wide vectors are available. MRRR algorithm relies on scalapack calls, which itself make use of BLAS/LAPACK calls, usually provided optimized by vendors in toolchains.

Vectorisation

As the code heavily relies on external libraries, several variants of toolchains were compared and showed different performance, while compiler change did not show much impact. Input was covid-8k, provided alongside the mini-app, which works on ~57k orbitals.
Provided inputs are either small and finish in seconds, or quite large (covid-8k requires ~60 core hours with MRRR and more than 100GB memory), so for further analysis we will try to get mid-sized inputs from developers.

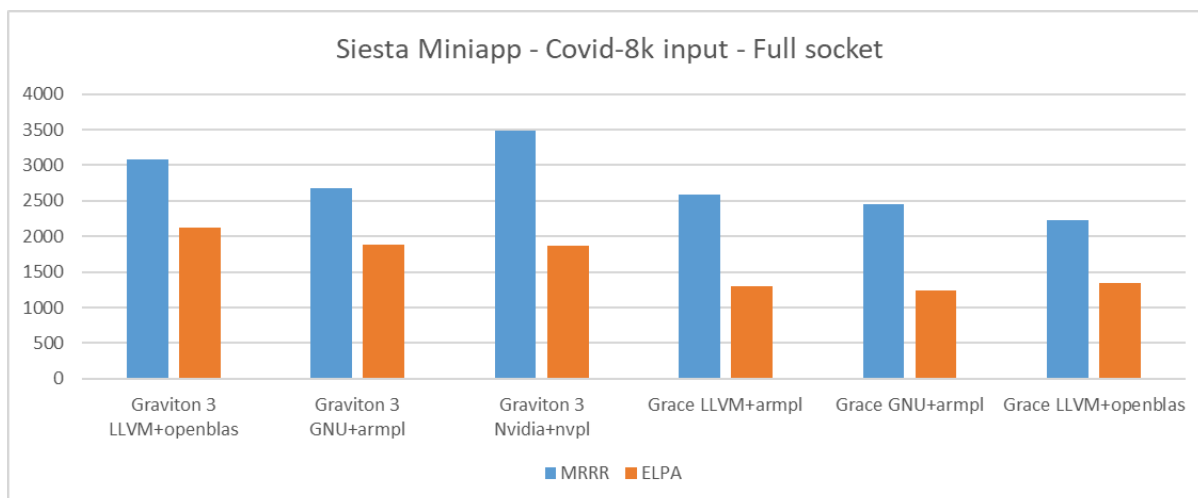

Fig. 27. Full socket ARM performance of Siesta mini-app with two different algorithms.

According to the results presented in Fig. 27 full socket performance with Grace (72 cores) is found to be 10 to 40% faster than Graviton 3, and the best performing BLAS/LAPACK library is ARM performance libraries. ELPA is also found to be much faster than MRRR, as expected.

Loop id	Source Location	Source Function	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
18297	libarmpl_mp.so -	dgemv_n_sve_kernel	31.08	100	100
18338	libarmpl_mp.so -	dgemv_t_sve_kernel	25.07	100	100
13091	libarmpl_mp.so -	dgemm_sve_big	16.21	100	93.55
11306	libarmpl_mp.so -	daxpy_sve_kernel	4.99	100	100
11770	libarmpl_mp.so -	ddot_sve_kernel	4.26	100	100
18520	libarmpl_mp.so -	interleave_2vl_sve_kernel_d	2.66	100	100
28289	libarmpl_mp.so -	tran_interleave_3vl_sve_kernel_d	1.32	100	100
28285	libarmpl_mp.so -	tran_interleave_2vl_sve_kernel_d	1.08	100	100

Fig. 28. Vectorisation report for MRRR run with ArmPl libraries on V2.

Loop id	Source Location	Source Function	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
13097	libarmpl_mp.so -	dgemm_vanilla_big	37.07	96.88	98.44
6280	libelpa.so.19.4.0 -	double_hh_trafo_real_SVE128_2hv_double	7.65	88.89	96.3
6281	libelpa.so.19.4.0 -	double_hh_trafo_real_SVE128_2hv_double	6.66	85.71	92.86
18524	libarmpl_mp.so -	interleave_3vl_sve_kernel_d	3.04	100	100
28214	libarmpl_mp.so -	t_interleave_kernel_d8	1.96	100	100
19991	libarmpl_mp.so -	n_interleave_kernel_d8	0.51	96	98
4	libastring.so -	__memcpy	0.32	0	100
6282	libelpa.so.19.4.0 -	double_hh_trafo_real_SVE128_2hv_double	0.18	100	100

Fig. 29. Vectorisation report for ELPA run with ArmPl libraries on V2.

MAQAO analysis (vectorisation reports Fig. 28 and Fig. 29) confirms that most of the time is spent inside fully vectorized BLAS/LAPACK and ELPA routines, which is consistent with expectations.

Scaling shown in Fig. 30 was found to be better with ELPA than with MRRR, as expected also, but still under the target (only 4.3 with 8 times more core). As most of the time is spent inside heavily SVE-vectorized BLAS routines, this may be symptomatic of the first generation of Graviton 3 processors having troubles with heavy compute-bound SVE loads, such as ZGEMM. This was particularly noted with other DGEMM-heavy benchmarks, such as HPL, where full-node performance improved by 35% after switching to Graviton 3E-based instances, which are understood to fix these issues.

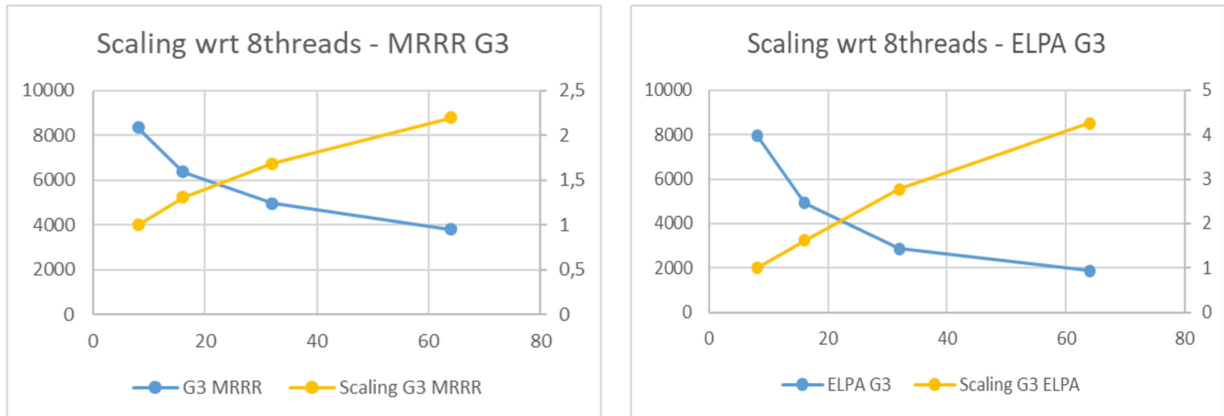

Fig. 30. Scaling for Siesta Miniapp - Covid 8k input, Graviton 3, ranging from 8 to 64 MPI processes.

Indeed, scaling with a Grace processor gives better results (see Fig. 31), with a 73% efficiency from 8 to 72 processes.

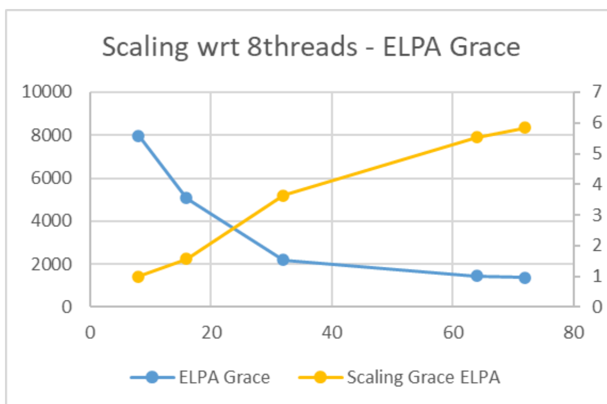

Fig. 31. Scaling for Siesta Miniapp - Covid 8k input, Graviton 3, ranging from 8 to 64 MPI processes.

Due to a timer bug in siesta miniapp (reported and now fixed), these scaling experiments were done without OpenMP. This caused MPI overhead to become important and represent ~25% of total execution time with 72 processes. It is understood that scaling will improve with a hybrid OpenMP/MPI spread.

Name	Module	Coverage run_0 (%)
▶ dgemm_vanilla_big	libarmpl_mp.so	49.82
○ mca_btl_sm_component_progress	libopen-pal.so.80.0.5	19.91
▶ double_hh_trafo_real_SVE128_2hv_double	libelpa.so.19.4.0	15.57
▶ __aarch64_ldadd4_relax	libmpi.so.40.40.7	3.3

Fig. 32. Function coverage of a 72 MPI process COVID-8 ELPA run on Grace.

Intel runs were perturbed by issues with OneAPI scalapack, which prevents runs for completing. Furthermore covid test case is too large for HBM studies, and no HBM as cache system was available at that time. Experiments with Siesta miniapp are ongoing.

As the MiniApp already makes use of very optimized routines and compute-bound kernels such as `dgemm` (Fig 32.), it is not expected that much optimization can be achieved once scaling has been confirmed to be good.

HBM vs DDR

To determine whether using HBM provides benefits when running Siesta, we performed tests with the full Siesta code on an Intel Sapphire Rapids node. In particular, for this benchmark, we used the code commit `6ad11e` from the official Siesta repo (https://gitlab.com/siesta-project/siesta) and we executed the si-quantum-dot simulation available on the MaX benchmarking repository. (https://gitlab.com/max-centre/JUBE4MaX/-/tree/develop/max-inputs/workloads/siesta). Note that the input file was slightly modified to run on a CPU only system and the following options were specified:

- ```
  %block supercell
  2 0 0
  0 2 0
  0 0 2
  %endblock supercell
  ```
- `PAO.BasisSize        sz`

Simulations were performed on the Partition 10 (Sapphire Rapids-HBM Server) of the complementary system cluster at IT4I. The node features 2 CPUs Intel® Xeon® Max 9468, each with 48 physical cores (base clock speed 2.1GHz, peak clock speed 3.5Ghz). The node is equipped with 16 banks of DDR5 RAM for a total of 256 GB of memory. Beside the RAM, each socket has 64 GB on-package HBM memory. The machine is set up in FLAT mode, which exposes
  ```

HBM memory as memory-only NUMA nodes, with each 12-core tile providing 16 GB of memory. For I/O, 2 Intel D3 S4520 960 GB SATA 6Gb/s are provided.

The following table illustrates the system and software configuration used for these tests:

OS	Compiler	BLAS and LAPACK	ScaLAPACK	MPI	numatcl	elsi
Rocky Linux 8.10	GNU 11.3.0	OpenBLAS 0.3.20	Netlib 2.2.0	Open MPI 4.1.4 UCX 1.12.1	2.0.14	2.9.1

Table 15. System and software configuration on the Sapphire Rapids-HBM Server at IT4I.

All the dependencies, except elsi were provided as environment modules on the system. The library elsi was compiled using CMake 3.23.1 and specifying the following compiler flags:

- Fortran: `-O3 -ffree-line-length-none -fallow-argument-mismatch -fopenmp -march=sapphirerapids -mtune=sapphirerapid`
- C: `-O3 -std=c99 -fopenmp -march=sapphirerapids -mtune=sapphirerapids`
- C++: `-O3 -std=c++11 -fopenmp -march=sapphirerapids -mtune=sapphirerapids`

When building Siesta using CMake 3.23.1, the following options were passed to generate the Makefile:

```
-DCMAKE_C_COMPILER=gcc
-DCMAKE_CXX_COMPILER=mpicxx
-DCMAKE_Fortran_COMPILER=gfortran
-DSIESTA_WITH_MPI=ON
-DSIESTA_WITH_ELSI=ON
-DSIESTA_WITH_NO_MPI_INTERFACES=ON
-DSIESTA_WITH_OPENMP=ON
-DLAPACK_LIBRARY=openblas
-DBLAS_LIBRARY=openblas
-DSCALAPACK_LIBRARY=scalapack
-DCMAKE_Fortran_FLAGS="-O3 -ffree-line-length-none -fallow-argument-mismatch
-march=sapphirerapids -mtune=sapphirerapids"
-DSIESTA_WITH_FLOOK=FALSE
```

We conducted tests by reserving all the Sapphire Rapids nodes in an exclusive way and asking 96 tasks per node, 48 tasks per socket, and 1 cpu per task via SLURM. Siesta is parallelised via MPI and OpenMP. However, in these tests, we did not leverage OpenMP parallelisation by

exporting `OMP_NUM_THREADS=1`. Overall, we did not observe performance improvements with OpenMP on other platforms, so we decided to investigate its impact on HBM and DDR at a later stage.

Simulations were launched with the following command

```
mpirun -n $mpitask --bind-to core --map-by ppr:$mpitask:node:PE=$dist
--report-bindings numactl --membind <0-7 | 8-15>
```

Where mpitask is the number of MPI tasks selected and dist = 96/mpitask. On the IT4I platform, NUMA nodes from 0 to 7 contain 12 cores and 32 GB of DDR DRAM, while NUMA nodes from 8 to 15 have no cores and 16 GB of HBM each.

Results are displayed in the following figure (Fig. 33), where walltime for runs with 8, 48, and 96 MPI tasks with DDR and HBM are reported. Each run was repeated 5 or 10 times to ensure reproducibility and error bars are shown.

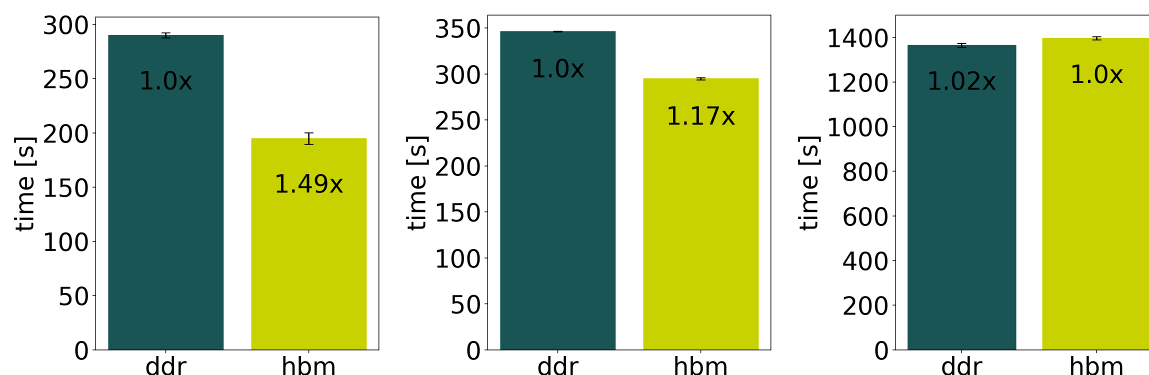

Fig. 33. Walltime for Siesta runs with 96 (left panel), 48 (central panel), and 8 (right panel) MPI tasks. Results are shown for bindings to NUMA cores 0–7 (dark green) and 8–15 (light green). Speedup values, relative to the longer time-to-solution in each case, are indicated in the plots.

These results indicate that utilising all available cores on the node while leveraging HBM memory leads to shorter time-to-solution when running Siesta. Specifically, a speedup of 1.49× was observed compared to simulations using only DDR memory. However, the performance advantage diminishes as fewer MPI tasks are used. Notably, when running with only 8 cores, the code performs slightly better when bound to DDR memory. Thus, full Siesta results appear to be consistent with those obtained from synthetic tests (Figure 4 of this report), which showed similar HBM and DDR bandwidth when using one core per tile (as is the case with 8 cores), and a theoretical peak bandwidth, and thus the maximum difference, when using 12 cores per tile.

4.3.5. FLEUR

Only preliminary runs were performed, using graviton 3 instance, Nvidia Grace and Sapphire rapids node with HBM.
Compilers tested were oneAPI 2025.1.0, LLVM sipearl 21, GNU15, Nvidia 25.3
BLAS/LAPACK libraries were OneMKL 2025.1.0, OpenBLAS 0.3.29, ArmPl 24.10, nvpl 25.3

Tests were run using CuAg "small" run, which takes ~8 core hours, making it suitable for single node tests (https://gitlab.com/max-centre/benchmarks/-/tree/master/FLEUR/inp_0.33).

Full node results - 8 OpenMP/MPI process are presented in Fig. 34.

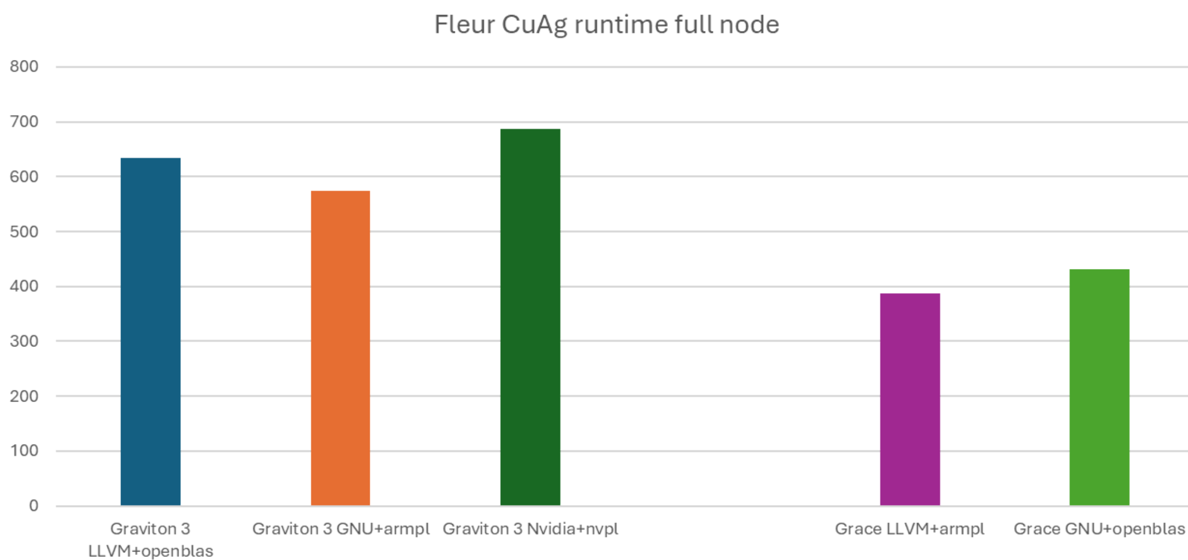

Fig. 34. Fleur ARM full node performance using various setups

ArmPL seems to be the more efficient BLAS Implementation. Indeed most of the time is spend in Diagonalization which makes heavy use of scalapak routines, and lapack/BLAS calls.

Most costly parts on the computation (GNU ArmPL version on Graviton 3)

```
  Diagonalization  ->SCALAPACK call        392.23sec  =  0h  6min 32sec ->  70.0%
  MT part          ->non-spherical setup    83.03sec  =  0h  1min 23sec ->  14.8%
  MT part          ->spherical setup        36.30sec  =  0h  0min 36sec ->   6.5%
```

70% of the time is spent in Diagonalization, and 91% in these 3 routines.

Loop id	Source Location	Source Function	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
29164	libarmpl_mp.so -	zgemm_sve_big_X_X	29.13	100	94.9
18323	libarmpl_mp.so -	zgemv_n_sve_kernel	24.08	100	100
18376	libarmpl_mp.so -	zgemv_c_sve_kernel	5.55	100	100
18374	libarmpl_mp.so -	zgemv_c_sve_kernel	4.63	100	100
18370	libarmpl_mp.so -	zgemv_c_sve_kernel	3.53	100	100
189207	libarmpl_mp.so -	void armpl::clag::spec::(anonymous namespace)::sve_interleave<armpl::clag::spec::sve_architecture_spec>::operator()<std::complex<double> const, std::complex<double> >(armpl::clag::general_matrix<armpl::clag::(anonymous namespace)...	1.53	0	25
280	fleur_MPI - hsmt_sph.F90:356-356	__m_hsmt_sph_MOD_hsmt_sph_cpu._omp_fn.0	1.04	100	100
28293	libarmpl_mp.so -	tran_interleave_4vl_sve_kernel_z	1.02	100	100

Fig. 35. FLEUR report on most time consuming kernels and vectorisation

Maqao confirms that most of the time is spent in BLAS routines such as GEMM and GEMV, working on complex double type (see Fig. 35). These routines are heavily optimized and vectorized by BLAS libraries. First FLEUR routine only accounts for 1% of the total run.

Strong scaling on graviton 3 was assessed, from 2 to 64 cores, as single core run is very time consuming on this cloud platform.

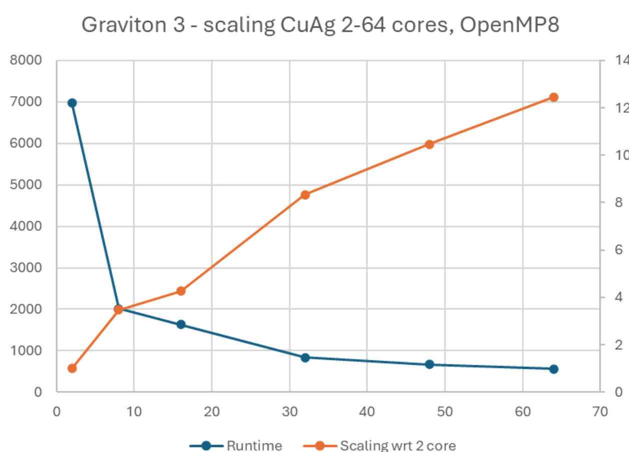

Fig. 36. FLEUR scaling on Graviton 3, full run

As seen in Fig. 36, scaling is shown to slow down and only reach 12 for the full application (with 32 times more cores). As most of the time is spent inside heavily SVE-vectorized BLAS routines, this is still probably the same issue of Graviton 3 processors having troubles with heavy compute-bound SVE loads, as reported with Siesta.

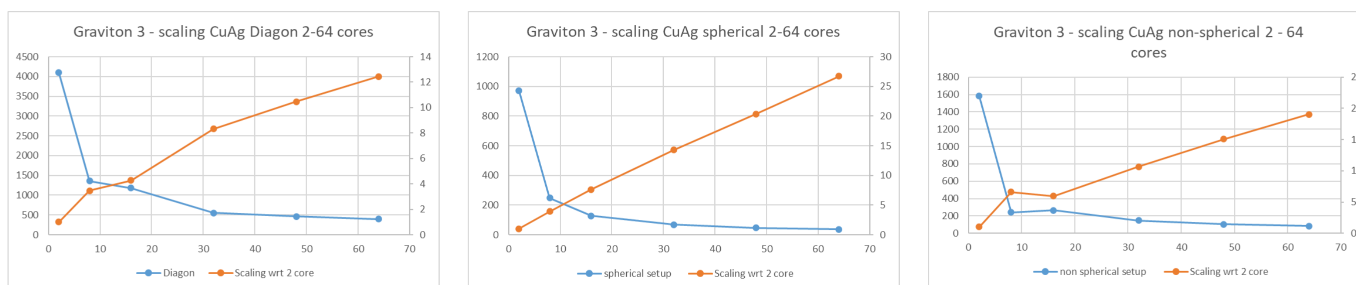

Fig. 37. Scaling for each of the three main kernels of FLEUR on Graviton 3 from 2 to 64 cores.

Scaling in Fig. 37 for all major parts of the application shows indeed that the Diagonalization part only reaches a scaling factor of 12 when the number of cores goes from 2 to 64.
This can be mitigated by moving to more recent Graviton 3E nodes, which should not exhibit this slowdown with compute-bound BLAS kernels.

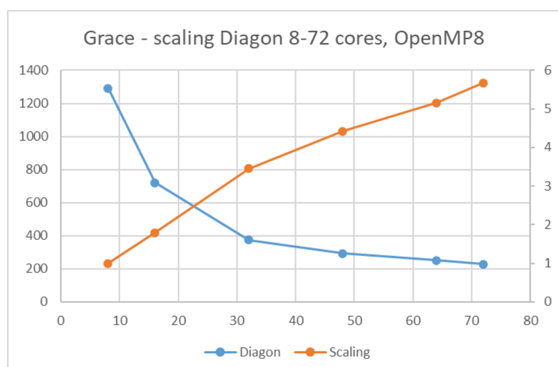

Fig. 38. FLEUR scaling of Diagon kernel on Grace

The same experiment was run on Grace, with core numbers varying from 8 to 72 (full socket). While perfect scaling would be 9 in this case, the Diagon part reaches a value of 6, which is better than Graviton 3.

Scaling up to 32 nodes is almost perfect, butl shows a margin for improvement after (Fig 38.).

As FLEUR mini-app would be targeting the setup parts of the computation (spherical and non spherical), further studies will focus on these.

HBM case

HBM test was run on the most optimistic case for HBM, with only 10 OpenMP processes attached to consecutive cores on a Sapphire Rapids 40-core CPU. This means that a single HBM bank was solicited, reducing potential overhead by communication between banks. More tests will have to be performed during the course of this project.

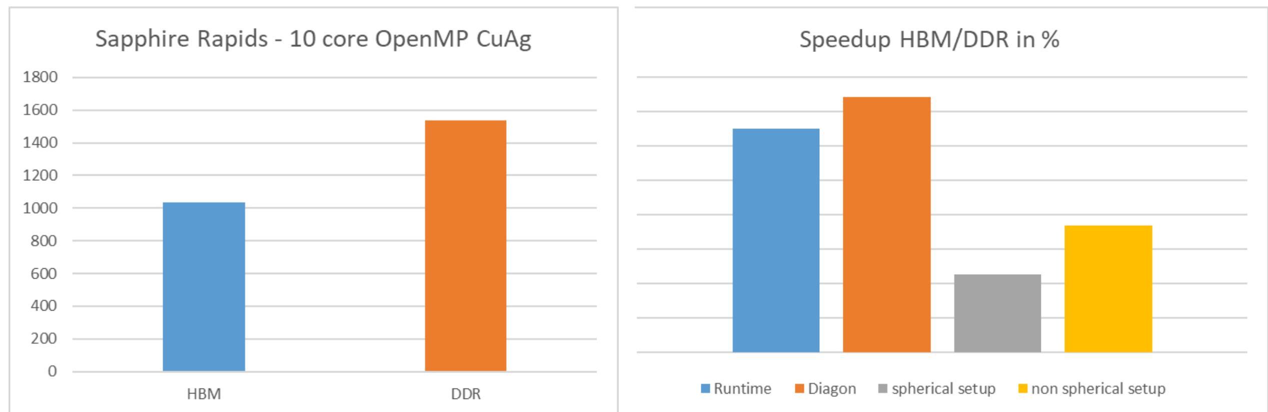

Fig. 39. First DDR/HBM comparison with FLEUR on a single HBM bank, with 10 threads on Sapphire Rapids 9460.

As one can see in Fig. 39, the diagonalization part benefits a lot from HBM, with a 35% improvement. ZGEMM part itself should be compute bound and moderately impacted, but it still benefits on sapphire rapids platforms from the more stable environment caused by HBM (better CPU frequency and power handling compared to constant waiting times with DDR which may cause frequency to change). MKL uses ZHEMV and ZGEMM as the main kernels for this scalapack computation part of the computation. When DDR is used, 50% of all execution time is spent inside the ZHEMV kernel, and 30.5% in ZGEMM. When HBM is used, ZHEMV accounts only for 42% of the time, and ZGEMM for 31%. This shows HBM benefits more to ZHEMV than ZGEMM (vector/matrix multiplication is indeed more compute bound than matrix/matrix multiplication).

4.4. Recommandations

4.4.1. Quantum Espresso

For HBM, the code section targeted by the wave mini-application of Quantum Espresso greatly benefits from HBM. Testing other parts of Quantum Espresso (or the full application) might be interesting to check if such gains are application-wide or if this code section should be targeted for specific memory allocations in HBM.

For the ARM ecosystem, we found out that vectorisation for ARM compilers can be improved compared to x86 for at least two locations in the code. We proposed optimisations for both of them, and we recommend backporting the proposed optimisations to Quantum Espresso to improve its efficiency for ARM-based CPUs. Since both optimisations are located in the fftxlib7 library of Quantum Espresso, its integration should be easier.

For SVE vectorisation on ARM, we can argue that SVE code is properly generated by GNU and ACFL compilers on Quantum Espresso second mini-application. However, SVE does not outperform NEON vectorisation on the NVIDIA Grace CPU. It is an expected result on such processors, as NVIDIA Grace has the same number of SVE and NEON vectors of the same size (4x 128b). Rhea CPU, with 2x256b SVE vectors, may yield better results with SVE vectorisation than NVIDIA Grace CPU for Quantum Espresso.

4.4.2. YAMBO

No code adaptations were possible at the time of the study as the Yambo mini-app was not numerically stable due to volatile random seed and uninitialized variables.
Improving OpenMP performance and parallel efficiency should be an important focus point. It can be started by studying better load balancing or investigating false sharing effects. We also recommended finding algorithmic solutions to not rely on level 1 BLAS routines as much.

Since our study and based on frequent feedback from our side, the developer has developed several improvements on the mini-app. The time measurement of the compute section is now more accurate as the random initialization of values is now not taken into account (it is a slow process).
The randomization is now fixed with a seed and several uninitialized variables have been adjusted. We are now able to have reproducible runs, which will be important for future performance optimizations.

The data collect algorithm has been changed to take advantage of BLAS3 gemm instead of BLAS1 axpy. It improved performance of the compute section (Xo procedure) significantly by about 60%.

A new study of the Yambo mini-app would be necessary as the application profile might have drastically changed. It would be especially interesting to do another vectorization analysis to find out why the application is not vectorized at all.

4.4.3. BigDFT

BigDFT aims to provide a new adaptation for the Fock mini-app and the poisson solver, allowing multiple backends for the FFT computation, alongside the original Goedecker FFT and the CUDA version. These additional backends would be SYCL-based mostly, using either Intel double batched FFT libraries, or UXL's OneMath interfaces to provide broader support for existing CPUs and GPUs. SiPearl implemented an ArmPL-based backend to OneMath interfaces in 2025, and works with BigDFT developers in the context of MaX to ensure that these developments make efficient use of T4.1 targets and study further optimization.

4.4.4. Siesta

As of the writing of this document, there have not been particular recommendations for Siesta. The mini-app already shows its use of optimized and vectorized kernels, either through vendor libraries or domain specific optimized libraries such as ELPA.

Further tests for the full application will take place during the next few months, and tests with a smaller test case will also be used for the mini-app once available.

4.4.5. Fleur

As no Mini-app was available yet, FLEUR was not extensively studied in the first phase of T4.1. Vectorisation has been found to be good for the biggest parts of the application, and that it would benefit from HBM. Full application profile shows that the plan for the mini-app development would be viable for studying several setup kernels which can be costly and hinder scalability. Focus will then be on extraction of said mini-apps in the next few months to allow fine grained analysis.

5. Conclusion

In the initial phase of the T4.1 task on co-design, the key achievement was the release of five mini-apps extracted from MaX codes. These serve as co-design vehicles, enabling to proceed with fine grained analysis. Analysis of the behaviour of these mini-apps has been started, and some feedback has already been provided to the developers and the platform developers, in order to improve performance. Additional mini-apps are scheduled for release in the coming months.

A methodology to perform these analyses has been drafted, using hardware with analog capacities as the targeted platforms together with tools from the community or developed for this project, such as IT4I HBM tool, which will be used for all mini-apps in the project. Portability of the applications and mini apps also has been studied, mainly targeting ARM platforms, with a focus on compilers and toolchains adequation. Vectorisation indeed is highly compiler-dependent and much performance can be extracted from giving feedback to compiler developers about the opportunities for optimization that may be missed in current versions.

In the next months more analysis will be performed, and most mini-apps will be integrated in SiPearl's internal test suites to run on the hardware emulators and the real hardware when available. This will help confirm the adequation of the recommendation for the targeted hardware platforms.

As Rhea's availability has been delayed, the workload of this package has been adapted to match with the estimated timeline of the project, with a higher load scheduled for the end of the project.