International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

Special Issue Paper

MAGMA: Enabling exascale performance with accelerated BLAS and LAPACK for diverse GPU architectures

The International Journal of High Performance Computing Applications
2024, Vol. 38(5) 468–490

DOI: 10.1177/10943420241261960
journals.sagepub.com/home/hpc
Sage

Ahmad Abdelfattah[1], Natalie Beams[1], Robert Carson[2], Pieter Ghysels[3], Tzanio Kolev[2], Thomas Stitt[2], Arturo Vargas[2], Stanimire Tomov[1] and Jack Dongarra[1]

Abstract

MAGMA (Matrix Algebra for GPU and Multicore Architectures) is a pivotal open-source library in the landscape of GPU-enabled dense and sparse linear algebra computations. With a repertoire of approximately 750 numerical routines across four precisions, MAGMA is deeply ingrained in the DOE software stack, playing a crucial role in high-performance computing. Notable projects such as ExaConstit, HiOP, MARBL, and STRUMPACK, among others, directly harness the capabilities of MAGMA. In addition, the MAGMA development team has been acknowledged multiple times for contributing to the vendors' numerical software stacks. Looking back over the time of the Exascale Computing Project (ECP), we highlight how MAGMA has adapted to recent changes in modern HPC systems, especially the growing gap between CPU and GPU compute capabilities, as well as the introduction of low precision arithmetic in modern GPUs. We also describe MAGMA's direct impact on several ECP projects. Maintaining portable performance across NVIDIA and AMD GPUs, and with current efforts toward supporting Intel GPUs, MAGMA ensures its adaptability and relevance in the ever-evolving landscape of GPU architectures.

Keywords

The MAGMA library, numerical linear algebra, GPU computing, performance portability

1. Historical perspective and introduction

The increasing parallel compute power of modern Graphics Processing Units (GPUs) has drawn attention from the High Performance Computing (HPC) community to exploit GPUs in general purpose computing. Early efforts such as Brook for GPUs by Buck et al. (2004) proposed data-parallel constructs, along with a compiler and a runtime system, that allow using GPUs as a streaming coprocessor. Such efforts sparked the use of GPUs in general purpose computing (GPGPU), and evolved into more mature and versatile programming models and runtime systems that are developed and maintained by the hardware vendors. As an example, NVIDIA's Compute Unified Device Architecture (CUDA, 2007) framework is the default programming model for NVIDIA GPUs since early 2007. In 2016, AMD introduced the Radeon Open Compute (ROCm, 2024) platform and the Heterogeneous-compute Interface for Portability (HIP, 2024) runtime for its GPUs; the HIP framework also includes a portability layer across NVIDIA and AMD GPUs. Finally, Intel's high-end GPUs have introduced the latest programming model to the GPGPU domain, which can be leveraged through (SYCL, 2024), an open-source standard for programming heterogeneous devices. Announced in 2019, Intel's Data Parallel C++ (DPC++) compiler implements the SYCL standard and includes several additional extensions Reinders et al. (2021). These three vendors dominate the market for general purpose GPUs. As of November 2023, the TOP500 list, by Strohmaier et al. (2023), shows that nine of the ten most powerful supercomputers are equipped with

[1]Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
[2]Lawrence Livermore National Laboratory, Livermore, CA, USA
[3]Lawrence Berkeley National Laboratory, Berkeley, CA, USA

Corresponding author:
Ahmad Abdelfattah, Innovative Computing Laboratory, University of Tennessee, Suite 203 Claxton, 1122 Volunteer Blvd, Knoxville, TN 37996, USA.
Email: ahmad@icl.utk.edu

GPUs from NVIDIA, AMD, or Intel. In addition to the programming models and runtime systems, vendors often provide standard numerical libraries, such as NVIDIA's (cuBLAS, 2024), AMD's (rocBLAS, 2024), and Intel's (MKL, 2024) libraries.

The United States Department of Energy's Exascale Computing Initiative (ECI), a partnership between two DOE organizations, the Office of Science and the National Nuclear Security Administration, was formed in 2016 to accelerate research, development, acquisition, and deployment projects to deliver exascale computing capability to the DOE laboratories by the early to mid-2020s. There are three major components of ECI: selected program applications development, procurement of three exascale computing systems, and the Exascale Computing Project (ECP). ECP was a 7-year project, focused on delivering specific applications, software products, and outcomes on DOE computing facilities. Integration across these elements with specific hardware technologies for the manifestation of exascale systems was fundamental to the success of ECP. NVIDIA, AMD, and Intel were among the main vendors to provide the ECP hardware. In 2016, ECP applications, along with their associated exascale challenge problems and software technologies, underwent rigorous external review and selection for inclusion in ECP Alexander et al. (2020). Figure 1 illustrates the software technologies within ECP (excluding standard necessities like C/C++/Fortran/MPI/OpenMP/Spack), along with the number of ECP applications dependent on each technology, as indicated by an ECP survey of developers. Notably, LAPACK and BLAS emerge among the "most needed," i.e., having the highest number of dependents. The need for LAPACK and BLAS on GPU-enabled supercomputers underscores MAGMA's importance to the ECP effort and the imperative to implement and continue accelerating these standards for GPUs.

1.1. Motivation and project launch

MAGMA's origins predate the ECP. The MAGMA project started shortly after the first CUDA SDK was made public in 2007, with the initial MAGMA publication featuring the work by Baboulin et al. (2008). The first release, MAGMA v0.1, was introduced by Tomov et al. (2009). MAGMA pioneered a design that uses hybrid CPU-GPU algorithms, featured in the work by Tomov et al. (2010a), (2010b), for standard dense linear algebra (DLA) operations that are often found in the LAPACK library Angerson et al. (1990). GPUs are naturally suited for embarrassingly-parallel compute workloads. The majority of DLA algorithms are often dominated by the GEneral Matrix-Matrix multiplication (GEMM), whose behavior is defined by the standard (BLAS, 1980) library. Consequently, MAGMA incorporated key BLAS routines, meticulously optimized for NVIDIA GPUs. With the advent of FP64 arithmetic in GPUs, MAGMA's GEMM routine, designed by Nath et al. (2010), stood out as the fastest at the time and was subsequently integrated into cuBLAS. Given the pivotal role of BLAS in DLA, MAGMA's original design philosophy was to offload compute-bound BLAS operations like GEMM to the GPU while maintaining the inherently-serial stages of the algorithms on the CPU. However, the data movement between the CPU and the GPU would create a performance bottleneck that needed to be handled. To address this, MAGMA employed CUDA concurrent stream execution in order to overlap communication with useful computation, a technique commonly known as "lookahead" in DLA algorithms Tomov et al. (2010a). To this day, most of the high-level DLA algorithms in MAGMA are based on hybrid algorithms that are now ported to the HIP and SYCL programming models, in addition to the originally targeted CUDA platform.

1.2. Complementing the industrial software stack

In addition to high-level LAPACK algorithms, MAGMA also provides numerous high performance GPU kernels for various BLAS and LAPACK operations. Originally, this initiative stemmed from the untapped potential observed in the vendor software stacks due to either missing BLAS/LAPACK functionality or sub-optimal implementations. For instance, certain BLAS kernels in cuBLAS took inspiration from the initial MAGMA-BLAS subpackage, such

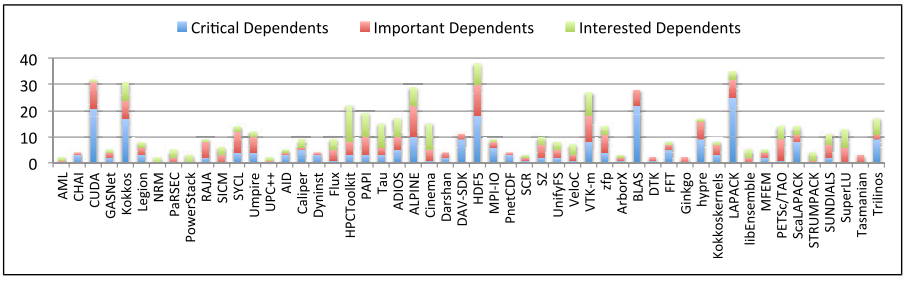

Figure 1. Number of ECP applications depending on the various ECP software technologies. Source: internal ECP application developers survey.

as the GEMM kernel, which underwent extensive design optimizations and tuning by Nath et al.(2010) and Kurzak et al. (2012). Other kernels such as the symmetric matrix-vector product (SYMV) outperformed cuBLAS, resulting in performance gains for symmetric eigenvalue decomposition, which is featured in the work by Nath et al. (2011). We would naturally expect the vendor software stack to become more optimized over time, potentially diminishing some of the unique value of MAGMA across scientific applications. However, MAGMA retains several advantages that enabled it to be a key component of the ECP, aiding several ECP applications, some of which we will discuss later.

The main purpose of this paper is to provide an overview of MAGMA, show how it continues to adapt to the architectural changes of GPU-accelerated compute nodes, and demonstrate its impact on selected ECP applications. Below are the key highlights of the paper:

1. World-class Performance. We acknowledge that MAGMA, being a research project, may not always outperform the industry in every BLAS/LAPACK operation. However, the paper shows sample performance results in which MAGMA's performance is superior to the vendor numerical software.
2. Performance Portability. MAGMA now supports different backends for major GPU vendors, namely a CUDA backend for NVIDIA GPUs, a ROCm/HIP backend for AMD GPUs developed by Brown et al. (2020), and a SYCL/DPC++ backend for Intel GPUs (which is in pre-release stage). This enables applications to build against MAGMA without the need to develop their own abstractions.
3. Critical Application Support. MAGMA provides some linear algebra functionality that is currently missing from the vendor software stack, yet is required by certain ECP applications, as we describe later in the paper.

2. Software architecture of MAGMA

Figure 2 shows an overview of MAGMA's internal structure. It requires a CPU (host) and one or more GPUs (devices) through an interconnect such as PCIe or NVLink. MAGMA also requires a CPU backend with an implementation of BLAS and LAPACK (e.g., ((OpenBLAS, 2023)), Intel (MKL, 2024), etc.). The CPU backend must provide the standard FORTRAN interfaces of BLAS and LAPACK, for which MAGMA builds high-level C APIs. For performance purposes, the CPU backend is preferred to be multithreaded (e.g., through OpenMP), although this is not required. The second requirement is the GPU programming model and runtime, namely CUDA for NVIDIA GPUs, ROCm/HIP for AMD GPUs, and SYCL/DPC++ for Intel GPUs. MAGMA builds robust abstractions for auxiliary functionalities such as (multi-)device management, type-safe memory allocations, stream and event management, and various synchronization functions. In addition, MAGMA builds high-level wrappers for some BLAS routines from the vendor library, especially those that are highly optimized, such as matrix multiplication. Expert users who interact with both MAGMA and the vendor software are provided with APIs to extract backend-specific information, such as the CUDA/HIP/SYCL stream or queue, and the handles of the BLAS libraries. These abstractions, coupled with the integration of standard BLAS and LAPACK APIs, facilitated the adoption of MAGMA within the ECP. This shields applications from the need to rely on vendor-provided programming models or seamlessly enables interaction with them when necessary. While memory management can be handled through MAGMA's interface, users can still pass raw device pointers to MAGMA routines, irrespective of their allocation method. All the above flexibility enables MAGMA to seamlessly integrate with other portability layers as well, including directive-based approaches like OpenMP offload and OpenACC, portability libraries such as Kokkos and RAJA, and domain-specific frameworks like AMReX. In general, we identify the following categories of linear algebra functionality in MAGMA:

1. BLAS. These are GPU kernels that provide either competitive or superior performance to the vendor BLAS library. The kernels were originally implemented in CUDA, and then translated, with architecture-specific changes, into HIP or SYCL. For performance-critical kernels (e.g., GEMM), MAGMA has a decision layer that selects between its own implementation and the vendor BLAS library, based on the input sizes and the call configuration (e.g., matrix transposition), so that the best performing version is invoked.
2. LAPACK. Dense matrix factorizations, linear solvers, eigensolvers, and SVD solvers represent most of the GPU-accelerated functionality in the dense component of MAGMA. Most of these implementations utilize hybrid CPU-GPU algorithms, and some of them are not supported by the vendor, such as the QR factorization with column-pivoting. As we mention later in the paper, MAGMA started providing GPU-only versions of selected LAPACK algorithms for better performance and energy efficiency on some system configurations.
3. Batch BLAS/LAPACK. The batch linear algebra functionality addresses workloads that involve applying the same algorithm to many relatively small matrices that are independent from each other. As we show later, hybrid algorithms become inefficient for this type of workload, so all the batch BLAS/

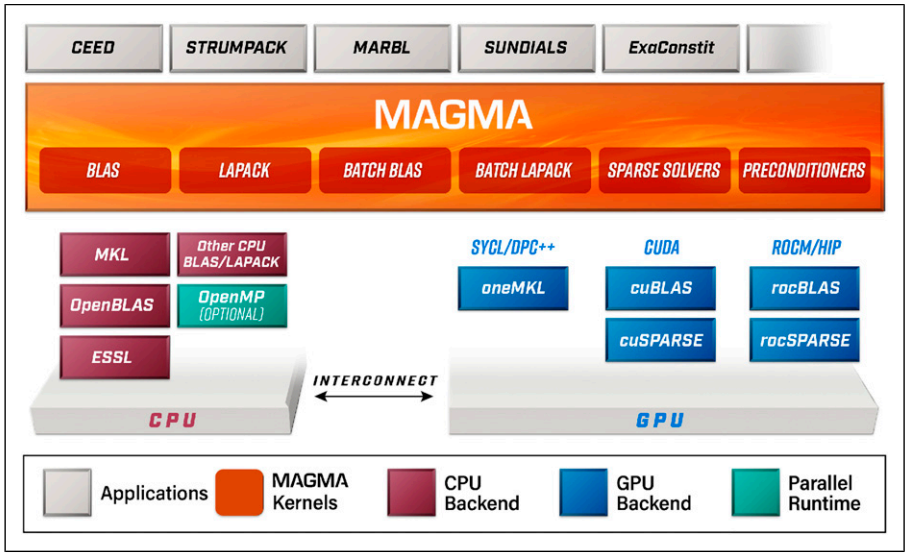

Figure 2. A high-level overview of the MAGMA library.

LAPACK operations work exclusively on the device memory.

4. Sparse iterative solvers and preconditioners. For sparse linear algebra, MAGMA implements several iterative solvers such as GMRES originally developed by Saad and Schultz (1986), CG on GPUs by Anzt et al. (2016a), and BiCG and BiCGStab also by Anzt et al.(2015). This requires optimized sparse matrix-vector product kernels (SpMV) for different sparse formats, such as the work by Anzt et al. (2014). Several preconditioners are also implemented, such as ParILU by Chow et al. (2015), ISAI by Anzt et al. (2016b), and Block Jacobi by Anzt et al. (2017). However, this paper focuses on the dense component of MAGMA, since it is the component used the most by ECP applications.

The wide range of numerical linear algebra functionality in MAGMA is appealing to many ECP applications that require portable high performance across current and future DOE systems accelerated by GPUs from NVIDIA (Perlmutter and Polaris), AMD (Frontier and El Capitan), and Intel (Aurora). In this paper we highlight few of these applications—STRUMPACK, MARBL, and ExaConstit—as well as MAGMA's impact in the CEED co-design center.

3. Standard dense matrix algorithms

MAGMA implements LAPACK algorithms optimized for GPUs. The algorithms in LAPACK are blocked, meaning they iterate over a loop, factorizing a panel (block of columns of the matrix), followed by trailing matrix updates. This process is illustrated in Figure 3 (top) for the LU factorization. Blocked algorithms can be implemented naturally as calls to level-3 BLAS. By expressing these algorithms as level-3 BLAS computations, MAGMA achieves significantly higher efficiency on current architectures compared to if the factorization proceeded one column at a time followed by matrix updates, which would require level-2 BLAS computations.

In addition to the level-3 BLAS structure, which enhances parallelism and efficient GPU utilization, MAGMA has pioneered hybrid algorithmic designs enabling the simultaneous utilization of both GPUs and CPUs. This is achieved by strategically scheduling the execution of critical algorithmic segments, such as panels, on the CPU, while delegating highly parallel matrix updates (e.g., GEMMs) to the GPU. This approach is illustrated in Figure 3 (top).

This design, tailored for parallelism and efficient CPU + GPU use, has proven highly successful across the entire MAGMA library, encompassing the main linear solvers, originally developed by Tomov et al. (2010), as well as eigenvalue problems and SVD by Tomov et al. (2010b). Keeping the difficult-to-parallelize panel factorization tasks on the CPU enables rapid code development, while utilization of GPUs for Level 3 BLAS operations, such as GEMMs, leverages their high efficiency in harnessing the full power of the GPU. The hybrid algorithms in MAGMA also allow for the overlapping of small computational tasks on the CPU with large matrix updates performed on the GPU. This overlap is illustrated by the execution trace in Figure 3 (bottom) for the LU factorization, with a closer look at a single iteration in Figure 4. Note that in this case, the work on the CPU along with the required CPU-GPU communications are totally overlapped with the GEMM

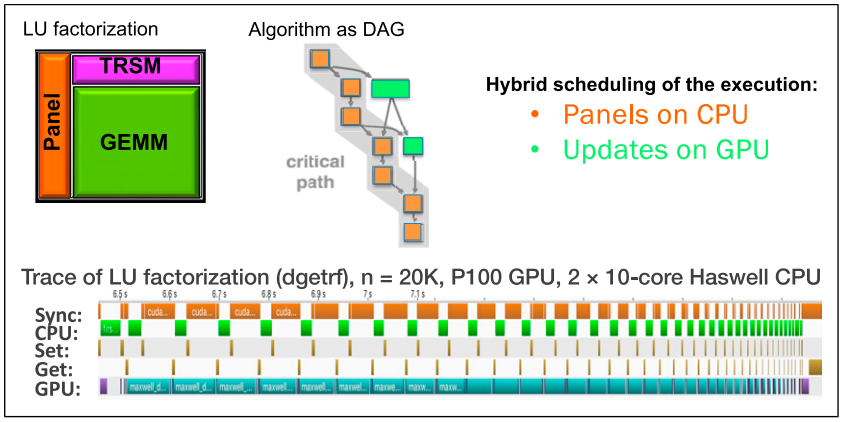

Figure 3. Hybrid CPU + GPU algorithm design and a typical execution trace overlapping CPU with GPU computations.

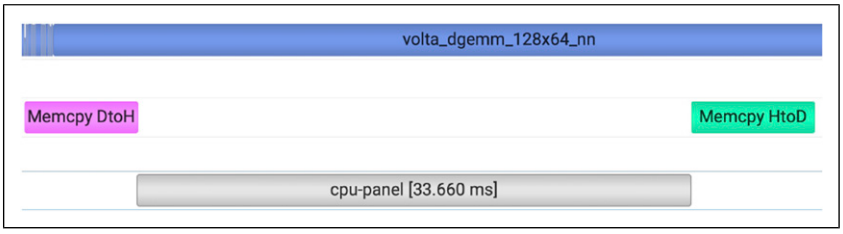

Figure 4. A partial trace of MAGMA's hybrid LU factorization, in double precision, of a square matrix (N = 10240), on Tesla V100-SXM2 GPU hosted by an Intel Xeon CPU E5-2698 v4 (Broadwell), running MKL-2023.0.2 using 40 threads.

computations on the GPU. Here, the use of the lookahead technique enables us to start the factorization of the next panel on the CPU, while the remaining trailing matrix is being updated on the GPU Tomov et al. (2010). One drawback of this approach arises when full overlap is not achievable, resulting in the GPU waiting for the panel factorization, which is discussed further below.

4. Challenges for hybrid algorithms on modern HPC systems

While hybrid algorithms exploit all the computational resources of the underlying system, they require certain balance between the CPU and the GPU. In a typical lookahead implementation, the best performance is achieved when the CPU activity, including communication, is completely hidden by the GPU's compute-intensive tasks, as shown in Figure 4. However, depending on the hardware configuration and the CPU software stack, hybrid algorithms can become vulnerable to bottlenecks on the CPU execution timeline. As an example, Figure 5 shows a similar trace to Figure 4, but using a faster GPU (A100) that is hosted by a CPU of similar compute power. Despite the slightly faster CPU execution time in Figure 5, the GPU encounters several idle times, due to the highly optimized GEMM kernels that are now hardware-accelerated by the Tensor Cores. This behavior results in performance regression for hybrid algorithms, especially if there is a relatively large gap in the compute power between the CPU and the GPU. Ideally, hybrid algorithms require a high-end CPU running a highly optimized LAPACK implementation.

In order to address the issue of hybrid algorithms underperforming on modern HPC systems, MAGMA started to provide GPU-only LAPACK routines, which are executed entirely on the GPU, while the CPU only manages kernel launches. In the MAGMA notation, these implementations are called "GPU-native routines", and early efforts in that direction were proven successful for the Cholesky factorization by Haidar et al. (2017) and the LU factorization by Abdelfattah et al. (2018b). Figure 6 shows performance results for the dense LU factorization on a V100 GPU hosted by an Intel Broadwell CPU, where the hybrid design achieves the best asymptotic performance. For small matrix sizes, native algorithms, especially cuSOLVER, provide a superior performance, due to the lack of enough computational workload on the trailing matrix updates, which results in partial overlap between the CPU and the GPU activities.

A different behavior is observed in Figure 7, where MAGMA's native routine is faster than the hybrid algorithm

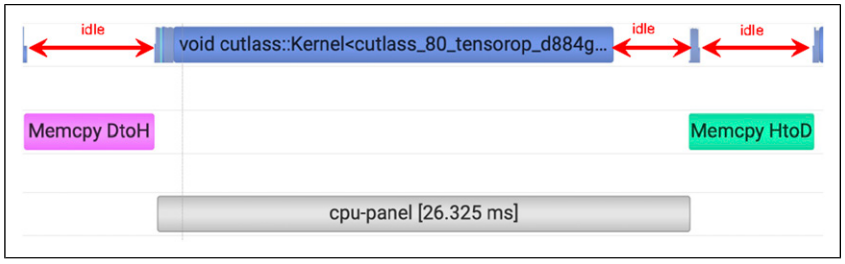

Figure 5. A partial trace of an LU factorization, in double precision, of a square matrix ($N = 10240$) on Tesla A100-SXM4 GPU hosted by an AMD EPYC 7742 64-Core Processor, running MKL-2023.0.2 using 128 threads.

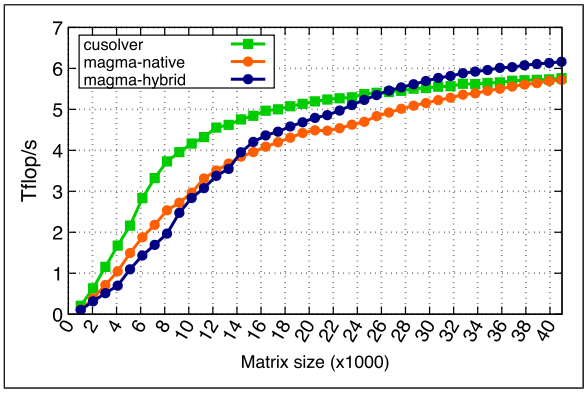

Figure 6. Performance of the dense LU factorization in double precision. Results are shown on Tesla V100-SXM2 GPU hosted by an Intel Xeon CPU E5-2698 v4 (Broadwell). MAGMA is compiled using CUDA-12.1 and MKL-2023.0.2 as the CPU backend.

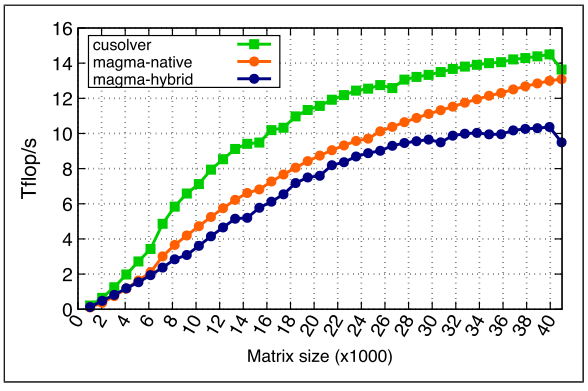

Figure 7. Performance of the dense LU factorization in double precision. Results are shown on Tesla A100-SXM4 GPU hosted by an AMD EPYC 7742 64-Core Processor. MAGMA is compiled using CUDA-12.1 and MKL-2023.0.2 as the CPU backend.

due to the GPU idle times shown earlier in Figure 5. In addition, while MAGMA's native routine used to outperform cuSOLVER as shown in the work by Abdelfattah et al. (2018b), the vendor's implementation has undergone significant improvements and optimizations since then, thus

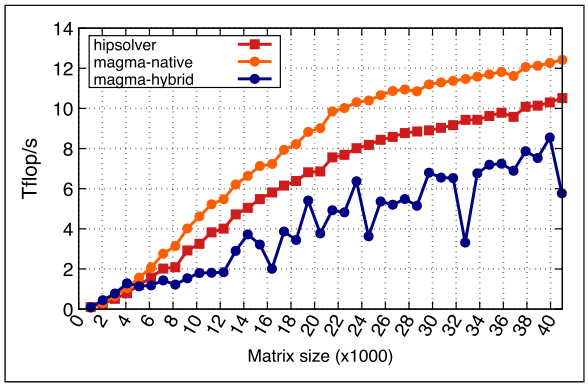

Figure 8. Performance of the dense LU factorization in double precision. Results are shown on an Instinct MI210 GPU hosted by an AMD EPYC 7413 24-Core Processor. MAGMA is compiled using ROCM-5.7 and MKL-2023.0.2 as the CPU backend.

achieving the best overall performance. Finally, Figure 8 shows that the MAGMA native LU factorization on the M210 GPU is superior to both the MAGMA hybrid algorithm and the vendor's routine, reaching an asymptotic speedup of 1.18× against hipSOLVER.

5. Mixed-precision (MXP) algorithms

Mixed-precision (MXP) algorithms try to accelerate applications requiring high accuracy (e.g., FP64) by performing the dominant parts of an algorithm in lower precisions (e.g., FP32), followed by a correction step at which the desired accuracy is recovered. In this section, we highlight MAGMA MXP solvers for solving a dense linear system of equations $Ax = b$, where A is a dense matrix, b is the input right-hand side (RHS), and x is the solution vector.

5.1. MXP solvers with iterative refinement (IR)

Classical MXP linear solvers perform the dominant matrix factorization $(\mathcal{O}(n^3))$ in a lower precision, while using *Iterative Refinement (IR)* during the less expensive triangular solve $(\mathcal{O}(n^2))$. IR is a well-established technique that

iteratively improves the solution x by solving a correction equation and adding this correction to the solution vector, as contributed by Wilkinson (2023), Moler (1967), and Demmel (1997). IR involves three main steps. The first is the original factorization/solve for an initial value of x. The second step is to compute the residual ($r = b - Ax$). The third step is to solve the correction equation ($Ac = r$) in order to update the solution vector ($x \leftarrow x + c$). In the 2000s, motivated by processors equipped with FP32 speed 2× that of FP64, mixed precision iterative refinement (MXP-IR) solvers performed the LU factorization and the correction equation in FP32 and everything else in FP64, as shown in the work by Buttari et al. (2007), Baboulin et al. (2009a), and Langou et al. (2006). Since the LU factorization is dominated by rank-k updates utilizing the GEMM kernel, the performance advantage of MXP-IR algorithms over a full-precision solver is bounded by the performance advantage of the FP32 GEMM (i.e. SGEMM) over the FP64 GEMM (i.e. DGEMM) on a particular hardware.

5.2. MXP solvers with GMRES-based iterative refinement (IRGMRES)

The introduction of NVIDIA's Volta architecture sparked significant interest in utilizing reduced precisions in scientific applications. The word "reduced" here means any precision less than 32-bits, but we focus only on the IEEE 16-bit floating point format, also known as "half precision", or FP16. The interest was driven by architectural advancements in which FP16 computations are hardware-accelerated using special units known as the "tensor cores", which accelerate matrix multiplication in FP16. The V100 GPU has theoretical peak performances of 7.8 Tflop/s and 15.7 Tflop/s for FP64 and FP32, respectively, which is a natural 1:2 ratio between the two precisions. However, the same hardware has a theoretical peak performance of 125 Tflop/s for FP16 utilizing the tensor cores, which is ≈8 × and 16× faster than FP32 and FP64, respectively. In addition, the V100 also included probably the first mixed-precision GEMM hardware, with the ability to perform $C = A \times B$, where A and B are in FP16, and C is accumulated in FP32. While the inclusion of FP16 arithmetic in high-end GPUs was mainly driven by machine learning applications, numerical analysts have studied the use of FP16 in numerical MXP solvers.

In particular, replacing the direct triangular solves of the correction equation ($Ac = r$) with an iterative method, as suggested by Carson and Higham (2017) in a mixed precision context, leads to "nesting" of two iterative methods; we refer to these as "inner-outer" iterations. The latter have been studied both theoretically and computationally by Golub and Ye (1999); Saad (1993); and Simoncini and Szyld (2002), including in mixed-precision computation scenarios Baboulin et al. (2009b). Recently, Carson and Higham (2017); Carson and Higham (2018) analyzed the convergence property of a three-precision iterative refinement scheme (factorization precision, working precision, residual precision) and concluded that if the condition number of A is not too large, $\kappa_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty < 10^4$, then using FP16 for the $\mathcal{O}(n^3)$ portion (the factorization) and FP32/FP64 or FP64/FP128 as the working/residual precision, respectively, for the $\mathcal{O}(n^2)$ portion (refinement loop), one can expect to achieve forward error and backward error on the order of 10^{-8} and 10^{-16} respectively. The same study also showed that when using the generalized minimal residual (GMRES) method preconditioned by the FP16 LU factorization as the refinement procedure, the constraint on the condition number can be relaxed to be $\kappa_\infty(A) < 10^8$ when the working/residual precision is FP32/FP64 and to 10^{12} when the working/residual precision is FP64/FP128, respectively.

MAGMA offered the first practical implementation on GPUs for a three-precision MXP linear solver that uses FP16 and FP32 in order to solve $Ax = b$ up to double precision accuracy, as demonstrated by Haidar et al. (2018). While many implementations have been studied, we highlight the most numerically stable approach in this paper. First, MAGMA avoids using FP128 to compute the residual and opts to use FP64 for practical considerations. Second, instead of running a full FP16 factorization, MAGMA uses a mixed-precision factorization where FP32 is used everywhere except in the dominant rank-k updates, which uses FP16. Third, the rank-k updates are performed using a mixed-precision GEMM kernel from cuBLAS/hipBLAS, such that the multiplicands are in FP16, while the accumulation occurs in FP32, which improves the numerical stability versus an FP16 GEMM with a cast to FP32. The MAGMA implementation has been embraced by NVIDIA, and later integrated into the cuSOLVER library.

Figures 9–11 show the advantage of MXP solvers (both classical IR and IRGMRES) over a full FP64 solver on three different GPUs (NVIDIA's V100 and A100 GPUs, and AMD's MI210 GPU). On the V100 GPU (Figure 9), the classical MXP-IR solver is able to achieve an asymptotic 1.9× speedup against the baseline (full FP64 solver), which aligns with the 1:2 ratio between FP32 and FP64 performances on this GPU. The MXP-IRGMRES solver achieves 4.1× against the baseline. There are two reasons behind this speedup not being close to the 16 fold difference between FP16 and FP64 performances. The first is that the performance of the rank-k updates peaks at ≈ 45% of the theoretical peak, which affects the overall performance of the mixed-precision factorization. The second reason is that the iterative refinement loop now uses a costly GMRES solver to solve $Ac = r$ instead of a direct triangular solve. Moving to Figure 10, the performance results on the A100 GPU show no significant advantage for the classical MXP solver, which is due to the 1:1 ratio between FP32 and FP64 theoretical peak performances

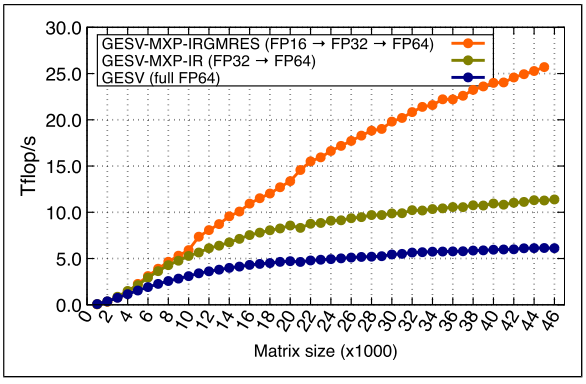

Figure 9. Performance of solving $Ax = b$ up to double precision accuracy in MAGMA. Results are shown on a Tesla V100-SXM2 GPU, running CUDA-12.1.

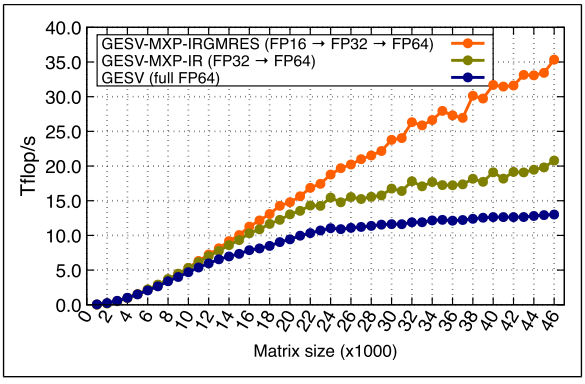

Figure 11. Performance of solving $Ax = b$ up to double precision accuracy in MAGMA. Results are shown on an Instinct MI210 GPU, running ROCM-5.7.1.

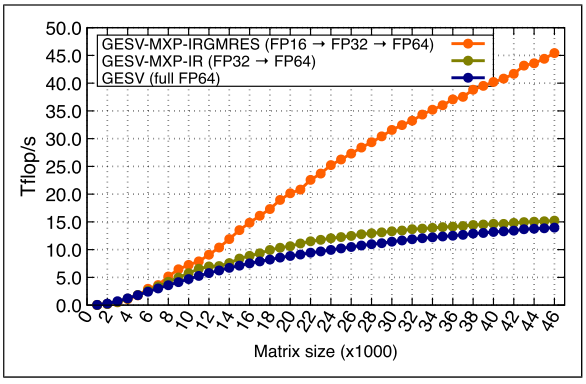

Figure 10. Performance of solving $Ax = b$ up to double precision accuracy in MAGMA. Results are shown on a Tesla A100-SXM4 GPU, running CUDA-12.1.

on this GPU (both at 19.5 Tflop/s). On the other hand, the MXP-IRGMRES solver continues to show an asymptotic 3× speedup against the baseline due to the performance advantage of FP16 on the A100 GPU. Finally, for the performance on MI210 GPU in Figure 11, while the theoretical peak performance of both FP32 and FP64 are the same (45.3 Tflop/s with matrix cores), there is a discrepancy between the baseline and the MXP-IR solver. This is due to a 40% gap between FP32 and FP64 in the performance of rank-k updates. The MXP-IRGMRES solver also maintains an asymptotic 2.7× speedup against the baseline.

6. Batch linear algebra

6.1. Uniform batch computations

The term "batch computation" or "batch linear algebra" refers to applying the same BLAS/LAPACK operation to a large number of relatively small problems having the same dimensions. Such a compute workload is different from a single matrix operation, where the matrix is often assumed to be large enough to saturate the GPU with sufficient parallel workload. Batch computations are relatively recent, and gained significant attention and momentum since 2014. This is due to advancements in many scientific domains that led to the demand of efficient batch linear algebra libraries. Examples include quantum chemistry Auer et al. (2006), sparse direct solvers Yeralan et al. (2017), astrophysics Messer et al. (2013), and signal processing Anderson et al.(2012). Batch matrix multiplication, especially at reduced precisions such as the work by Abdelfattah et al. (2019a), is a performance critical component in many machine learning frameworks. The huge demand from applications and early efforts in the MAGMA library have driven major vendors to introduce selected batch linear algebra operations into their numerical software stack.

The MAGMA library pioneered high performance implementations of batch linear algebra on GPUs. Early efforts showed that dedicated GPU kernel optimizations significantly outperformed implementations utilizing the vendor software stack inside a parallel loop using streams/queues, as shown by Haidar et al. (2015) Abdelfattah et al. (2016b). Several research efforts in MAGMA have also shown that batch linear algebra often requires multiple design strategies in order to achieve high performance for any dimensions. For example, batch computations on very small matrices, especially for sizes smaller than a warp, are considered memory bound, even for numerical algorithms requiring $\mathcal{O}(n^3)$ operations, as demonstrated by Abdelfattah et al. (2017a). Figure 12 shows two different implementations of the MAGMA batch LU factorization with partial pivoting. The first one uses a *fused kernel* that implements the entire factorization in a single execution context, thus maximizing data reuse by reading/writing each matrix exactly once from/to the global memory of the GPU. The second implementation breaks down the factorization into its standard components, following the LAPACK structure, and

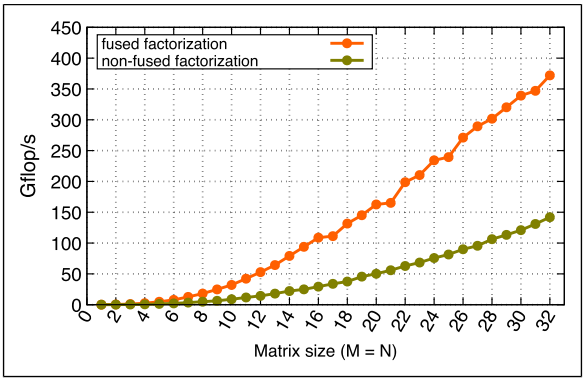

Figure 12. The performance advantage of the MAGMA fused batch LU factorization over the MAGMA non-fused implementation. Results are shown for 1000 matrices in double precision, on a Tesla A100-SXM4 GPU, using CUDA-12.1.

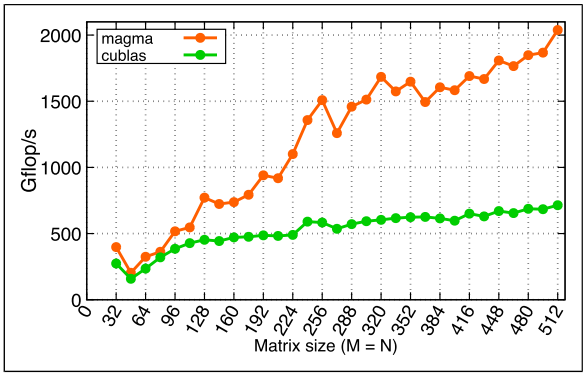

Figure 13. Performance of the batch LU factorization. Results are shown for 1000 matrices in double precision, on a Tesla A100-SXM4 GPU, using CUDA-12.1.

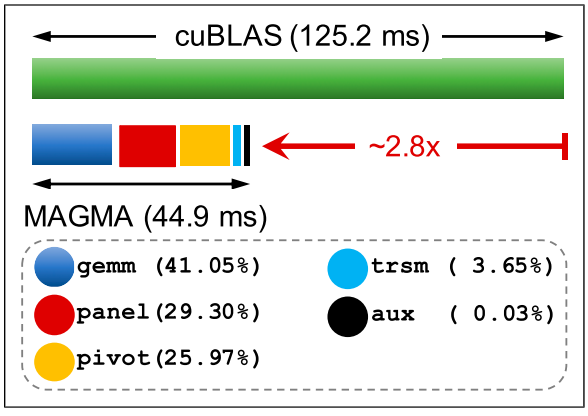

Figure 14. Breakdown of the execution time of a batch LU factorization on 1000 square matrices of size 512 in double precision. Results are shown for a Tesla A100-SXM4 GPU, using CUDA-12.1.

implements each primitive operation into a separate batch GPU kernel. The clear advantage of the first approach proves that optimizations for the memory bandwidth are more critical for very small matrix computations. Masliah et al. (2016) and Abdelfattah et al. (2018a) showed that the same approach is successful for batch matrix multiplication and batch QR factorization, respectively.

However, the performance of fused batch computations does not scale as the problem size grows, due to several factors. First, caching the entire matrix in the fast memory levels of the GPU increases the resource pressure, which leads to poor kernel occupancy (i.e. the number of resident computations on the same multiprocessor/compute unit). Second, larger matrices benefit more from the optimized level-3 BLAS routines, such as the batch GEMM kernels in the vendor libraries. Figure 13 shows the performance of the MAGMA batch LU factorization against cuBLAS for larger sizes. The primary reason for the performance advantage of MAGMA is due to the use of optimized batch level-3 BLAS, unlike cuBLAS which uses a single kernel to perform the factorization, according to our profiling results in Figure 14. In addition to the batch level-3 BLAS, it has been shown by Abdelfattah et al. (2019b) that different optimizations are necessary for the panel factorization, depending on the panel dimensions.

The same design strategy of the batch LU factorization can be applied to a QR factorization, another critical LAPACK algorithm. Due to the reliance on batch level-3 BLAS, MAGMA maintains a significant performance advantage over cuBLAS and hipBLAS for square matrices, where the trailing matrix updates are rich in batch matrix multiplication Abdelfattah et al. (2022b). However, unlike the LU factorization, which focuses mostly on square problems, the QR factorization is broadly applied to tall-skinny matrices, e.g., for solving least square problems, and even batch SVD solvers such as the work by Boukaram et al. (2018). It has been shown that applying a batch of block Householder reflectors efficiently on the GPU may require a mix of design strategies, often re-organizing the computational steps into kernels that do not exactly follow the LAPACK structure. The performance results in Figures 15 and 16 show the performance advantage of the MAGMA batch QR factorization over the vendor numerical libraries on tall-skinny problems. The relatively small shared memory on the MI210 GPU (64 KB) limits the overall occupancy of the MAGMA kernels for applying the block reflectors. This is in contrast to the A100 GPU, which provides a much larger shared memory (164 KB), thus enabling a much higher performance.

In terms of coverage, MAGMA provides a wide range of batch linear algebra functionality, including all level-3 BLAS, general and symmetric matrix-vector products, and one-sided factorizations. In addition, MAGMA is the

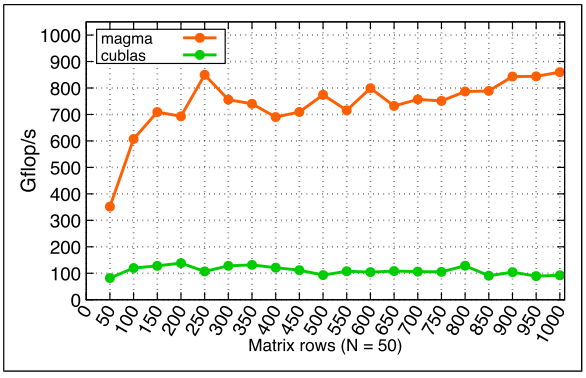

Figure 15. Performance of the batch QR factorization on tall-skinny matrices. Results are shown for 1000 matrices in double precision, on a Tesla A100-SXM4 GPU, using CUDA-12.1.

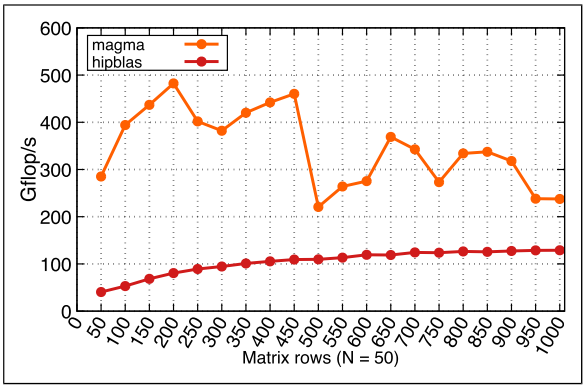

Figure 16. Performance of the batch QR factorization on tall-skinny matrices. Results are shown for 1000 matrices in double precision, on an Instinct MI210 GPU, using ROCM-5.7.1.

only library that supports fully non-uniform batch computations, which are discussed in the next section.

6.2. Non-uniform batch computations

As a natural progression to uniform batch computations, there is indeed a demand for batches of problems with different dimensions. One particular example is multi-frontal sparse direct solvers, such as STRUMPACK, which we highlight later in the paper. However, in this section, we focus on how to support non-uniform batch linear algebra on GPUs. MAGMA supports non-uniform batch linear algebra where only the dimensions of matrices differ, meaning that the exact same BLAS/LAPACK operation, with the same options (e.g., transposition), is being performed on matrices/vectors of different sizes. The non-uniformity in the dimensions is unconstrained; as long as the batch is resident on the GPU memory, any spectrum and distribution of dimensions is supported. This is different from the oneMKL approach, which supports non-uniform batch computations through *groups of uniform batches*, where dimensions and options are the same within a single group, but could vary across groups.

MAGMA supports non-uniform batch computations for all level-3 BLAS operations Abdelfattah et al. (2017b) and the level-2 matrix-vector products (GEMV and SYMV), as well as the Cholesky Abdelfattah et al. (2016a); Abdelfattah et al. (2016c) and LU factorizations Abdelfattah et al. (2022a). The core design idea for all these operations is to consider a GPU kernel as a one-dimensional array of 2D subgrids, where each subgrid is assigned a single BLAS/LAPACK operation. Each subgrid is originally configured to accommodate the largest problem in the batch, but then truncated on-the-fly to fit the problem it is assigned to. In addition, the BLAS/LAPACK interface and semantics for non-uniform batches have been extended in order to facilitate the description and developments of algorithms for different sizes, as demonstrated by Abdelfattah et al. (2022a). Some design optimizations have also been introduced for mitigating the load imbalance across the batch Abdelfattah et al. (2016c), but their generalization to a wide range of algorithms remains a future work.

Figures 17 and 18 show the performance advantage of MAGMA for the LU factorization on non-uniform batches of square matrices whose sizes are randomly sampled based on a uniform distribution. It is clear that launching a non-batch factorization in parallel streams/queues does not yield an acceptable performance. This is mainly due to the launch overhead, and the fact that non-batch solvers are not properly tuned for small sizes. The oneMKL performance is competitive, especially against the MI210 GPU. Due to the relatively large caches, most of the smaller matrices in the batch can fit in the fast cache levels of the CPU, resulting in a very good performance.

7. Performance portability and preliminary results on intel GPUs

A key legacy of the ECP on MAGMA will certainly be its expansion to include support for AMD and Intel GPUs, done in anticipation of the exascale machines Frontier (AMD), El Capitan (AMD), and Aurora (Intel). The HIP backend for AMD GPUs, originally developed by Brown et al. (2020), shares most of its codebase with MAGMA's original CUDA code, employing code generation scripts at compile time. This is possible due to the large similarity shared between the CUDA and HIP APIs. However, the extremely different syntax required by SYCL kernels and Intel's oneMKL compared to CUDA kernels and NVIDIA's cuBLAS/cuSPARSE libraries means that for Intel GPUs, the MAGMA team has elected to create separate code for the components that were originally CUDA-specific.

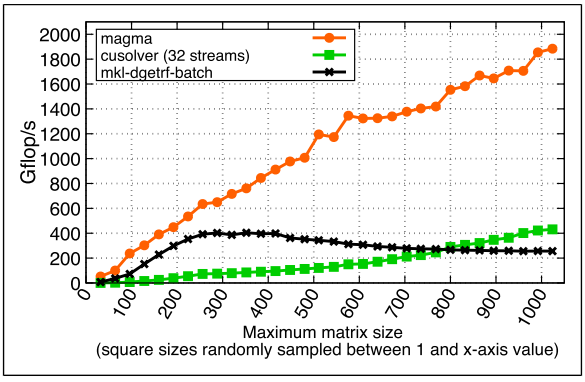

Figure 17. Performance of the non-uniform batch LU factorization in double precision. Each point represents 1000 square matrices of different sizes that are randomly sampled between 1 and the x-axis value. The GPU performance results are obtained on a Tesla A100-SXM4 GPU using CUDA-12.1 The CPU performance results are obtained by running the getrf_batch routine in one MKL 2024.0.2 on a 36-core dual socket Intel CPU (Intel Xeon Gold 6140 CPU running @ 2.30 GHz).

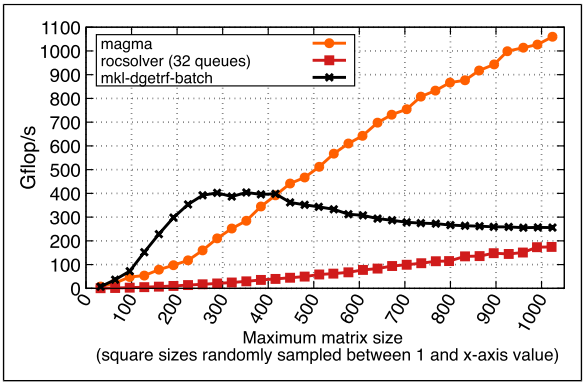

Figure 18. Performance of the non-uniform batch LU factorization in double precision. Each point represents 1000 square matrices of different sizes that are randomly sampled between 1 and the x-axis value. The GPU performance results are obtained on an Instinct MI210 GPU using ROCM-5.7.1. The CPU performance results are obtained by running the getrf_batch routine in one MKL 2024.0.2 on a 36-core dual socket Intel CPU (Intel Xeon Gold 6140 CPU running @ 2.30 GHz).

After initial exploration by Fortenberry and Tomov (2022) with a limited set of MAGMA routines, we have made heavy use of (Intel's DPC++ Compatibility Tool, 2024) (dpct) to automate the foundations of the CUDA-to-SYCL porting process. Intel also currently maintains dpct headers to provide some "CUDA-like" functionality that does not have a direct SYCL replacement. For example, in CUDA, there is no need to specify a device when allocating device memory, as it will automatically be allocated on the "current" device. In SYCL, there is no notion of a "current" device, which means that allocating device memory through sycl::malloc_device requires the user to provide a device and context (or a queue object, from which the associated device can be determined). dpct's device manager allows for setting a current device as one would do for CUDA, and it creates and stores a "default queue" for each device which can be used for memory allocation and transfer between host and device.

Even with the help of dpct, there are places where the clear influence of CUDA on MAGMA has posed challenges for porting to SYCL. Error handling with CUDA/cuBLAS error types–which are enums–had to be manually updated to use oneMKL's exceptions, for example. There are device properties available in CUDA which have no direct equivalent in SYCL, for which dpct does not provide a substitution. CUDA and HIP libraries use C-friendly types for complex numbers, while oneMKL uses C++ std::complex; as MAGMA has a C interface, this requires a split between MAGMA's type and the vendor's type and, of course, casting within MAGMA's code that calls oneMKL. However, Intel continues to provide new extensions which often help bridge gaps between CUDA/HIP and SYCL. While many of these are still experimental, they may become part of the SYCL standard in the future, and many are or will be officially supported by Intel in the meantime.

In addition to the expected software-related challenges associated with porting a large library like MAGMA, the SYCL porting efforts driven by ECP also illustrate the challenges of achieving performance portability on different architectures. Consider as an example the performance of MAGMA's dgetrf for LU factorization of dense matrices. MAGMA provides both a hybrid CPU-GPU and a GPU-native version of this routine. Figures 19 and 20 show preliminary results for dgetrf on the Aurora supercomputer at Argonne Leadership Computing Facility; each Aurora node has two Intel Xeon Max Series CPUs and six Intel Data Center Max Series GPUs (codename PVC). Each PVC GPU contains two tiles that can be used as separate SYCL devices, and we limit our results to one tile only. Note that this work was done on a pre-production supercomputer with early versions of the Aurora software development kit, using the 2024.0 release of oneMKL and the Intel icpx compiler. For all MAGMA kernels, the flag -ze-opt-intel-enable-auto-large-GRF-mode was used to allow the compiler to choose whether a kernel should use large register file mode, which reduces kernel occupancy by half but can reduce or eliminate register spilling. The performance of oneMKL's getrf is included for comparison. Both the hybrid and native MAGMA routines offer a performance boost compared to oneMKL. It is worth noting that the 2024.0 release of oneMKL requires the use of 64-bit integers for the pivot indices argument, while MAGMA is using 32-bit integers.

Both the hybrid and native versions utilize a mixture of MAGMA kernels and oneMKL BLAS calls and have

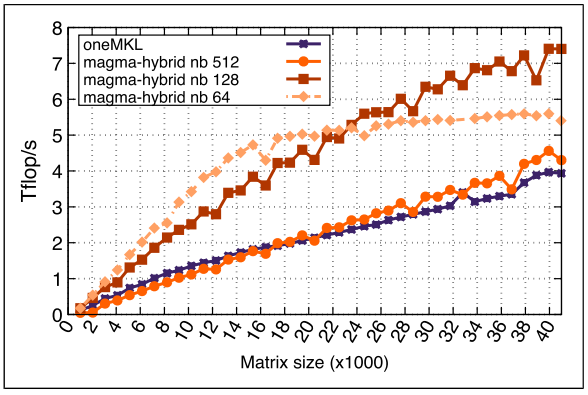

Figure 19. MAGMA hybrid CPU-GPU LU factorization with three different block sizes vs Intel's oneMKL (2024.0), on one tile of an Intel PVC GPU.

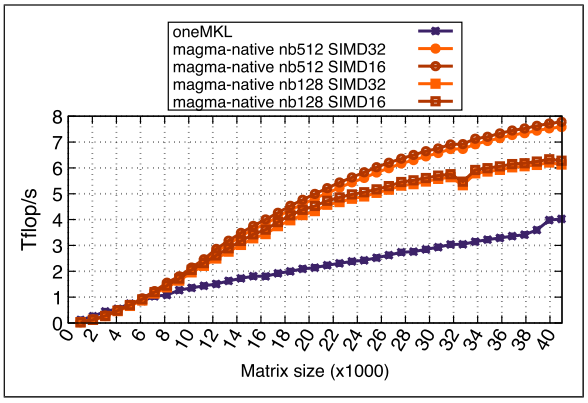

Figure 20. MAGMA native GPU LU factorization with two different block sizes and two different SIMD width choices vs Intel's one MKL (2024.0), on one tile of an Intel PVC GPU.

several parameters that can affect performance: the block size, nb, as well as kernel-specific parameters that are set at compile time to values found from previous tuning efforts on other hardware. For parameters other than nb, for which different values are plotted, we begin with MAGMA's current values chosen for NVIDIA GPUs. There is additional complexity due to the fact that in SYCL, the size of a sub-group – analogous to a warp in CUDA – is not fixed. For the GPUs on Aurora, for example, a sub-group can be either 32 or 16 work-items (= CUDA threads). In Figure 20 we include results from SIMD widths of 32 and 16 for the native MAGMA routine, and find that they are very similar. However, these tests set the default sub-group size for all kernels through a compiler flag. Individual kernels' sub-group sizes can also be set with an attribute. This increases the total possible configurations to try when tuning: it might raise the ceiling for performance improvement through parameter tuning alone, but it also augments the search area for finding the optimal combination of choices. The challenge of performance portability across different GPUs is a key advantage of MAGMA for application codes needing good BLAS/LAPACK performance, as they can write their code once and let MAGMA handle the details of tuning for a particular architecture. In the rest of the paper, we will give an overview of some of the ECP applications impacted by MAGMA.

8. Impact on selected ECP applications

8.1. STRUMPACK (STRUctured matrix PACKage)

STRUMPACK is a software library providing linear algebra routines and linear system solvers for sparse and for dense rank-structured linear systems Ghysels et al. (2016); Ghysels and Synk (2022). This section shows the impact of MAGMA on the performance of STRUMPACK in the direct solution of a sparse linear system of equations. Sparse direct solvers have the benefit that they are robust, compared to iterative methods, and require relatively little tuning by the user. Whereas iterative methods can have unpredictable convergence, direct methods solve linear systems with a predictable number of operations and memory usage. This is especially beneficial for numerically challenging problems, like highly indefinite and ill-conditioned linear systems, which can be hard to precondition. Another benefit of factorization based methods is that the factorization of the operator can be reused multiple times for the solution of different right hand sides. However, compared to most iterative methods, sparse direct solvers can require asymptotically more memory and floating point operations.

8.2. Overview of sparse direct solvers

There exists a number of sparse direct solvers, including SuperLU_Dist by [Li and Demmel (2003)], SuiteSparse (UMFPACK by [Davis (2004)] and Cholmod by [Chen et al. (2008) Chen, Davis, Hager and Rajamanickam]), PaStiX by Hénon et al. (2002), STRUMPACK by Ghysels et al. (2016), MUMPS by Amestoy and Duff (1989)]and WSMP by Gupta (2000). Out of these, SuperLU_Dist, Cholmod (only SPD), PaStiX and STRUMPACK have GPU support. SuperLU_Dist, PaStiX and Cholmod – supernodal but not multifrontal methods – have more complicated data dependencies than the multifrontal solver STRUMPACK. The multifrontal algorithm lends itself better to the use of batch routines on GPUs. However, the batches are often non-uniform, i.e. having matrices of different sizes, which is currently unsupported by the vendor software stack. Before the successful integration with MAGMA in the work done by Abdelfattah et al. (2022a), STRUMPACK originally relied on an internal implementation that uses a batch kernel restricted to matrix blocks smaller than 32×32. Larger matrices are handled through a loop calling

cuBLAS/cuSOLVER or rocBLAS/rocSOLVER, thus leading to significant kernel launch overhead.

In order to solve a sparse linear system $Ax = b$ with $A \in \mathbb{C}^{N\times N}$, STRUMPACK computes a decomposition of A as $P\,(D_r A D_c Q)P^T = LU$, where P and Q are permutation matrices, D_r and D_c are diagonal scaling matrices, and L and U are sparse lower and upper triangular factors. The permutation P, which aims to reduce the number of nonzeros in the triangular factors, is computed using the nested dissection algorithms from the METIS library, which is contributed by Karypis and Kumar (1998). The optional permutation Q, and the scaling factors D_r and D_c are computed using the MC64 matching code by Duff and Koster (1999). The goal of Q is to maximize the product of the diagonal elements, which are then scaled by D_r and D_c such that all diagonal entries are 1 and all off-diagonal entries are less than 1 in absolute value. There are three separate phases for the direct solution of sparse linear systems:

1. Reordering and symbolic analysis
2. Numerical factorization
3. Solve using forward and backward substitution

In Phase 1, the permutation and scaling vectors are computed, the sparsity pattern is analyzed, and data-structures are initialized to guide the numerical factorization phase. After computation of the sparse factorization (computationally the most expensive phase), a linear system can be solved efficiently by applying the permutations and scalings, and performing the sparse triangular solves with the L and U factors. To illustrate the numerical factorization phase, we consider a nested dissection permutation P with only two levels, and a single vertex separator:

$$PAP^T = \begin{bmatrix} A_1 & & X_{1S} \\ & A_2 & X_{2S} \\ X_{S1} & X_{S2} & S \end{bmatrix} \tag{1}$$

The lower-right sub-block S corresponds to a separator in the graph of A, effectively splitting the problem in two unconnected components, represented by A_1 and A_2. We can now construct three frontal matrices

$$F_1 = \begin{bmatrix} A_1 & \widehat{X}_{1S} \\ \tilde{X}_{S1} & \end{bmatrix}, F_2 = \begin{bmatrix} A_2 & \widehat{X}_{2S} \\ \tilde{X}_{S2} & \end{bmatrix}, F_0 = S, \tag{2}$$

where $\widehat{X}_{1S}/\tilde{X}_{S1}$ is the matrix consisting of only the columns/rows of X_{1S}/X_{S1} which contain nonzero elements. These fronts are put in a binary tree with F_0 as the root and F_1 and F_2 as the children. The numerical phase of the multifrontal LU factorization algorithm then traverses this binary tree from the leaves to the root. At each front $F_\tau = [F_{11}\,F_{12}; F_{21}\,F_{22}]$ (except the root), the following steps are performed:

- $P_\tau L_\tau U_\tau \leftarrow LU(F_{11})$
- $F_{12} \leftarrow U_\tau^{-1} L_\tau^{-1} P_\tau^T F_{12}$
- $F_{22} \leftarrow F_{22} - F_{21}F_{12}$

At the root front, only the first of these steps needs to be performed. The first step computes a dense LU factorization with partial pivoting. Note that the pivoting is restricted to the diagonal blocks, but for most problems, especially when combined with the permutation Q, this is sufficient to ensure numerical stability, and it greatly simplifies the implementation. After these operations are applied to front F_τ, the Schur complement $F_{\tau;22}$ is added into the parent frontal matrix. However, since the parent front is typically larger than the Schur complement of its child, this requires a scatter operation.

The described approach can be generalized by recursively applying the nested dissection heuristic to the subdomains A_1 and A_2, leading to a binary tree, referred to as the assembly tree, with $\mathcal{O}(\log N)$ levels. Going down the tree from the root to the leaves, subdomains and separators become smaller, leading to smaller frontal matrices, while the tree becomes wider. Note that all fronts in a given level can be handled concurrently. STRUMPACK's GPU implementation traverses the tree level-by-level, from leaves to root, using non-uniform batch algorithms for the dense linear algebra operations (LU, triangular solve (TRSM) and GEMM) for all fronts on a given level. This is where MAGMA is used to accelerate the processing of these batches, as shown by Abdelfattah et al. (2022). If the entire assembly tree does not fit in the device memory, then the factorization is split in multiple traversals of subtrees that do fit on the device.

8.3. Performance impact of MAGMA on STRUMPACK

We solve the electromagnetics problem corresponding to the second order Maxwell equation, $\nabla \times \nabla \times \mathbf{E} - \Omega^2 \mathbf{E} = \mathbf{f}$, which is given in the weak formulation as $(\nabla \times \mathbf{E}, \nabla \times \mathbf{E}') - (\Omega^2 \mathbf{E}, \mathbf{E}') = (\mathbf{f}, \mathbf{E}')$ with a testing function $\mathbf{E}'$. A given tangential field $\mathbf{f}(\mathbf{x}) = (\kappa^2 - \Omega^2)$ (sin (κx_2), sin (κx_3), sin (κx_1)) is used as the boundary condition for $\mathbf{E}$. For large wave numbers Ω, the problem is highly indefinite and hard to precondition, so typically a direct solver is used. The weak form is discretized with first order Nédélec elements using the modular finite element library MFEM Anderson et al. (2021). The results for $\Omega = 16$, $\kappa = \Omega/1.05$ and a toroidal geometry with hexahedral finite elements are shown in Figure 21. Results in this section all use FP64. Note that for these tests, the MC64 static pivoting is not required, and all sparse solves reach a backward error close to machine precision after a single step of iterative refinement. Figure 22 illustrates the distribution of the matrix

sizes, as well as the batchsize, for each batch. As the assembly tree is traversed from the leaves to the root (level 0), the average matrix size increases, while the batchsize decreases. Table 1 compares the total runtime for the numerical factorization on the A100 and MI100 using MAGMA against the loop-based implementation using cuBLAS/cuSOLVER or rocBLAS/rocSOLVER, with STRUMPACK v6.3.1, and with SuperLU_Dist Li and Demmel (2003) v7.2.0. From SuperLU_Dist, the 3D factorization code pdgssvx3d was used, which offloads more operations to the GPU, and performs better, compared to the earlier implementation in pdgssvx (the 2D algorithm). We used the default parameters for the STRUMPACK and SuperLU_Dist solvers. The CPU runs use 16 OpenMP threads and Cray LibSci for BLAS and LAPACK. The systems used for Table 1 both have 4 GPUs per node and 64 CPU cores per node, so the GPU and CPU tests use 1/4th of a node.

There are few takeaways that can be extracted from Table 1. First, STRUMPACK-MAGMA outperforms all the other solvers, especially on the AMD GPU, where STRUMPACK-MAGMA is 6.9× faster than STRUMPACK. Second, the smaller 9% gain for STRUMPACK-MAGMA on the A100 GPU is due to special internal optimizations in STRUMPACK that target very small matrices at the bottom of the assembly tree. All blocks larger than 32 × 32 are handled by cuBLAS/cuSOLVER. The 9% improvement in favor of STRUMPACK-MAGMA comes from supporting all sizes. On the MI100 GPU, the kernel launch overheads are more significant, which leads to bigger performance gains (13.56s for STRUMPACK vs 1.97s for STRUMPACK-MAGMA). Third, although Figures 17 and 18 show a clear advantage for the A100 GPU versus the MI210 GPU (and consequently the MI100), the final solution times in Table 1 are close (1.77s for the A100 GPU vs 1.97s for the MI100 GPU). We observe that reducing the kernel launch overhead on the MI100 GPU has a bigger impact on performance compared to the A100 GPU. By removing the launch overheads for the large batches on the lower levels of the assembly tree, the total runtime of the solver is dominated by nodes at the higher levels of the assembly tree. More specifically, the runtime is most impacted by the non-uniform batch GEMM on the largest nodes. Since MAGMA does not yet use the Tensor Cores on the A100 GPU, the theoretical peak performances for FP64 GEMMs are close (9.7 Tflop/s on the A100 GPU, vs 11.5 Tflop/s on the MI100 GPU).

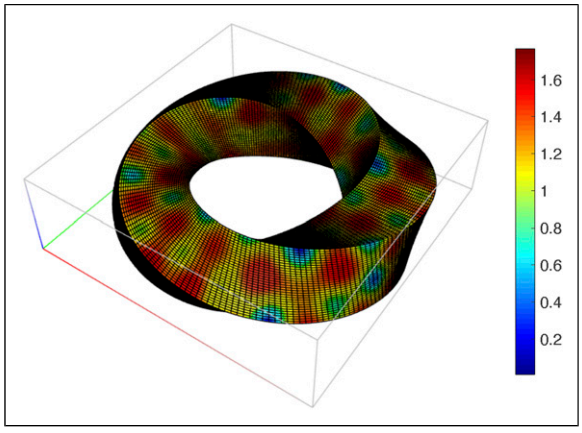

Figure 21. Example solution of the second order Maxwell equation on a toroidal domain discretized using Nédélec finite elements. The mesh has 614, 592 finite element unknowns with approximately 24 points per wavelength and the corresponding sparse matrix has 19, 636, 416 nonzeros.

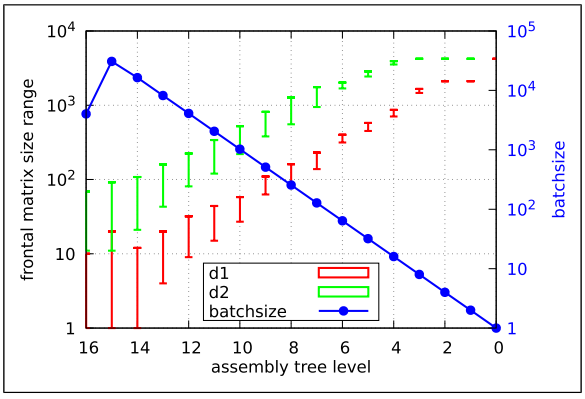

Figure 22. Distribution of the matrix sizes within each batch (level of the assembly tree). $d1$ are the dimensions of the upper-left block of each front, i.e., the sizes for the non-uniform batch LU operation. $d2$ are the dimensions for the lower-right part of each front. The non-uniform batch TRSM operation is applied with $m = d1$ and $n = d2$, and the non-uniform batch GEMM with $m = n = d2$ and $k = d1$. Level 15 has batchsize 30, 727.

8.4. Multiphysics simulations in MARBL

MARBL is a next-gen Multi-physics code developed at LLNL for simulating high energy density physics experiments including inertial confinement fusion. Since inception there has been a focus on scalability and performance of the code. To enable platform portability, the MARBL team adopted the RAJA portability suite and MFEM's memory management capabilities. In the work of Vargas et al. (2022), it is outlined how a combination of kernel abstractions and algorithmic refactoring was necessary to be productive and realize performance on the GPU. For high-order finite element operations it was necessary to switch from assembling and storing the operator as a matrix to matrix-free implementations. New kernels were written using RAJA, enabling the team to expose loop level parallelism, as shown by Beckingsale et al. (2019). MFEM's memory management system simplified host to device transfers and kept track of where the data was last modified,

Table 1. Numerical factorization performance on both NVIDIA A100 and AMD MI100, and comparison with STRUMPACK-MAGMA Abdelfattah et al. (2022a), STRUMPACK Ghysels and Synk (2022), and SuperLU_dist Li and Demmel (2003).

		A100		MI100	
		AMD 7763 (16c)		AMD 7662 (16c)	
	Solver	Time (s)	Gflop/s	Time (s)	Gflop/s
GPU	STRUMPACK-MAGMA	1.77	1394.7	1.97	1256.1
	STRUMPACK	1.93	1275.9	13.56	182.0
	SuperLU	12.58	190.3	-	-
CPU	STRUMPACK	8.62	286.6	10.16	243.2
	SuperLU	17.89	132.8	31.94	74.7

as shown by Anderson et al. (2021). Sitting underneath MFEM is the Umpire memory manager from the RAJA portability suite. Umpire is used to create memory pools enabling fast device allocations and deallocations as well as sharing device memory between physics packages, as shown by Beckingsale et al. (2020).

The focus in the work by Vargas et al. (2022) was on achieving performance on NVIDIA GPUs. With the upcoming El Capitan exascale supercomputer, based on AMD hardware, running performantly on AMD GPUs has now become a target. The work by Stitt et al. (2024) describes the continuing efforts to integrate new physics and achieve high performance on AMD GPUs while maintaining performance portability on existing platforms. The article dives into the importance of integrating vendor and third-party linear algebra routines. Integration of linear algebra libraries began with NVIDIA's packages, followed by AMD's, and finally MAGMA. A core development goal of the MARBL team is to minimize platform-specific code, which brought them to explore the MAGMA library. In the following section we highlight use-cases of the MARBL code and observed performance improvements.

8.5. Dense linear algebra in MARBL

In the MARBL code, dense linear algebra can easily become a GPU performance bottleneck and can be difficult to optimize for. MARBL uses a combination of continuous and discontinuous finite element spaces to represent kinematic and thermodynamic fields respectively. It is under the discontinuous finite element spaces that local dense linear algebra arises.

A core routine of MARBL requires inverting element local dense matrices such as the mass matrix from a discontinuous space. Dense matrix inversion is pivotal in the MARBL's ALE-remap algorithm as the inverse is applied to multiple fields. Compared to MARBL's own implementation the use of MAGMA v2.7.0 has delivered a 10x improvement in computing the matrix inverse on both NVIDIA V100's and the AMD MI250X. Depending on matrix size, MARBL found MAGMA or the vendor libraries to be more performant for computing the matrix inverse. The matrix inverse here was computed using batched LU factorization routines as described in the work by Stitt et al. (2024).

A second area which MARBL has been integrating MAGMA is within the radiation diffusion module. The key algorithm here solves a nonlinear problem through inexact Newton's method. Computing the Jacobian requires inverting dense local matrices that vary in size throughout the mesh. MAGMA provides a variable batch LU factorization routine which has delivered a 10× speedup over MARBL's native version. The 10× performance was observed using a mesh with 38720 3D elements on 8 V100 GPUs (4840 elements per device, Q2-Q1 discretization). The size of the matrices within an element are determined by

$$\left(\#num\_materials + \#radiation\_groups\right) \times \#DOF\right)^2$$

where DOF corresponds to degrees of freedom. For this type of computation neither NVIDIA and nor AMD provided a batched LU factorization routine.

Lastly, the MARBL team has been focusing on integrating new algorithms for linear solvers which target high performance on the GPU, as shown by Stitt et al. (2024). Following the work by Pazner et al. (2023), the MARBL team has integrated a saddle point system solver for their radiation diffusion module. The new method solves a saddle point system that generally requires 10s to 100s of MINRES iterations to converge. Solving small dense linear systems is required as part of each iteration and can easily become a computational bottleneck. The original implementation from Pazner et al. (2023) used an iterative matrix-free method that scales well with increasing polynomial order, however; most production uses of MARBL use quadratic

elements. It has been found that computing and applying the direct inverse with MAGMA is between 15× to 20× faster on AMD MI250X GPUs compared to the matrix-free implementation. Additionally, they have found that MAGMA 2.7.2 is 1.5× faster than hipBLAS 5.7.1 for quadratic elements. While the assembly and inversion of the matrices has a cost, it equivalent to only two to three solves of the matrix-free formulation and thus has low impact in practice. The MABRL team observes a diminishing advantage for the direct method when increasing to cubic elements – dropping to a 3× to 4× speedup relative to the matrix-free version – and at order 4, the performance advantage is a wash. MARBL now uses the direct method for its main production use cases and computing of mass matrix inverses in the saddle point system is no longer a performance bottleneck.

8.6. Crystal plasticity simulations using ExaConstit

ExaConstit is a GPU-capable, quasi-static, velocity-based, nonlinear solid mechanics finite element code designed around the MFEM library, as shown in the work by Carson et al. (2019) and Anderson et al. (2021). It is a part of a suite of software codes to model additive manufacturing processes for metals developed under the ECP ExaAM project, as demonstrated by Turner et al. (2022). As part of this suite of software, it is responsible for connecting the simulated microstructure produced by the additive manufacturing process to local properties for part scale models. Due to the anisotropic nature of additively manufacture materials, this connection between the microstructure and the part scale response requires running numerous crystal plasticity simulations to probe the response of the material under varying loading directions and temperature ranges. From these various simulations, an anisotropic macroscopic material model can be fit to improve part scale simulations. Typically for the ExaAM project, these material model fits would require on the order of 63 high-fidelity ExaConstit simulations per microstructure across multiple temperature ranges. Each one of these simulations on average would require roughly 10 min utilizing the GPUs on 8 nodes of Frontier. The ExaAM project was focused on studying the uncertainty within AM parts which required simulating the response of 125 microstructures which resulted in 7850 ExaConstit simulations, as featured in the work by Carson et al. (2023).

Due to the large number of ExaConstit simulations required in this ExaAM workflow, a great deal of focus was placed in the GPU port of ExaConstit for both NVIDIA's V100 (Summit) and AMD's MI250X (Frontier) hardware. The initial port of ExaConstit from the V100s to the MI250X resulted in a roughly 2× slowdown in a sizable test case when comparing runs on Summit to those on Oak Ridge National Laboratory's (ORNL) Crusher, an early access machine for the Frontier system. The test problem itself consisted of 4 million linear hexahedron elements and utilized 8 nodes of either Summit or Crusher. Through the use of performance annotations, the slowdown was determined to be solely caused by the action of the material tangent stiffness matrix within the linear/Krylov solve portion of the FEM calculations. This operation accounts for approximately 65% of the runtime of ExaConstit while running on GPUs due to the sheer number of times it is called during the life of the simulation. When using GPUs, the action of the material tangent stiffness matrix is a series of batch GEMV operations which represents the element local GEMV operations within the Krylov solver. The local GEMV products are then prolongated back to the global DOFs and then used by the Krylov solver. The matrix size for this test problem was a 24 × 24 system batched over 4 million elements across 64 GPUs. This batch operation initially utilized a naive GPU forall loop that performed the matrix vector operations. As an initial step to improve this operation, we utilized the batch GEMV operation in hipBLAS v5.4.3. From this code modification, we saw a minor performance improvement in the test problem, but the runs were still slower than the Summit runs. Utilizing an existing collaboration between the ExaAM project and the CEED project, we reached out to the MAGMA team who were able to optimize their batch GEMV calls for these small matrix sizes. These changes were incorporated into the v2.7.0 release of MAGMA and resulted in a 1.5× improvement when comparing Crusher to equivalent Summit runs, and a 3× improvement over the initial HIP port. A summary and timing for these runs is provided in Table 2. It should be noted that after these improvements were made we also reached out to AMD, and they were able to optimize their batch GEMV operations for small matrices so that they are now comparable to MAGMA as of hipBLAS v5.7.0. Ultimately, the changes to MAGMA enabled the 7850 ExaConstit simulations runs conducted in May 2023 on 8000 nodes of Frontier for the ExaAM UQ study to complete in 2 hours 15 min. This is a drastic saving in node-hours as the v5.4.3 of hipBLAS would have taken approximately 4.5 hours to complete with the same resource sets.

9. MAGMA-derived and MAGMA-inspired components for high-order finite element analysis

Throughout ECP, the MAGMA team was directly involved in the Center for Efficient Exascale Discretizations, [CEED (2020)], one of ECP's co-design centers, which is featured in the work by Kolev et al. (2021). In addition to innovations in existing software and collaboration with hardware vendors, the center produced a new library, libCEED, for enabling performant evaluation of high-order finite element

Table 2. Summary of batch matrix-vector optimizations within ExaConstit/MFEM for different platforms and total runtime of simulations.

Platform	Code modification	Time (s)
Summit	Initial HIP port	720
Crusher	Initial HIP port	1450
Crusher	hipBLAS batch GEMV operation	960
Crusher	MAGMA batch GEMV operation	480

operators, thanks to the work by Brown et al. (2021). libCEED has a low-level, general API for defining custom operators, which are typically applied through the use of *partial assembly*. In this type of matrix-free application, the full operator A is represented as a series of sub-operations, $A = \mathcal{E}^T B^T D B \mathcal{E}$, where some pre-computed data is stored ahead of time for use in the D operator (hence, "partial assembly"). The sub-operators $\mathcal{E}$, B, and D are implemented in libCEED's backends, which allows for efficiently targeting different architectures. libCEED has a MAGMA backend that benefits from MAGMA in two ways: directly, through targeted use of MAGMA routines; and indirectly, through the "MAGMA-inspired" specialized kernels which were developed specifically for libCEED.

Among libCEED's current backends for NVIDIA and AMD GPUs, the best performance for non-tensor basis functions is achieved with the MAGMA backend. This is realized through accelerating the B and B^T sub-operations. For non-tensor basis functions, B and B^T can be written in terms of standard GEMM routines. The dimensions of the GEMM are determined by P, the number of basis nodes per element; Q, the number of quadrature points per element; and N, the total number of elements in the operation (that is, the elements local to one GPU) multiplied by the number of components in the basis. In typical use, P and Q will be much smaller than N. The heavily-skewed dimensions mean that a batched GEMM routine will often outperform a single GEMM for the full B action.

To address this issue, the MAGMA backend introduced the "GEMM selector," which considers four options for performing B or B^T: batched GEMM from MAGMA or a vendor library (with batch size returned by the GEMM selector), or a single GEMM from MAGMA or a vendor library. The decision depends on P, Q, and N, and is based on tuning data collected into lookup tables within the libCEED MAGMA backend. When first introduced, the GEMM selector provided significant speedups over what was previously available in libCEED, but the benefit for lower orders of basis functions was not as clear, as demonstrated by Beams et al. (2020). Therefore, for lower orders, the MAGMA backend introduced new, "MAGMA-inspired" batched GEMM kernels. They are specialized to the case where one small input matrix (in this case, containing basis-function-defining information) is reused in all of the batched GEMM operations, while the other input matrix is split according to the batch size. Compared with the original GEMM selector, the new kernels improved asymptotic performance of an MFEM diffusion benchmark (CEED BP3, as defined by Fischer et al. (2020), modified to use tetrahedral elements) by 3.85× for linear basis functions on an NVIDIA A100 GPU and by nearly 2× on one GCD of an AMD MI250X GPU Kolev et al. (2023).

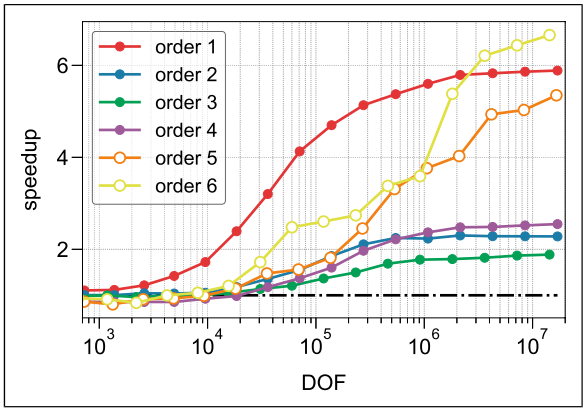

Figure 23. Speedup in MFEM-libCEED non-tensor diffusion benchmark for the deterministic MAGMA backend compared to the reference CUDA backend, for different orders of basis functions. Filled markers indicate the use of the specialized "MAGMA-inspired" kernels; unfilled markers mean the order uses the standard GEMM selector with the MAGMA and cuBLAS libraries. The reference of 1.0 is marked with a black dash-dot line. Results obtained on an NVIDIA A100-SXM4 GPU with MFEM v4.6, libCEED v0.12.0, and CUDA 12.1.

The final result is a clear benefit for lower and higher orders compared to the reference backends – the other GPU options for non-tensor elements. For CUDA backends, this is illustrated in Figure 23, where the diffusion benchmark speedup of the deterministic MAGMA backend over cuda-ref is plotted versus increasing total local degrees of freedom (DOF). The filled markers indicate the backend is using the specialized kernels, while the two lines with unfilled markers are using the standard GEMM selector. The worst-performing case (third-order basis functions) still achieves 70% speedup for larger local problem sizes, where the GPU backends are expected to perform most efficiently.

10. Conclusion

This paper highlighted the MAGMA library, an open source research project that provides optimized BLAS and LAPACK algorithms for GPU-accelerated compute nodes. The value of MAGMA is three-fold. First, it provides world-class performance for standard algorithms in dense linear algebra, with designs and implementations that are often

competitive or superior to the industrial software stack. Second, it ensures performance portability across NVIDIA, AMD, and Intel high-end GPUs, so that applications relying on MAGMA can easily be ported from one GPU architecture to another. Third, MAGMA provides crucial application support through introducing performance-critical software components that are not supported by the vendor. The paper detailed the importance of MAGMA to various ECP projects, including STRUMPACK, MARBL, ExaConstit, and CEED. The future of MAGMA lies in maintaining its advantage and appeal for applications, continuing to push the frontier in terms of performance and functionality, while ensuring portability across different GPU architectures.

The authors acknowledge that as the vendor math libraries become more mature over time, MAGMA's relevance and appeal to applications is challenged. However, MAGMA can maintain its position in scientific GPU computing in two directions. The first one is to provide robust abstractions to vendor libraries, which is already available in MAGMA for some vendor BLAS functions. This could evolve to a thin interfacing layer that not only supports more vendor routines, but also recommends the best performing option to the user (e.g., the vendor's implementation vs the MAGMA implementation). The second direction is to maintain focusing on new frontiers in numerical linear algebra, which are often developed first in research software libraries such as MAGMA before being embraced by the vendors. The expansion of mixed-precision algorithms to support FP16 arithmetic, the batch linear algebra software, and hybrid CPU-GPU algorithms are three examples of such frontiers.

Acknowledgements

This work was supported by the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357. Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-JRNL-860479. We also thank NVIDIA and AMD for their support and hardware donations.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the US Exascale Computing Project (17-SC-20-SC), U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357, Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-JRNL-860479.

ORCID iDs

Ahmad Abdelfattah https://orcid.org/0000-0001-5054-4784
Natalie Beams https://orcid.org/0000-0001-6060-4082
Robert Carson https://orcid.org/0000-0003-4490-2244
Tzanio Kolev https://orcid.org/0000-0002-2810-3090
Arturo Vargas https://orcid.org/0000-0001-8001-5517

References

Abdelfattah A, Haidar A, Tomov S, et al. (2016a) On the development of variable size batched computation for heterogeneous parallel architectures. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016, 1249–1258. DOI: 10.1109/IPDPSW.2016.190.

Abdelfattah A, Haidar A, Tomov S, et al. (2016b) Performance, design, and autotuning of batched GEMM for gpus. In: ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, 21–38. DOI: 10.1007/978-3-319-41321-1_2.

Abdelfattah A, Haidar A, Tomov S, et al. (2016c) Performance tuning and optimization techniques of fixed and variable size batched Cholesky factorization on GPUs. In: International Conference on Computational Science 2016 ICCS 2016, San Diego, California, USA, 6-8 June 2016, 119–130. DOI: 10.1016/j.procs.2016.05.303.

Abdelfattah A, Haidar A, Tomov S, et al. (2017a) Factorization and inversion of a million matrices using GPUs: challenges and countermeasures. In: International Conference on Computational Science, ICCS 2017, Zurich, Switzerland, 12-14 June 2017, 606–615. DOI: 10.1016/j.procs.2017.05.250.

Abdelfattah A, Haidar A, Tomov S, et al. (2017b) Novel HPC techniques to batch execution of many variable size BLAS computations on gpus. In: Gropp WD, Beckman P, Li Z, et al. (eds) In: Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14-16, 2017. ACM, 5:1–5:10. DOI: 10.1145/3079079.3079103.

Abdelfattah A, Haidar A, Tomov S, et al. (2018a) Batched one-sided factorizations of tiny matrices using GPUs: challenges and countermeasures. *Journal of Computational Science* 26: 226–236. DOI: 10.1016/j.jocs.2018.01.005. https://www.sciencedirect.com/science/article/pii/S1877750317311456

Abdelfattah A, Haidar A, Tomov S, et al. (2018b) Analysis and design techniques towards high-performance and energy-efficient dense linear solvers on GPUs. *IEEE Transactions*

on Parallel and Distributed Systems 29(12): 2700–2712. DOI: 10.1109/TPDS.2018.2842785.

Abdelfattah A, Tomov S and Dongarra JJ (2019a) Fast batch matrix multiplication for small sizes using half precision arithmetic on GPUs. In: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019. pp. 111–122.

Abdelfattah A, Tomov S and Dongarra JJ (2019b) Progressive optimization of batched LU factorization on GPUs. In: *2019 IEEE High Performance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019*. Piscataway: IEEE, 1–6.

Abdelfattah A, Ghysels P, Boukaram W, et al. (2022a) Addressing irregular patterns of matrix computations on GPUs and their impact on applications powered by sparse direct solvers. In: Wolf F, Shende S, Culhane C, et al. (eds) SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022, 26:1–26:14.

Abdelfattah A, Tomov S and Dongarra JJ (2022b) Batch QR factorization on GPUs: design, optimization, and Tuning. In: Groen D, de Mulatier C, Paszynski M, et al. (eds) Computational Science - ICCS 2022 - 22nd International Conference, London, UK, June 21-23, 2022, 60–74.

Alexander F, Almgren A, Bell J, et al. (2020) Exascale applications: skin in the game. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences* 378(2166): 20190056. https://www.osti.gov/biblio/1594850

Amestoy PR and Duff LS (1989) Vectorization of a multiprocessor multifrontal code. *The International Journal of Supercomputing Applications* 3(3): 41–59. DOI: 10.1177/109434208900300303.

Anderson MJ, Sheffield D and Keutzer K (2012) A predictive model for solving small linear algebra problems in GPU registers. In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. Piscataway: IEEE, 2–13. DOI: 10.1109/IPDPS.2012.11.

Anderson R, Andrej J, Barker A, et al. (2021) MFEM: a modular finite element methods library. *Computers & Mathematics with Applications* 81: 42–74. DOI: 10.1016/j.camwa.2020.06.009. https://www.sciencedirect.com/science/article/pii/S0898122120302583

Angerson E, Bai Z, Dongarra J, et al. (1990) Lapack: a portable linear algebra library for high-performance computers. In: *Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Piscataway: IEEE, 2–11. DOI: 10.1109/SUPERC.1990.129995.

Anzt H, Tomov S and Dongarra J (2014) *Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus*. Knoxville, USA: University of Tennessee. https://www.icl.utk.edu/sites/icl/files/publications/2014/icl-utk-772-2014.pdf

Anzt H, Tomov S, Luszczek P, et al. (2015) Acceleration of GPU-based Krylov solvers via data transfer reduction. *The International Journal of High Performance Computing Applications* 29(3): 366–383. DOI: 10.1177/1094342015580139.

Anzt H, Baboulin M, Dongarra JJ, et al. (2016a) Accelerating the conjugate gradient algorithm with GPUs in CFD simulations. In: Dutra I, Camacho R, Barbosa JG, et al. (eds) High Performance Computing for Computational Science - VECPAR 2016 - 12th International Conference, Porto, Portugal, June 28-30, 2016, 35–43.

Anzt H, Chow E, Huckle T, et al. (2016b) Batched generation of incomplete sparse approximate inverses on GPUs. In: *7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2016, Salt Lake, UT, USA, November 14, 2016*. Piscataway: IEEE Computer Society, 49–56.

Anzt H, Dongarra JJ, Flegar G, et al. (2017) Batched gauss-Jordan elimination for block-Jacobi preconditioner generation on GPUs. In: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM@PPoPP 2017, Austin, TX, USA, February February 2017. pp. 1–10. DOI:10.1145/3026937.3026940.

Auer AA, Baumgartner G, Bernholdt DE, et al. (2006) Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics* 104(2): 211–228. DOI: 10.1080/00268970500275780.

Baboulin M, Dongarra JJ and Tomov S (2008) Some issues in dense linear algebra for multicore and special purpose architectures. https://www.netlib.org/lapack/lawnspdf/lawn200.pdf

Baboulin M, Buttari A, Dongarra J, et al. (2009a) Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications* 180(12): 2526–2533.

Baboulin M, Buttari A, Dongarra J, et al. (2009b) Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications* 180(12): 2526–2533. DOI: 10.1016/j.cpc.2008.11.005. https://www.sciencedirect.com/science/article/pii/S0010465508003846

Beams N, Abdelfattah A, Tomov S, et al. (2020) High-order finite element method using standard and device-level batch gemm on gpus. In: *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. Piscataway: IEEE, 53–60.

Beckingsale DA, Burmark J, Hornung R, et al. (2019a) Raja: portable performance for large-scale scientific applications. In: *2019 Ieee/acm International Workshop on Performance, Portability and Productivity in Hpc (P3hpc)*. Piscataway: IEEE, 71–81.

Beckingsale DA, McFadden MJ, Dahm JP, et al. (2020) Umpire: application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development* 64(3/4): 1.

BLAS (1980) BLAS (basic linear algebra subprograms). https://www.netlib.org/blas

Boukaram WH, Turkiyyah G, Ltaief H, et al. (2018) Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Computing* 74: 19–33. DOI: 10.1016/j.parco.2017.09.001.

Brown C, Abdelfattah A, Tomov S, et al. (2020) Design, optimization, and benchmarking of dense linear algebra algorithms on AMD gpus. In: 2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020.

Brown J, Abdelfattah A, Barra V, et al. (2021) libceed: fast algebra for high-order element-based discretizations. *Journal of Open Source Software* 6(63): 2945.

Buck I, Foley T, Horn D, et al. (2004) Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics* 23(3): 777–786. DOI: 10.1145/1015706.1015800.

Buttari A, Dongarra J, Langou J, et al. (2007) Mixed precision iterative refinement techniques for the solution of dense linear systems. *The International Journal of High Performance Computing Applications* 21(4): 457–466. DOI: 10.1177/1094342007084026.

Carson E and Higham NJ (2017) A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM Journal on Scientific Computing* 39(6): A2834–A2856.

Carson E and Higham NJ (2018) Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing* 40(2): A817–A847. DOI: 10.1137/17M1140819.

Carson RA, Wopschall SR and Bramwell JA (2019) ExaConstit. https://github.com/LLNL/ExaConstit

Carson R, Rolchigo M, Coleman J, et al. (2023) Uncertainty quantification of metal additive manufacturing processing conditions through the use of exascale computing. In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023*. New York: ACM. DOI: 10.1145/3624062.3624103.

CEED (2020) Ceed. URL https://ceed.exascaleproject.org/.

Chen Y, Davis TA, Hager WW, et al. (2008) Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software* 35(3): 1–14. DOI: 10.1145/1391989.1391995.

Chow E, Anzt H and Dongarra JJ (2015) Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In: Kunkel JM and Ludwig T (eds) High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, 1–16. DOI: 10.1007/978-3-319-20119-1_1.

cuBLAS (2024) NVIDIA CUDA Basic Linear Algebra Subprograms. Available at: https://developer.nvidia.com/cublas

CUDA (2007) *Compute Unified Device Architecture Programming Guide, Version 1.0*. Santa Clara, CA, USA: NVIDIA Corporation.

Davis TA (2004) Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software* 30(2): 196–199. DOI: 10.1145/992200.992206.

Demmel JW (1997) *Applied Numerical Linear Algebra*. Philadelphia: Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611971446. https://epubs.siam.org/doi/abs/10.1137/1.9781611971446

Duff IS and Koster J (1999) The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications* 20(4): 889–901. DOI: 10.1137/S0895479897317661.

Fischer P, Min M, Rathnayake T, et al. (2020) Scalability of high-performance pde solvers. *The International Journal of High Performance Computing Applications* 34(5): 562–586.

Fortenberry A and Tomov S (2022) Extending magma portability with oneapi. In: SC 2022 Workshop on Accelerator Programming Using Directives (WACCPD), Dallas United States, Nov. 13 - 18, 2022, 22–31. DOI: 10.1109/WACCPD56842.2022.00008.

Ghysels P and Synk R (2022) High performance sparse multifrontal solvers on modern gpus. *Parallel Computing* 110: 102897. DOI: 10.1016/j.parco.2022.102897. https://www.sciencedirect.com/science/article/pii/S0167819122000059.

Ghysels P, Li XS, Rouet FH, et al. (2016) An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing* 38(5): S358–S384. DOI: 10.1137/15M1010117.

Golub GH and Ye Q (1999) Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM Journal on Scientific Computing* 21(4): 1305–1320. DOI: 10.1137/S1064827597323415.

Gupta A (2000) *WSMP: Watson Sparse Matrix Package Part II – Direct Solution of General Systems*. Yorktown Heights: Watson Research Center. https://s3.us.cloud-object-storage.appdomain.cloud/res-files/1331-wsmp2.pdf.

Haidar A, Dong T, Luszczek P, et al. (2015) Batched matrix computations on hardware accelerators based on GPUs. *The International Journal of High Performance Computing Applications* 29: 193–208. DOI: 10.1177/1094342014567546.

Haidar A, Abdelfatah A, Tomov S, et al. (2017) High-performance Cholesky factorization for GPU-only execution. In: *Proceedings of the General Purpose GPUs, GPGPU-10*. New York, NY, USA: ACM, 42–52. DOI: 10.1145/3038228.3038237.

Haidar A, Tomov S, Dongarra J, et al. (2018) Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18*. Piscataway, NJ, USA: IEEE Press, 47:1. DOI: 10.1109/SC.2018.00050 DOI: 10.1109/SC.2018.00050.

Hénon P, Ramet P and Roman J (2002) PaStiX: a high-performance parallel direct solver for sparse symmetric

positive definite systems. *Parallel Computing*. 28(2): 301–321. DOI: 10.1016/S0167-8191(01)00141-7. https://www.sciencedirect.com/science/article/pii/S0167819101001417.

HIP (2024) AMD HIP framework. https://rocm.docs.amd.com/projects/HIP/en/latest/.

Intel's DPC++ Compatibility Tool (2024) Intel® DPC++ compatibility Tool: migrate your CUDA* code to portable C++ with SYCL* multiarchitecture code. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compatibility-tool.html.

Karypis G and Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20(1): 359–392. DOI: 10.1137/S1064827595287997.

Kolev T, Fischer P, Min M, et al. (2021) Efficient exascale discretizations: high-order finite element methods. *The International Journal of High Performance Computing Applications* 35(6): 527–552.

Kolev T, Fischer P, Abdelfattah A, et al. (2023) *CEED ECP Milestone Report: support ECP applications in their exascale challenge problem runs*.

Kurzak J, Tomov S and Dongarra JJ (2012) Autotuning GEMM kernels for the fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23(11): 2045–2057. DOI: 10.1109/TPDS.2011.311.

Langou J, Langou J, Luszczek P, et al. (2006) Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06*. New York, NY, USA: Association for Computing Machinery, 113.

Li XS and Demmel JW (2003) SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software* 29(2): 110–140.

Masliah I, Abdelfattah A, Haidar A, et al. (2016) High-performance matrix-matrix multiplications of very small matrices. In: Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016. pp. 659–671. DOI:10.1007/978-3-319-43659-3.

Messer OEB, Harris JA, Parete-Koon S, et al. (2013) Multicore and accelerator development for a leadership-class stellar astrophysics code. In: Manninen P and Öster P (eds) *Applied Parallel and Scientific Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 92–106.

MKL (2024) Intel oneAPI math kernel library. Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

Moler CB (1967) Iterative refinement in floating point. *Journal of the ACM* 14(2): 316–321. DOI: 10.1145/321386.321394.

Nath R, Tomov S and Dongarra J (2010) An improved MAGMA GEMM for fermi graphics processing units. *The International Journal of High Performance Computing Applications* 24(4): 511–515, URL. DOI: 10.1177/1094342010385729.

Nath R, Tomov S, Dong T, et al. (2011) Optimizing symmetric dense matrix-vector multiplication on GPUs. In: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011.

OpenBLAS (2023) OpenBLAS: an optimized BLAS library. https://www.openblas.net/.

Pazner W, Kolev T and Vassilevski P (2023) *Matrix-free Gpu-Accelerated Saddle-point Solvers for High-Order Problems in H(div)*. Ithaca: Cornell University.

Reinders J, Ashbaugh B, Brodman J, et al. (2021) *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Berlin: Springer Nature.

rocBLAS (2024) rocBLAS, next generation BLAS implementation for ROCm platform. https://github.com/ROCmSoftwarePlatform/rocBLAS.

ROCm (2024) AMD ROCm™ software. https://www.amd.com/en/products/software/rocm.html.

Saad Y (1993) A flexible inner-outer preconditioned gmres algorithm. *SIAM Journal on Scientific Computing* 14(2): 461–469. DOI: 10.1137/0914028.

Saad Y and Schultz MH (1986) GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 7(3): 856–869. DOI: 10.1137/0907058.

Simoncini V and Szyld DB (2002) Flexible inner-outer krylov subspace methods. *SIAM Journal on Numerical Analysis* 40(6): 2219–2239. DOI: 10.1137/S0036142902401074.

Stitt T, Belcher K, Campos A, et al. (2024) Performance portable gpu acceleration of a high-order finite element multiphysics application. *Journal of Fluids Engineering* 146: 1–69.

Strohmaier E, Dongarra J, Simon H, et al. (2023) Top 500: novermber 2023. https://www.top500.org/lists/top500/2023/11/.

SYCL (2024) SYCL. https://www.khronos.org/sycl.

Tomov S, Dongarra J, Volkov V, et al. (2009) *Magma Library*. Knoxville, TN, and Berkeley, CA: Univ. of Tennessee and Univ. of California.

Tomov S, Nath R, Ltaief H, et al. (2010) Dense linear algebra solvers for multicore with GPU accelerators. In: *Proc. Of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, 1–8. DOI: 10.1109/IPDPSW.2010.5470941.

Tomov S, Dongarra JJ and Baboulin M (2010a) Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 36(5-6): 232–240. DOI: 10.1016/j.parco.2009.12.005.

Tomov S, Nath R and Dongarra J (2010b) Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel*

Computing 36(12): 645–654. DOI: 10.1016/j.parco.2010.06.001.

Turner JA, Belak J, Barton N, et al. (2022) ExaAM: metal additive manufacturing simulation at the fidelity of the microstructure. *The International Journal of High Performance Computing Applications* 36(1): 13–39. DOI: 10.1177/10943420211042558.

Vargas A, Stitt TM, Weiss K, et al. (2022) Matrix-free approaches for gpu acceleration of a high-order finite element hydrodynamics application using mfem, umpire, and raja. *The International Journal of High Performance Computing Applications* 36(4): 492–509.

Wilkinson JH (2023) Rounding errors in algebraic processes. Philadelphia, PA: Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611977523. https://epubs.siam.org/doi/abs/10.1137/1.9781611977523.

Yeralan SN, Davis TA, Sid-Lakhdar WM, et al. (2017) Algorithm 980: sparse QR factorization on the GPU. *ACM Transactions on Mathematical Software* 44(2): 1–29. DOI: 10.1145/3065870.

Author biographies

Ahmad Abdelfattah is a research assistant professor at the Innovative Computing Laboratory at the University of Tennessee. He received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). His research interests span high performance computing, parallel numerical algorithms, and general purpose GPU computing. He currently serves as the lead developer of the MAGMA library. He has been acknowledged by NVIDIA and AMD for contributing to their numerical BLAS libraries, cuBLAS and rocBLAS, respectively.

Natalie Beams is a research assistant professor in the Innovative Computing Laboratory at the University of Tennessee. She holds a Ph.D. in Theoretical and Applied Mechanics from the University of Illinois at Urbana-Champaign. Her research interests include numerical methods for PDEs, scientific computing, and high-performance computing.

Robert Carson is a Staff Scientist at Lawrence Livermore National Laboratory working on micromechanical and material modeling. His main research interest lies at the intersection of high-performance computing and computational mechanics and material modelling. Additionally, he enjoys connecting that research to modelling novel experimental platforms in-order to push material models forward.

Pieter Ghysels is a research scientist in the Scalable Solver group of the Computational Research Division at Lawrence Berkeley National Laboratory. He holds a PhD in Applied Mathematics and Computer Science from the KULeuven, Belgium. Pieter is the main author of the STRUMPACK linear algebra package, which provides fast dense and sparse solvers and preconditioners based on rankstructured matrix approximations. His interests are in High Performance Computing and Numerical Linear Algebra. Current research activities also include GPU acceleration and the application of deep learning algorithms to graph and sparse matrix computations.

Tzanio Kolev is a computational mathematician at the Center for Applied Scientific Computing (CASC) in Lawrence Livermore National Laboratory, where he works on finite element discretizations and solvers for problems in compressible shock hydrodynamics, multi-material arbitrary Lagrangian Eulerian methods, radiation hydrodynamics, and computational electromagnetics. He won an R&D100 award as a member of the hypre team. Tzanio is leading the high-order finite element discretization research and development efforts in the MFEM and BLAST projects in CASC and was the director of the Center for Efficient Exascale Discretizations (CEED) in DOE's Exascale Computing Project.

Thomas Stitt is a software developer at the Lawrence Livermore National Laboratory focusing on high performance computing applications, particularly performance portable routines and abstractions, for the Multi-Physics on Advanced Platforms Project.

Arturo Vargas is a computer scientist at Lawrence Livermore National Laboratory (LLNL). At LLNL, he works on the Multi-Physics on Advanced Platforms Project focusing on computational performance for high-order finite element methods. Additionally, he works on the RAJA project, a collection of C++ software abstractions which enable architecture portability.

Stan Tomov was a Research Associate Professor in the Electrical Engineering and Computer Science Department at the University of Tennessee, Knoxville until January 2024. He worked at the Brookhaven National Laboratory before joining ICL in 2004. Tomov's research interests are in parallel algorithms, numerical analysis, and HPC. He led the development of the MAGMA libraries, heFFTe, Batched BLAS and LAPACK for GPUs, MagmaDNN, and the hardware thrust for the Center for Efficient Exascale Discretizations (CEED) from the DOE ECP project. Tomov led industrial collaborations including the CUDA Center of Excellence at UTK, Intel Parallel Computing Center at ICL, and longterm collaborations with AMD and MathWorks.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his PhD in Applied Mathematics from the University of New

Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as Professor Emeritus and Research Professor in the Computer Science Department at the University of Tennessee, has the position of a Distinguished Research Staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow in the Computer Science and Mathematics Schools at the University of Manchester, and an Adjunct Professor in the Computer Science Department at Rice University. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced-computer architectures, programming methodology, and tools for parallel computers. His research includes the development, testing and documentation of high quality mathematical software. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 300 articles, papers, reports and technical memoranda and he is coauthor of several books. He was awarded the IEEE Sid Fernbach Award in 2004, the first IEEE Medal of Excellence in Scalable Computing in 2008, the first award of the SIAM Special Interest Group on Supercomputing in 2010, the IEEE Charles Babbage Award in 2011, and the ACM/IEEE Ken Kennedy Award in 2013. He received the 2021 A. M. Turing Award from the Association of Computing Machinery, an award that is often described as the ”Nobel Prize of computing.” He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a foreign member of the Russian Academy of Sciences and a member of the US National Academy of Engineering.