D3.2
Co-Design and Energy Efficiency Report I

Project Title	dealii-X: an Exascale Framework for Digital Twins of the Human Body
Project Number	101172493
Funding Program	European High-Performance Computing Joint Undertaking
Project start date	1 October 2024
Duration	27 months

dealii-X has received funding from the European High-Performance Computing Joint Undertaking Programme under grant agreement Nº 101172493

Deliverable title	Co-Design and Energy Efficiency Report
Deliverable number	D3.2
Deliverable version	v1
Date of delivery	August 31, 2025
Actual date of delivery	August 31, 2025
Nature of deliverable	Report
Dissemination level	Public
Work Package	WP3
Partner responsible	BADW-LRZ

Abstract	This report details the progress made during the first 11 months of the dealii-X project concerning high-performance kernels for finite element problems. The focus is on benchmarking of algorithms, porting to different GPU architectures and analysis of energy efficiency.
Keywords	performance; benchmarks; GPU programming; Kokkos; matrix-free finite elements

Document Control Information

Version	Date	Author	Changes Made
0.1	22/08/2025	Ivan Pribec	Initial draft
0.2	25/08/2025	Enes Mustafa Soydan	Added results of RUB group
0.3	29/08/2025	Martin Kronbichler	Review
1.0	31/08/2025	Ivan Pribec	Final version

Approval Details

Approved by: Martin Kronbichler
Approval Date: August 31, 2025

Distribution List

- Project Coordinators (PCs)
- Work Package Leaders (WPLs)
- Steering Committee (SC)
- European Commission (EC)

Disclaimer: This project has received funding from the European Union. The views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European High-Performance Computing Joint Undertaking (the “granting authority”). Neither the European Union nor the granting authority can be held responsible for them.

Contents

1 Introduction

This report describes our advances investigating novel hardware and software solutions, and evaluating emerging technologies for exascale computing with the final goal of achieving state-of-the-art performance and optimal technology exploitation.

Our main achievement in this period has been the preparation and analysis of benchmark kernels for the matrix-free application of discrete finite element operators, see also our overall frameworks described in Kronbichler and Kormann (2012, 2019), focused on addressing newer GPU architectures.

The benchmark repository is made publically available at: `https://github.com/dealii-X/benchmarks/`

The preparation of these benchmarks has been the focus of Milestone #3 – Public repository of benchmark sets for performance evaluation – verified by means of runs of the software on at least three different accelerated platforms.

For performance optimization purposes we have focused on two families of kernels, the CEED[1] Bakeoff Problems[2] and Streaming Kernels, that serve as a suitable proxy for the algorithmic core of simulation codes using high-order finite elements.

The CEED Bakeoff Problems (Fischer et al., 2020) are a benchmark to measure the evaluation speed of the basic finite element operators (discretized bilinear forms), posed for problems on hexahedral grids, and the primary focus being sum factorization techniques. Both scalar and vector problems are included in the benchmarks. So far our efforts have been focused on the scalar problems. Each of the Bakeoff Problems (BP) has an associated Benchmark Kernel (BK) that evaluates a particular finite element operator and is of high arithmetic intensity. The efficient implementation of these kernels is paramount to obtain good performance.

The benchmark kernels are based on a numerical quadrature approach to evaluate the finite-element integrals and exploit algebraic sum factorization techniques (Deville et al., 2002) to achieve an affordable evaluation of high-order finite element operators in comparison to the naive nested sum algorithm or dense interpolation matrices. This type of technique has been popularized in recent years by several

[1] Center for Efficient Exascale Discretization, Exascale Computing Project, `https://ceed.exascaleproject.org/`

[2] `https://libceed.org/en/latest/examples/bps/`

groups, including the deal.II authors (Kronbichler and Kormann, 2012, 2019; Arndt et al., 2021). The translation of these techniques to modern GPU architectures, see also the work by Świrydowicz et al. (2019), remains an open challenge at both software and hardware level due to the heterogeneity of both devices (CPUs, GPUs and other accelerators) and parallel programming frameworks provided by different vendors.

The Streaming Kernels (Chalmers and Warburton, 2020, Table 1) capture a second group of low intensity operations, where memory movement is dominant. The kernels generally take one or two long vectors as arguments. The first four kernels included are: copy (COPY), scaled sum of vectors (AXPBY), squared norm (NRM2), and inner product of two vectors (DOT). The fifth kernel is the fused conjugate gradient update that involves the combination of a scaled sum and norm of a residual vector in a single sweep. The last two kernels are the gather and scatter kernels that involve packing and unpacking the degrees of freedom associated with a given finite element mesh into a linear vector. The gather and scatter kernels are tied to a particular mesh and finite element space, requiring more extensive driver setup. Hence, they are not addressed in this report.

- CUDA Support: Enabling programming support for NVIDIA GPUs using the CUDA parallel computing model and supporting compatibility with AMDs HIP framework.
- SYCL Integration: Incorporating SYCL (Standard C++ for heterogeneous computing) for programming heterogeneous systems using C++. Porting from and compatibility with CUDA.
- Using OpenMP and/or OpenACC offloading techniques and ensuring their compatibility on different platforms.
- Employing parallel programming techniques like SIMD (Single Instruction, Multiple Data) for efficient accelerator and CPU utilization.
- We will support programming approaches and frameworks such as Kokkos (already supported by deal.II) and RAJA that supporting a range of accelerator programming paradigms ensure flexibility and compatibility with different hardware architectures, enabling developers to harness the full potential of accelerators in their applications. This avoids maintaining paradigm-specific code branches in parallel.

2 CEED Bakeoff Problems

To study the performance and portability of high-order finite element implementation we focus on a subset of the CEED Bakeoff problems, designed to test and compare the performance of high-order codes.

The subset we have focused on so far are,

- BK1: scalar PCG with mass matrix, $q = p + 2$
- BK3: scalar PCG with stiffness matrix, $q = p + 1$
- BK5: scalar PCG with stiffness matrix, $q = p + 2$

The benchmarks kernels form the algorithm core of the benchmark problems (BP), which include a complete finite element workflow, including mesh setup, partitioning, and parallel computation.

The initial kernels have been implemented in three variants: 1) sequential kernels for verification, 2) using Kokkos Core, a C++ programming model for writing portable applications, part of the larger Kokkos C++ Performance Portability Programming Ecosystem[3] and 3) using the CUDA Driver API. We note that the deal.II library currently bases the portable kernels on Kokkos, and the investigations presented in this report have the objective of providing insight into the behavior of the methods on modern hardware. In order to evaluate the impact of parallelization strategies, the Kokkos and CUDA-based kernels have several sub-variants that use different thread-mapping strategies, i.e., the assignment of degrees-of-freedom and quadrature points to GPU threads.

In order to obtain a wider impression of GPU programming models available we have also created variants using OpenCL, OpenMP target offloading and a variant for the Tiny Tensor Compiler, a recently introduced domain specific language for tensor operations from Intel.

In the following sub-sections we describe our preliminary findings from the BK1 benchmark. We present experiments from multiple accelerator devices, showcasing their throughput capabilities in terms of degrees of freedom updated per second versus the problem size. In the BK benchmarks, the problem size is measured in

[3] Kokkos, `https://kokkos.org`

number of degrees of freedom (DoFs), a number obtained by the number of polynomial basis functions per element times the number of elements. Besides the variation w.r.t. problem size, we also study the influence of the quadrature order.

In the experiments presented below, the following compilers (including OpenMP) are used:

- Intel oneAPI Toolkit version 2024.1 ('icpx')
- Nvidia HPC SDK v2024.5 ('nvc++')
- Tiny Tensor Compiler v0.3.1

The libraries used are

- Kokkos Version 4.6.2
- Intel OpenCL (driver version:)
- Nvidia OpenCL (driver version:)

2.1 Kokkos and CUDA Performance Results

BK1. Early performance results for the different test cases are shown in Figure 1 and Figure 2, evaluated on an Nvidia H100 GPU. Each test case uses 3D thread blocks, determines the number of quadrature points at run time, and employs thread blocks for element-wise computations. The only difference between the two cases is the programming model: Figure 1 presents the results obtained with CUDA, while Figure 2 shows those from the Kokkos library. The first finding is that polynomial order 1 shows lower performance compared to higher-order elements in both test cases. This behavior, rooted in a first-order polynomial, is configured with 27 threads per block, see the benchmarks described in Fischer et al. (2020). This leads to under-utilization of the warp (size 32) on the target hardware. The second finding is that Kokkos achieves approximately 90% of the performance observed with the CUDA model, demonstrating that the performance portability model introduces only a minor overhead.

BK5. Unlike the multiple polynomial order evaluations in the BK1 test cases, we analyzed the kernels with a fixed cubic polynomial order (p=3) and templated

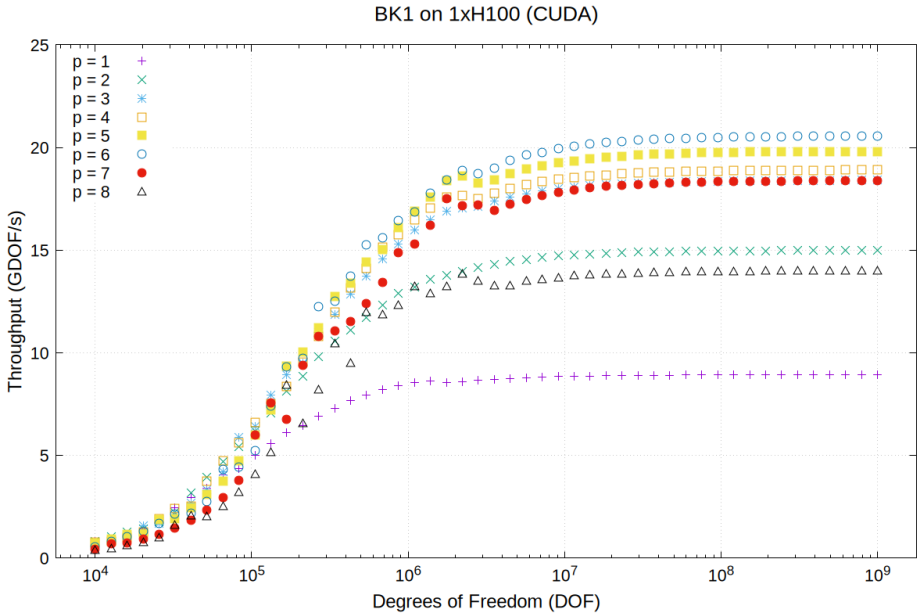

Figure 1: BK1 performance results on Nvidia H100 GPU with CUDA programming model. Various polynomial orders are tested, and saturation of hardware resources is visible after 10^6 degrees of freedom. Polynomial order 1 performs the worst due to under-occupation of the single warp.

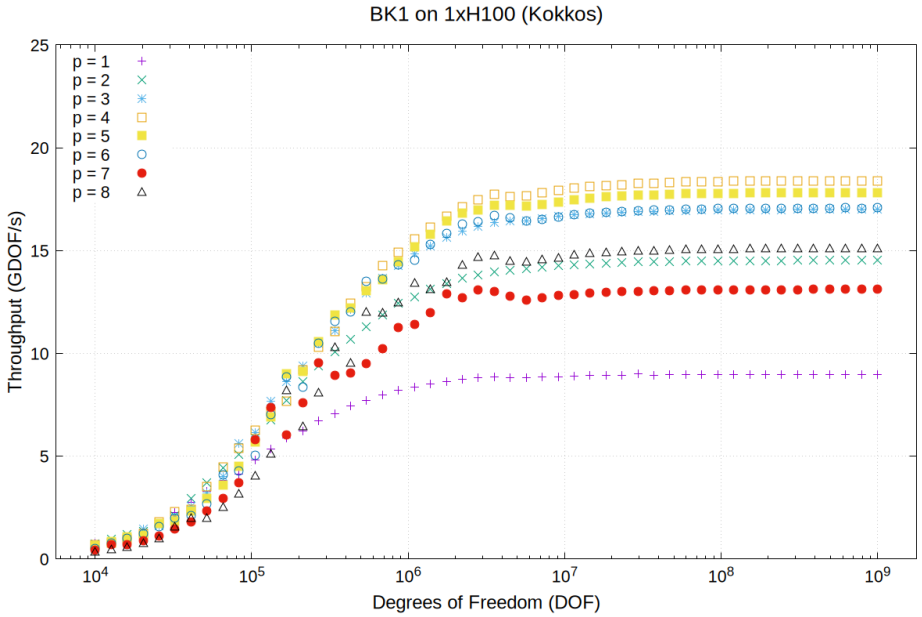

Figure 2: BK1 performance results on an NVIDIA H100 GPU using the Kokkos library. The same test conditions and optimization techniques as in Figure 1 were applied. Kokkos attains nearly 90% of the performance delivered by CUDA.

quadrature points. Using templates shifts certain computations from runtime to compile-time, allowing the compiler to apply optimizations. As a result, templated implementations are often significantly faster (Kronbichler and Kormann, 2019), in most cases improving performance by 30% at least, compared to their runtime counterparts.

Both CUDA and Kokkos kernels were implemented with three-dimensional thread blocks and a simple data mapping approach, where each thread operates on a single data entry. The total number of degrees of freedom (DOF) in this test case is 27 million. Kernel performance values were measured by NVIDIA Nsight Compute. In these measurements, both CUDA and Kokkos executions completed in 0.71 ms, showing identical performance. Nsight Compute reported an arithmetic intensity of 1.97 FLOP/byte, which places these kernels in the memory-bound region of the roofline model for the H100 GPU. The achieved performance was 5.7 TFLOPs, compared to the hardware limit of 6.6 TFLOPs at this arithmetic intensity. This corresponds to 86% of peak bandwidth utilization (2.88 TB/s out of 3.2 TB/s). The kernels also achieved 75% occupancy, and register usage per thread was identified as the primary limiting factor.

Overall, the results demonstrate that both CUDA and Kokkos BK5 implementations deliver near-optimal performance on the H100, efficiently exploiting available memory bandwidth and achieving performance levels very close to the hardware limits.

2.2 OpenCL

For experimentation across a wider array of potential accelerator devices, the CUDA kernels have also been translated to the OpenCL-C language for use with the vendor-neutral OpenCL framework from Khronos. OpenCL remains widely supported across both CPU and GPU devices from multiple vendors. The verbose setup required via the OpenCL runtime API, the split kernel source model, and restriction to C level language semantics, are some of the main criticisms from scientific software developers. The growing popularity of C++ coupled with powerful syntactic features has shifted the attention to more expressive programming models including SYCL and Kokkos, which provide similar capabilities as OpenCL, but are arguably simpler to use.

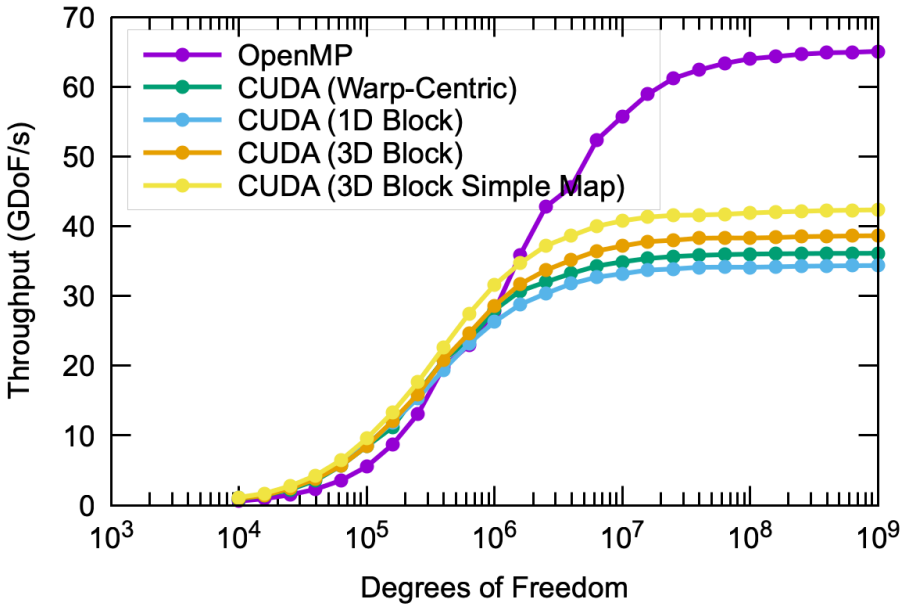

Figure 3: BK1 performance for templated kernel, $q = 4$, Nvidia GH200 (Hopper).

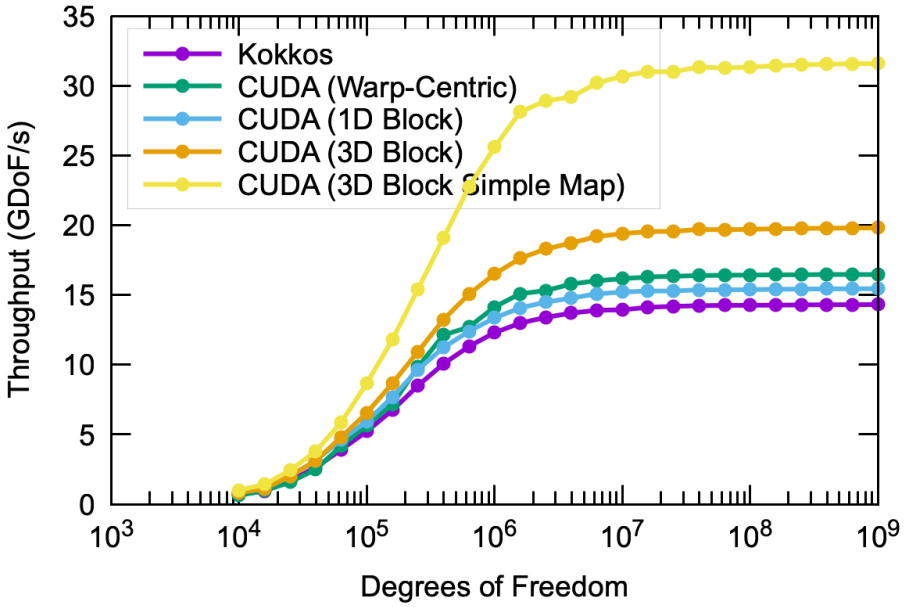

Figure 4: BK1 performance for variable kernel, $q = 4$, Nvidia GH200 (Hopper).

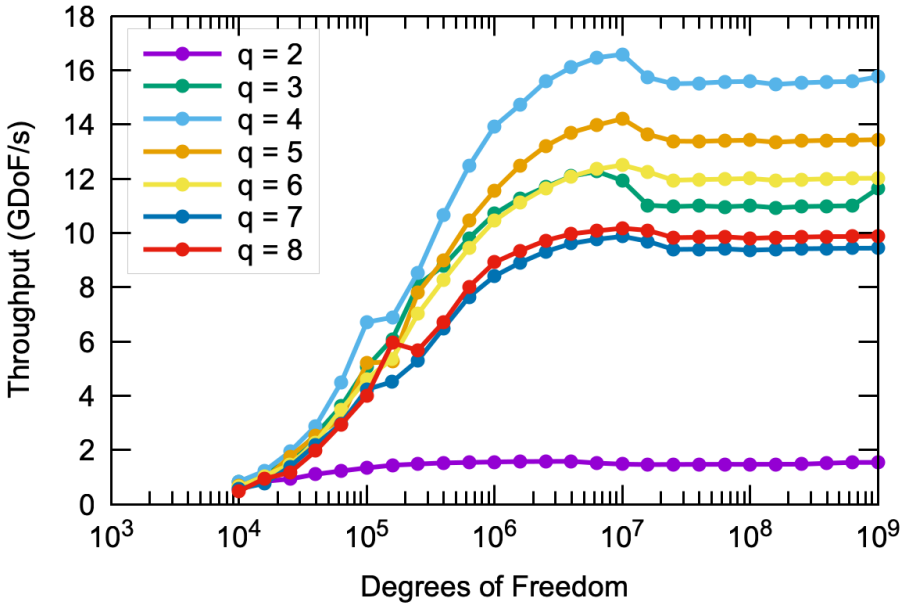

Figure 5: BK1 using OpenCL and 3D Simple Map work-item strategy, Intel Max Series 1550 GPU (Ponte Vecchio).

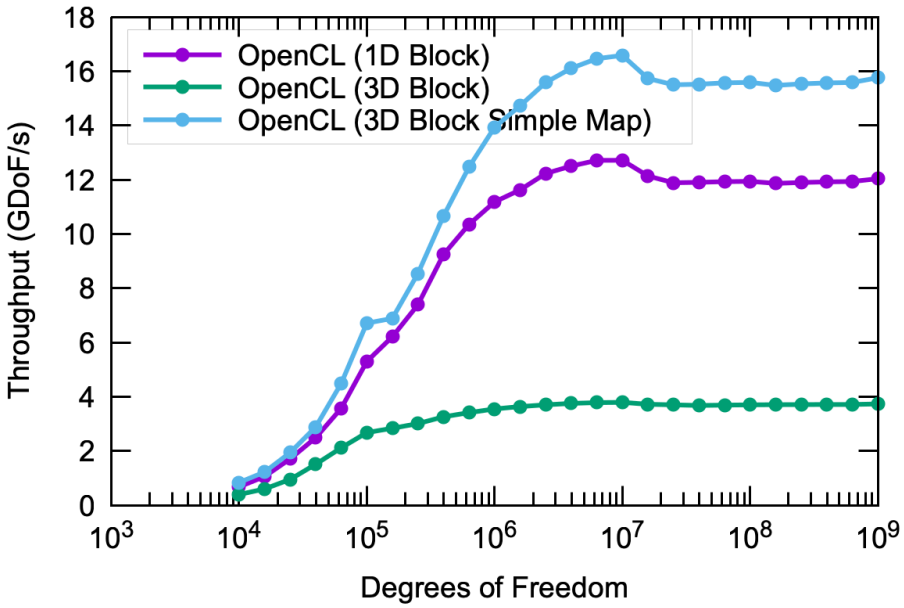

Figure 6: BK1 performance for templated kernel, $q = 4$, Intel Max Series 1550 GPU (Ponte Vecchio).

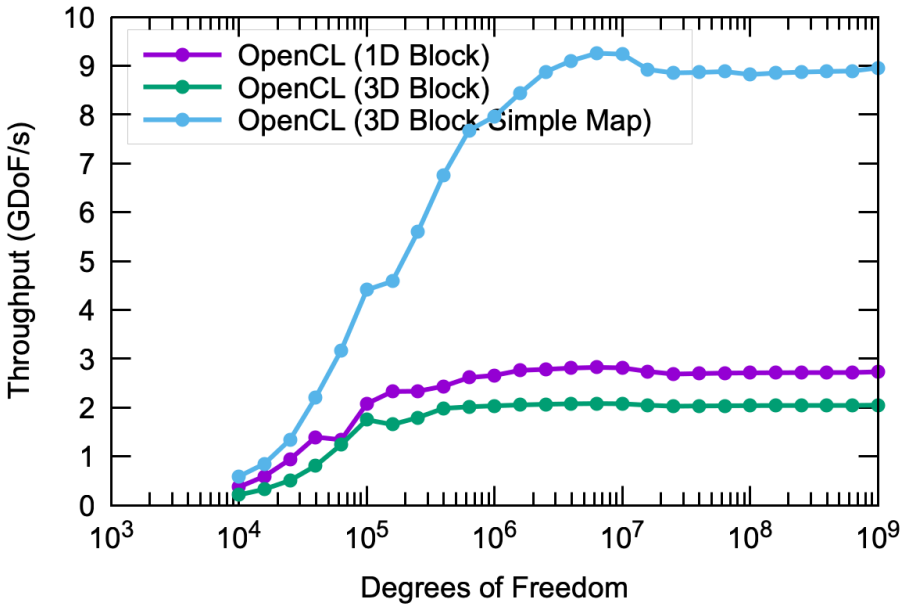

Figure 7: BK1 performance for variable kernel, $q = 4$, Intel Max Series 1550 GPU (Ponte Vecchio).

2.3 OpenMP Offloading

Since its introduction in OpenMP 4.5, offloading has become an integral component of this directive-based parallel programming framework. Adding OpenMP directives for offloading to an existing code can be quite straightforward, assuming the right data structures and loop patterns are already in place. When using OpenMP for GPU offloading, it is important to distinguish between two categories of directives for memory movement and work sharing.

Similar to other GPU programming models, OpenMP offers fine-grained control over the hierarchical parallelism in terms of teams and threads. These are reflected by the two directives, ‘omp teams’ which replicates execution across a league of teams and ‘omp parallel’ which replicates execution across threads of a team. The two directives can also be used to program in a SPMD-like mode (also called “me” mode). Typically however, the directives are combined with work-sharing constructs like ‘distribute’ and ‘for’ which control the parallelization of (nested) loops. A very useful work-sharing construct is the ‘[teams] loop‘ directive, which provides a descriptive form of parallelism in contrast to other prescriptive directives. With the ‘target [teams] loop’ directive, the precise distribution of the loop iteration space across teams and threads is left to the compiler. As explained by Deakin and Mattson (2023), this kind of descriptiveness can provide superior performance portability,

assuming the compiler succeeds to auto-parallelize the loop well.

Listing 1: The ‘omp teams loop’ directive

```
#pragma omp target \
    map(to: basis0[:nm0*nq0], basis1[:nm1*nq1], basis2[:nm2*nq2]) \
    map(to: in[:nelmt*nm0*nm1*nm2], JxW[:nelmt*nq0*nq1*nq2]) \
    map(from: out[:nelmt*nm0*nm1*nm2])
#pragma omp teams loop
for(size_t e = 0; e < nelmt; ++e) {
    /* ... sum factorization ... */
}
```

Listing 1 shows an excerpt of from the BK1 algorithm, parallelized using using the `omp teams loop` directive. The loop shown here is the outer loop across elements of the E-vector in CEED terminology. First a ‘target’ region is opened, instructing the compiler to generate device code for the scope of the upcoming for-loop. The ‘map’ clauses are used to communicate the array sizes and exert control over the direction of memory movement to the minimum necessary. Finally, the ‘teams loop‘ directive is used to divide the calculation of elements across threads in a concurrent fashion. For this particular kernel the simpler ‘omp loop’ directive (without ‘teams‘) would also be sufficient, but instead we wanted to emphasize the assigment of elements across teams first.

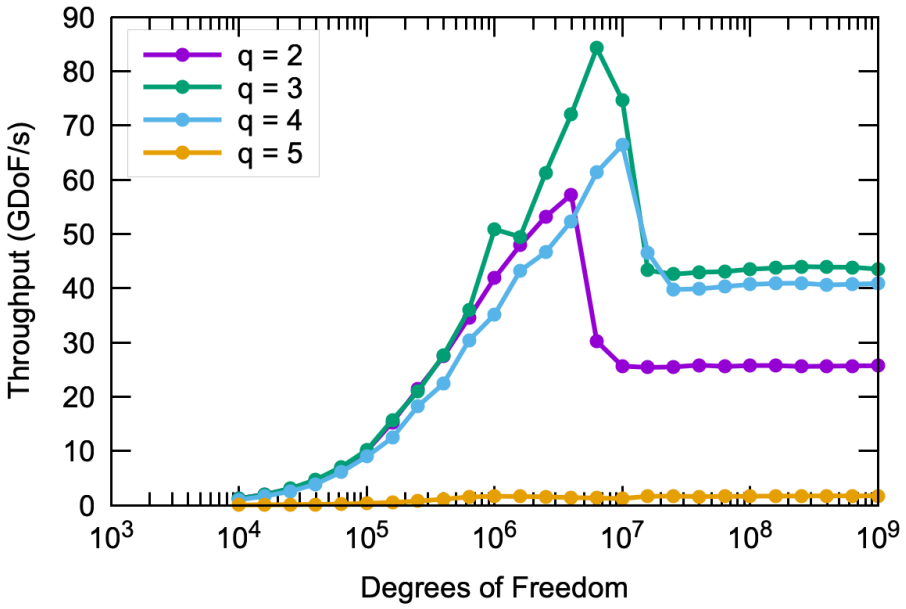

Figure 8: BK1 using OpenMP, Intel Max Series 1550 GPU (Ponte Vecchio).

2.4 Tiny Tensor Compiler

The Tiny Tensor Compiler (TinyTC) is an open-source tensor compiler developed by Intel for efficient execution of tensor computations on CPUs and GPUs. TinyTC compiles programs written in a domain-specific tensor language into OpenCL-C or SPIR-V, supporting runtime environments such as OpenCL, Level Zero, and SYCL. C and C++ APIs are available. TinyTC assumes a batched execution model, where each kernel is executed by a work-group with concurrent work-items, and the compiler maps tensor operations to efficient GPU instructions, including cooperative GEMMs and subgroup vectorization. Further details of the execution model and available instructions can be found in the online documentation.[4]

Domain-specific languages (DSLs) provide abstraction and performance portability by expressing operations in a form closer to the mathematical problem rather than low-level loops. In recent years, DSLs such as FreeFem++ or FEniCS UML allow variational forms to be compiled into optimized C or LLVM IR. Similarly, TinyTC aims to represent tensor operations, representing multi-dimensional contractions, fusions, and GEMMs in a dedicated tensor language, but remains unaware of finite element specifics.

Our first test of TinyTC aims to accelerate the finite element mass matrix evaluation (BK1) using sum factorization. As a first demonstration of tensor language we present the BK1 kernel using an algorithm presented by Świrydowicz et al. (2019), see Listing 2, where each tensor product is implemented using the loop over GEMM approach. For simplicity the implementation uses four temporary arrays, `wsp0`-`wsp4`, allocated in group local storage.

Listing 2: BK1 using Tensor IR

```
func @sum_factorization(%basis0: memref<f32x3x4>,
                        %basis1: memref<f32x3x4>,
                        %basis2: memref<f32x3x4>,
                        %JxW:    memref<f32x4x4x4x?>,
                        %in:     memref<f32x3x3x3x?>,
                        %out:    memref<f32x3x3x3x?>) {

    %gid = group_id

    %wsp0 = alloca -> memref<f32x3x3x3>; // Reserve temporary memory
    %wsp1 = alloca -> memref<f32x3x3x4>; // Reserve temporary memory
    %wsp2 = alloca -> memref<f32x3x4x4>; // Reserve temporary memory
    %wsp3 = alloca -> memref<f32x4x4x4>; // Reserve temporary memory
    %wsp4 = alloca -> memref<f32x4x4x4>;

    %J_e = subview %JxW[:,:,:,%gid] : memref<f32x4x4x4x?>;
    %in_e = subview %in[:,:,:,%gid] : memref<f32x3x3x3x?>;
```

[4] https://intel.github.io/tiny-tensor-compiler/manual/tensor-ir.html, accessed on 25/08/2028

```
    %out_e = subview %out[:,:,:,%gid] : memref<f32x3x3x3x?>;

    for %j1 = 0, 3 { ; Batch of GEMMs
        %tmp = subview %in_e[:,%j1,:] : memref<f32x3x3x3>
        %res = subview %wsp1[:,%j1,:] : memref<f32x3x3x4>
        gemm.n.n 1.0, %tmp, %basis0, 0.0, %res :
            f32, memref<f32x3x3,strided<1,9>>, memref<f32x3x4>, f32, memref<f32x3x4,strided<1,9>>;
    }

    for %j2 = 0, 4 { ; Batch of GEMMs
        %tmp = subview %wsp1[:,:,%j2] : memref<f32x3x3x4>
        %res = subview %wsp2[:,%j2,:] : memref<f32x3x4x4>
        gemm.n.n 1.0, %tmp, %basis1, 0.0, %res :
            f32, memref<f32x3x3>, memref<f32x3x4>, f32, memref<f32x3x4,strided<1,12>>
    }

    for %j3 = 0, 4 { ; Batch of GEMV
        for %p3 = 0, 4 {
            %tmp = subview %wsp2[:,%p3,%j3] : memref<f32x3x4x4>
            %res = subview %wsp3[:,%j3,%p3] : memref<f32x4x4x4>
            gemv.t 1.0, %basis2, %tmp, 0.0, %res :
                f32, memref<f32x3x4>, memref<f32x3>, f32, memref<f32x4>
        }
    }

    %flat1 = fuse %J_e[0,2] : memref<f32x4x4x4>
    %flat2 = fuse %wsp3[0,2] : memref<f32x4x4x4>
    %flat3 = fuse %wsp4[0,2] : memref<f32x4x4x4>

    hadamard 1.0, %flat1, %flat2, 0.0, %flat3 :
        f32, memref<f32x64>, memref<f32x64>, f32, memref<f32x64>

    for %q6 = 0, 4 {                              ; Step 6
        for %p6 = 0, 4 {
            %tmp = subview %wsp4[:,%q6,%p6] : memref<f32x4x4x4>
            %res = subview %wsp2[:,%p6,%q6] : memref<f32x3x4x4>
            gemv.n 1.0, %basis2, %tmp, 0.0, %res :
                f32, memref<f32x3x4>, memref<f32x4>, f32, memref<f32x3>
        }
    }

    for %p7 = 0, 4 {
        %tmp = subview %wsp2[:,%p7,:] : memref<f32x3x4x4>
        %res = subview %wsp1[:,:,%p7] : memref<f32x3x3x4>
        gemm.n.t 1.0, %tmp, %basis1, 0.0, %res :
            f32, memref<f32x3x4,strided<1,12>>,memref<f32x3x4>, f32, memref<f32x3x3>
    }

    for %j8 = 0, 3 {
        %tmp = subview %wsp1[:,%j8,:] : memref<f32x3x3x4>
        %res = subview %out_e[:,%j8,:] : memref<f32x3x3x3>
        gemm.n.t 1.0, %tmp, %basis0, 0.0, %res :
            f32, memref<f32x3x4,strided<1,9>>, memref<f32x3x4>, f32, memref<f32x3x3,strided<1,9>>
    }

}
```

As the `alloca` instruction reserves temporary memory in shared local memory, TinyTC automatically inserts thread barriers between each of the six GEMM passes to ensure functional correctness. The kernel shown above achieved 10 GDoF/s on the Intel Ponte Vecchio GPU, roughly one quarter of the performance we observed using OpenMP. One of several optimizations possible is fusing dimensions, to obtain larger GEMMs with a more favourable operation count. For instance in step 1 (the `j1`-loop), the loop over GEMMs can be rewritten as a single GEMM by fusing dimensions (Listing 3).

Listing 3: BK1 using Tensor IR

```
%tmp1 = fuse %in_e[0,1] : memref<f32x9x3>
%res1 = fuse %wsp1[0,1] : memref<f32x9x4,local>
gemm.n.n %c1, %tmp1, %basis0, %c0, %res1
```

Other GEMM steps can be fused in a similar manner, however this only delivered a few percent improvement. Upon consultation with the TinyTC author it became apparent a different strategy would be needed to obtain performant execution and optimal SIMD utilization of the PVC architecture. In collaboration with Intel, we are currently investigating the application of index fusion, batch vectorization and memory layout transformations to guarantee optimal register and SIMD usage.

Despite these initial challenges, the use of a specialized tensor DSL offers numerous potential benefits including,

- Readability and maintenance: kernels are expressed in a high-level tensor operations rather than nested loops
- Hardware portability: TinyTC maps tensor operations to different architectures without rewriting kernels. Currently, this is portability is limited to Intel devices.
- Runtime specialization: kernels written in IR are ingested as strings for just-in-time compilation, allowing constant loop bounds for optimal looping and GEMM optimizations
- Optimized memory and SIMD usage: temporary arrays are automatically placed in shared local memory; cooperative matrix instructions can reduce synchronization overhead.

3 Streaming Kernels

Iterative linear system solvers make heavy use of low arithmetic intensity operations, so called streaming operations. When matrix-free high-order FEM schemes are combined with iterative solvers. As shown previously by Kronbichler et al. (2023) on the case of the conjugate gradient method, above a certain problem size the streaming operations contribute a significant chunk of the solver run-time.

Chalmers and Warburton (2020) proposed a suite of streaming kernels to focus on the distinct operations requite by iterative solvers. As an initial step toward the performance characterization and modelling of streaming operations, we have implemented the streaming kernels as a portable Fortran program using OpenMP for parallel CPU and GPU execution. We have also implemented "backends" using other programming solutions, including C++ array libraries for linear algebra such as Eigen[5] and/or common linear algebra libraries for Fortran and C such as BLAS and BLIS [6]. In the remaining months of the work package we also plan to expand the backends to cover Kokkos and C++ standard parallelism for better cross-platform assesment.

Figures 9 and 10 show preliminary bandwidth scaling plots with respect to number of threads for the streaming kernels on two recent CPU architectures: an Nvidia Grace CPU (part of the Grace Hopper Superchip) and an Intel Sapphire Rapids (Intel Xeon Platinum 8480+) CPU. On Grace, the benchmark driver was compiled using 'gfortran' from GCC version 11.4.0. On Intel Sapphire Rapids (SPR), the benchmark driver was compiled using 'ifx' from Intel OneAPI version 2024.2.1. For both runs we enabled '-O3', host-machine specific microarchitectural optimizations ('-mcpu=native', '-xHOST') and OpenMP. To stress the main memory bus, the vector length was set to four-times the L3 cache size (117 MB in Nvidia Grace, 105 MB on Sapphire Rapids). In both cases we also employed OpenMP thread placement and pinning. As the Sapphire Rapids Xeon server was a dual-socket system, we limit the thread placement to one socket (56 cores). Further technical details of the CPUs can be found in Table 2.

On figure 9 we observe the expected saturating behavior. The Grace CPU achieves a throughput of over 300 GB/s over all kernels, exceeding 50% of the peak memory bandwidth (567 GB/s). The BS3 kernel reached almost 450 GB/s, a value close to the maximum memory bandwidth of 467 GB/s measured by Laukemann et al. (2024). In case of BS2 and BS5 the saturated regime was reached with only 20 out 72 cores.

On figure 10 we observe similar saturating behavior from the Intel Sapphire Rapids, with bandwidths in the range of 250–350 GB/s, close to the peak bandwidth. A slightly slower rate can be observed for the BS1 kernel, including a step increase when using more than 46 cores, warranting more investigation. This be-

[5] Eigen, https://eigen.tuxfamily.org/
[6] BLIS, https://github.com/flame/blis

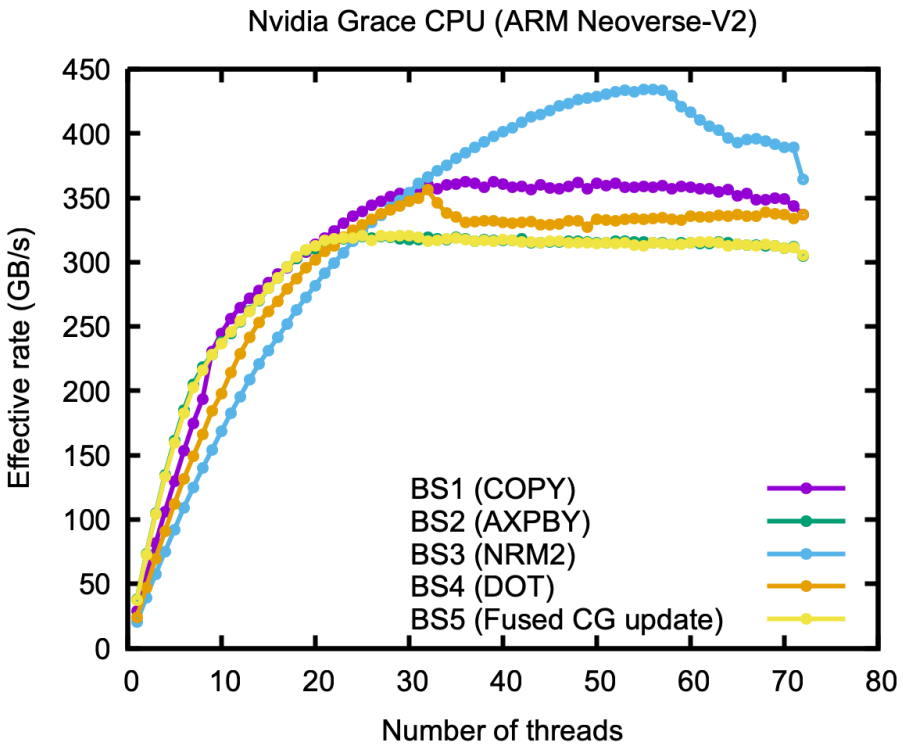

Figure 9: Streaming kernel results on Nvidia Grace for vectors of size N = 61341696 (468 MB).

havior appears to be linked to the write-allocation evasion mechanism of modern Intel CPUs (Hager, 2021).

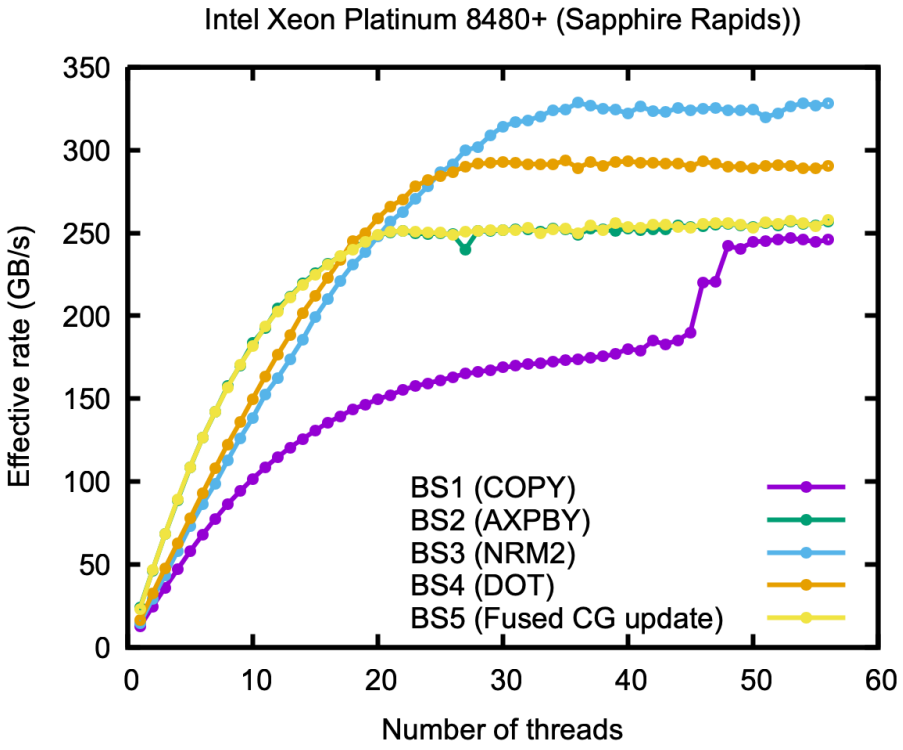

Figure 10: Streaming kernel results on Intel Sapphire Rapids for vectors of size N = 55050240 (420 MB).

Overall, our initial results confirm that the proposed streaming kernels are able to expose memory bandwidth behavior consistently across different architectures, demonstrating their usefulness as a lightweight proxy for iterative solver workloads. The observed saturation patterns highlight the importance of systematic performance characterization. In the next stage, we aim to extend the benchmarking to a broader set of CPUs and GPUs and integrate the results into predictive performance models for matrix-free iterative solvers.

4 Energy Efficiency

The shift to heterogeneous computing with accelerators is driven not only by the demand for higher performance but also by the increasing power and thermal constraints of modern semiconductor technology. These pressures have led to hard-

ware specialization for more energy-efficient computing, as reflected in the Green500 list, where GPU-accelerated systems dominate. As shown by Cielo et al. (2025) in a case study of an astrophysical simulation, accelerators can deliver a 5–10× improvement in energy efficiency (flops per watt) over CPUs. This advantage stems from their throughput-oriented parallel design, which contrasts with the latency-focused nature of traditional CPUs.

Maximizing these benefits requires energy-aware practices from both HPC practitioners and application developers. Since power and energy usage vary across devices, workloads, and runtime conditions, precise modeling is difficult, making empirical measurement essential. System-level monitoring frameworks such as Intel's RAPL, AMD uProf, NVIDIA's NVML, or Arm's Streamline provide access to hardware counters, while job- and process-level tools such as the Energy Aware Runtime, perf, likwid-powermeter, and PAPI enable finer-grained insights. For kernel-level breakdowns, vendor tools like NVIDIA Nsight or Intel VTune are available, but they often involve heavy instrumentation and vendor lock-in. In practice, lightweight and portable solutions remain highly attractive.

To this end, Cielo et al. (2025) proposed a process-based approach in which applications call an external "energy-meter" script during execution. The script samples instantaneous power through utilities such as 'nvidia-smi', 'xpu-smi', or 'perf', and integrates the readings to estimate total energy consumption. A key advantage is that programmers can call the meter only around the sections of interest, thereby excluding initialization or I/O phases and focusing on the algorithmic kernels. The method is portable, simple to maintain, and adaptable to a variety of devices or custom meters.

In the remaining period of this work package, we plan to adopt this strategy to measure the energy usage of the core algorithmic kernels and gain an overview of their power behavior. Although we have not yet carried out such experiments, we take inspiration from a recent presentation by S. Cielo (LRZ), who demonstrated the benefits of these techniques, and we intend to implement them as part of our future performance and energy-efficiency evaluations.

5 Algorithmic development and next steps

The two main ongoing efforts in terms of dealii-X project are the integration of the optimized kernels into the deal.II finite element library. As has been described in the report for the deliverable D1.3, several improvements were integrated into the deal.II library release 9.7, which was released on July 22, 2025.

An important future step is to enhance the algorithms by using matrix instructions (e.g., tensor cores). The abstractions described above will be developed further to assess the various options and performance possibilities. One step is to work on TinyTC kernel optimization using fusion and cross-element vectorization, picking up what worked well for CPUs in Kronbichler and Kormann (2019) and adapting the methods to GPUs. Especially on Intel GPUs, we believe that further advances can be made.

The second ongoing effort is to extend the methods from the pure operator evaluation of the BK1, BK3 and BK5 kernels to the full solver chain available in deal.II, including

- preconditioned conjugate gradient and GMRES solvers,
- advanced multigrid solvers, which include polynomial multigrid and geometric multigrid, as well as
- combination of the solvers with other activities in dealii-X, such as the advanced algebraic multigrid methods and incomplete factorizations developed in PSCToolkit and sparse direct solvers of MUMPS.

6 Hardware systems

The experiments in the previous sections are based on the following hardware.

Table 1: GPUs used for evaluation

Feature	NVIDIA Grace Hopper GPU (H100)	Intel Ponte Vecchio (GPU Max)
TDP	900 W (CPU + GPU + memory)	450 W (capped)
SM/EU count		128
Memory Type	HBM3 / HBM3e	HBM2e
Base Frequency	1.1 GHz	900 MHz
Memory		128 GB
Memory Bandwidth	3–4 TB/s (HBM3)	1.2 TB/s (HBM2e)

Table 2: CPU Hardware Systems use for evaluation

Feature	Intel Sapphire Rapids	NVIDIA Grace Hopper Superchip
TDP	350 W	500 W
Core Count	56 cores	72 cores
Memory Type	DDR5-4800	LPDDR5X
Base Frequency	2.0 GHz	3.1 GHz
SIMD Width / Extension	AVX-512 (512-bit)	SVE2 (4 × 128-bit)
Last-Level Cache (L3)	105 MB	114 MB
Memory Bandwidth	307 GB/s	546 GB/s

References

Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. The deal.II finite element library: design, features, and insights. *Comput Math with Appl*, 81:407–422, 2021. DOI: 10.1016/j.camwa.2020.02.022.

Noel Chalmers and Tim Warburton. Portable high-order finite element kernels i: Streaming operations. *arXiv preprint arXiv:2009.10917*, 2020.

Salvatore Cielo, Alexander Pöppl, and Ivan Pribec. SYCL for energy-efficient numerical astrophysics: the case of DPEcho. *arXiv preprint arXiv:2508.14117*, 2025.

Tom Deakin and Timothy G. Mattson. *Programming Your GPU with OpenMP: Performance Portability for GPUs*. MIT Press, 2023.

Michel O. Deville, Paul F. Fischer, and Ernest H. Mund. *High-order methods for incompressible fluid flow*, volume 9. Cambridge University Press, 2002.

Paul Fischer, Misun Min, Thilina Rathnayake, Som Dutta, Tzanio Kolev, Veselin Dobrev, Jean-Sylvain Camier, Martin Kronbichler, Tim Warburton, Kasia Świrydowicz, and Jed Brown. Scalability of high-performance PDE solvers. *The*

International Journal of High Performance Computing Applications, 34(5):562–586, 2020. DOI: 10.1177/1094342020915762.

Georg Hager. Write-allocate evasion has finally arrived at Intel – or has it? https://blogs.fau.de/hager/archives/8997, October 2021. Accessed: 2025-08-30.

Martin Kronbichler and Katharina Kormann. A generic interface for parallel cell-based finite element operator application. *Computers & Fluids*, 63:135–147, 2012.

Martin Kronbichler and Katharina Kormann. Fast matrix-free evaluation of discontinuous Galerkin finite element operators. *ACM Transactions on Mathematical Software*, 45(3):29:1–29:40, 2019. DOI: 10.1145/3325864.

Martin Kronbichler, Dmytro Sashko, and Peter Munch. Enhancing data locality of the conjugate gradient method for high-order matrix-free finite-element implementations. *The International Journal of High Performance Computing Applications*, 37(2):61–81, 2023.

Jan Laukemann, Georg Hager, and Gerhard Wellein. Microarchitectural comparison and in-core modeling of state-of-the-art CPUs: Grace, Sapphire Rapids, and Genoa. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1405–1412. IEEE, 2024.

Kasia Świrydowicz, Noel Chalmers, Aali Karakus, and Tim Warburton. Acceleration of tensor-product operations for high-order finite element methods. *International Journal of High Performance Computing Applications*, 33(4):735–757, 2019.