SIAM J. SCI. COMPUT.
Vol. 48, No. 6, pp. B1009–B1036
© 2026 Society for Industrial and Applied Mathematics

AN ADAPTIVE FINITE ELEMENT MULTIGRID SOLVER USING GPU ACCELERATION*

M. LIEBCHEN†, R. JENDERSIE‡, U. KAYA†, C. LESSIG§, AND T. RICHTER¶

Abstract. Adaptive finite elements combined with geometric multigrid solvers are one of the most efficient numerical methods for problems such as the instationary Navier–Stokes equations. Yet despite their efficiency, computations remain expensive and the simulation of, for example, complex flow problems can take many hours or days. GPUs provide an interesting avenue to speed up the calculations due to their very high theoretical performance. However, the large degree of parallelism and non-standard API make the use of GPUs in scientific computing challenging. In this work, we develop a GPU acceleration for the adaptive finite element library Gascoigne and study its effectiveness for different systems of partial differential equations. Our goal is thereby to integrate the GPU acceleration into the existing code with minimal changes. Through the systematic formulation of all computations as linear algebra operations, we can employ GPU-accelerated linear algebra libraries, which simplifies the implementation and ensures the maintainability of the code while achieving very efficient GPU utilizations. Our results for a transport-diffusion equation, linear elasticity, and the instationary Navier–Stokes equations show substantial speedups of up to 20X compared to multicore CPU implementations.

Key words. adaptive finite elements, GPU parallelization, Navier–Stokes equations

MSC codes. 68Q25, 68R10, 68U05

DOI. 10.1137/24M170435X



See reproducibility of computational results at end of the article.

1. Introduction.

The combination of adaptive finite elements and geometric multigrid solvers is one of the most efficient approaches for the numerical approximation of partial differential equations. Adaptive mesh refinement schemes based on numerical a posteriori error estimates allow for optimal refinement [50] while geometric multigrid solvers provide the solution to the arising linear systems of equations in linear complexity [25]. For many partial differential equations, the combination yields an overall approach with optimal computational complexity for a prescribed accuracy.

The mathematical and computational optimality come at the price of complex algorithms and a lack of regular structure in the computations. This is due to the adaptivity of the discretization, resulting in an irregular mesh and stencils, and the multiscale nature of the solvers. Furthermore, memory is often accessed in an unstructured way, hampering the parallelization of the computations.

*Submitted to the journal's Software, High-Performance Computing, and Computational Science and Engineering section October 22, 2024; accepted for publication (in revised form) August 3, 2026; published electronically November 23, 2026.

https://doi.org/10.1137/24M170435X

Funding: The work of the second author was supported by Schmidt Sciences. The work of the first, third, and fifth author was supported by the GRK 2297 Math-CoRe, funded by the Deutsche Forschungsgemeinschaft, grant 314838170. The work of the fourth and fifth author was supported by the Deutsche Forschungsgemeinschaft, grant 537063406.

†Institute of Analysis and Numerics, OVGU Magdeburg, Germany (manuel.liebchen@ovgu.de; utku.kaya@ovgu.de).

‡Institute of Analysis and Numerics and Institute of Simulation and Graphics, OVGU Magdeburg, Germany (robert.jendersie@ovgu.de).

§European Centre for Medium-Range Weather Forecasts (christian.lessig@ecmwf.int).

¶Corresponding author. Institute of Analysis and Numerics, OVGU Magdeburg, Germany (thomas.richter@ovgu.de).

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

These challenges are one of the central reasons that parallel accelerator coprocessors, such as GPUs and TPUs, are still only rarely used in numerical simulations, including adaptive multigrid methods. Accelerators are highly attractive since the computational power of CPUs has stagnated in the last decade while those accelerators has increased dramatically.

Accelerators use task- and data-parallelism as well as specialization to achieve a very large peak performance. While a high degree of parallelism is inherent in some problems, e.g., in computer graphics and deep learning, it poses challenges for the implementation of adaptive algorithms in numerical linear algebra and scientific computing. One direction to address this gap is by changing to algorithms with highly regular computations, such as lattice Boltzmann methods in fluid mechanics [40, 42].

The performance characteristics of modern accelerator hardware, in particular GPUs, fundamentally differ from those of traditional CPUs. While GPUs offer massive parallelism and high arithmetic throughput, the performance in computations is in practice is often limited by bandwidth and latency between GPU-RAM and the computational units on a GPU, as well as by the cost of data transfers between host (the CPU) and device (the GPU). These constraints pose significant challenges for classical finite element methods, which typically rely on sparse matrix storage and irregular memory access patterns to be efficient. As a consequence, a substantial body of recent work has focused on algorithmic reformulations of finite element methods that reduce memory traffic and increase arithmetic intensity to make more efficient use of GPUs. A key development in this direction is the use of matrix-free methods, where discrete operators are applied without explicitly assembling and storing sparse matrices. Instead, operator entries are recomputed on the fly, typically using tensor-product structure and sum-factorization techniques for higher-order elements. An overview of such approaches is given in [33], where the potential of tensor-product evaluations and sum factorization for high-performance finite element computations is discussed, albeit not yet in the context of multigrid solvers. These ideas have since become central to GPU-oriented finite element design due to their favorable compute-to-memory ratio [36].

On GPUs, particularly high performance for multigrid methods has been demonstrated for fixed-stencil discretizations, which can be interpreted as a limit case of matrix-free computation. Thereby the application of an operator reduces to a small number of regular, local stencil operations with predictable memory access patterns. Recent studies such as [3, 43] report excellent GPU efficiency for such methods, including for example p-multigrid techniques. While these results represent an important performance benchmark, their applicability to general, adaptive finite element methods is limited by the strong reliance on regular grid structures and fixed discretizations.

Also general finite element frameworks have made significant progress in leveraging GPUs. The MFEM library [2] represents a modern, GPU-centric approach that supports matrix-free higher-order finite elements and provides p-multigrid preconditioning for Krylov subspace solvers. A distinguishing feature of MFEM is its partial realization of an end-to-end GPU workflow, in which most stages of the solver—including operator application and multigrid components—can be executed on the device, thereby minimizing data transfers. A similar direction is pursued in the deal.II library [36]. There GPU support has been steadily expanded within a matrix-free high-order finite element framework. Recent developments emphasize performance portability and the integration of accelerator backends into existing multigrid and matrix-free infrastructures. While these efforts demonstrate the feasibility of combin-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

ing geometric multigrid with GPUs in general-purpose finite element libraries, they also highlight the complexity of designing algorithms that remain efficient across different hardware architectures and use cases.

A central difficulty in matrix-free multigrid methods on GPUs lies in the design of effective smoothers. Due to the lack of an explicitly assembled matrix, smoothers are often restricted to point-wise operations such as Jacobi or Chebyshev iterations. This may lead to reduced multigrid efficiency, especially for high-order discretizations [36]. Notable progress is in recent contributions on multigrid methods for high-order discontinuous Galerkin discretizations [19, 17], where efficient GPU-based vertex-patch smoothers are developed. These approaches demonstrate that carefully designed local solvers exploiting tensor-product structure can overcome some of the traditional limitations of matrix-free smoothing. The same concepts have also been successfully applied to the Stokes equations using multigrid directly in the coupled velocity–pressure [18]. In [26], the authors investigate the GPU parallelization of several smoothers within a matrix-based geometric multigrid method. The implementation allows for locally refined unstructured meshes, but, the obtained speedup is limited and does not substantially exceed that of multicore CPUs.

In contrast to matrix-free methods, matrix-based approaches on GPUs are less common, primarily due to the high memory footprint of sparse matrices and the irregular memory access patterns of sparse matrix–vector products. Nevertheless, these approaches remain attractive because they can often be integrated more easily into existing simulation frameworks. In [56], a two-level multigrid scheme with ILU-based smoothing on GPUs is investigated, demonstrating that meaningful acceleration can still be achieved for certain problem classes. A broader perspective on GPU usage in large-scale solvers is provided in the context of the PETSc framework [35]. The authors discuss solver design choices and scaling behavior toward exascale systems.

The primary advantage of matrix-free methods lies in their ability to drastically reduce memory usage by recomputing operator entries when needed. In situations where matrix-free solvers are not readily available—such as in complex multiphysics applications or legacy codes—mixed-precision techniques offer an alternative strategy. A general discussion of mixed-precision algorithms on GPUs, CPUs, and hybrid systems is given in [1]. In [45] mixed-precision is made possible by using a hierarchical approach to modify the linear systems, denoted as prehandling. In the context of multigrid solvers, mixed precision has been explored particularly for algebraic multigrid methods in [55]. The potential of these approaches stems not only from reduced memory bandwidth requirements, but also from the significantly higher memory and compute throughput of low-precision arithmetic on modern GPUs.

For algebraic multigrid methods, GPU accelerated libraries exist, e.g., AmgX [37] and Ginkgo [4]. NVIDIA [49] has also presented a highly efficient geometric multigrid solver for 3d linear elliptic problems. A highly optimized pressure Poisson solver in GPUs is discussed in [52].

An alternative memory-reduction strategy orthogonal to matrix-free methods is proposed in [11]. There sparse matrices are represented in a block-wise manner using embedded stencil-like structures. This approach aims to combine some of the flexibility of matrix-based methods with the reduced memory footprint and regular access patterns typically associated with stencil computations.

Finally, a fundamentally different direction is the use of machine learning techniques, in particular deep neural networks, as surrogate models or solver components in scientific simulations. Such models can be executed very efficiently on accelerator hardware and can typically exploit from reduced numerical precision, leading to

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

speedups of several orders of magnitude in specific applications, as demonstrated, for example, in [10, 34]. While these approaches are conceptually distinct from classical multigrid methods, they are increasingly explored as complementary tools in large-scale simulation workflows.

A minimally invasive use of GPUs in classical finite element simulation tools is also possible but often yields little to no acceleration [22, 23]. The obtained speedup was moderate but the work describes how a transparent implementation into an existing software can be achieved. The consequences of mixed precision arithmetic were also discussed in this work. In [21], the same authors presented a geometric multigrid method that is based entirely on sparse matrix-vector multiplications and can thus be easily and efficiently implemented on different hardware using suitable libraries. Applied to stationary linear differential equations, the solver exhibited up to a fivefold speedup relative to CPU-based systems.

In this article, we show that highly efficient adaptive finite elements with multigrid solvers can be implemented efficiently on GPU accelerators without limiting the flexibility of the method and without the large implementation effort that GPU-implementations can involve. For this, we use a matrix-based formulation that allows us to use the cuBLAS and cuSPARSE libraries [39] that are highly optimized while the code remains close to those with BLAS calls on the CPU. Hence, the CPU and GPU versions can retain the same structure and differ only in the linear algebra function calls and additional CPU-to-GPU memory transfer in the GPU version. It, nonetheless, allows for the use of custom CUDA kernels when necessary and we demonstrate that the use of simple native CUDA code can provide significant speedups for operations that do not naturally map to (sparse) linear algebra. Due to these optimizations, the largest performance bottleneck comes from the memory transfer between CPU and GPU, in particular, if system matrices are assembled on the CPU and then transferred to the GPU. In section 5 we study numerical examples, where the repeated assembly can be avoided such that matrices are copied only once for all time steps. This reduces the overhead and the remaining time for memory transfer is negligible.

In our final code, large parts of the computations have been transferred to the GPU and thus the slowdown through the transfer is also small.

We demonstrate the generality and flexibility of our approach by applying it to two linear elliptic problems, namely the transport-diffusion equation in 2d and a 3d linear elasticity problem. The discretization of these two equations results directly in a linear system of equations that can be approximated with the GPU-accelerated geometric multigrid solver. As a third example, we consider the instationary Navier–Stokes equations. For these, we first derive an explicit pressure-correction method, which can be represented by matrix-vector multiplications and a pressure-Poisson problem, the latter one being solved with the multigrid solver. Special attention is given to the nonlinearity of the Navier–Stokes equations. Through a reformulation, also this will be approximated by a matrix-vector multiplication with a precomputed sparse matrix on the GPU. For all three examples, we obtain significant speedups between 5X and 20X for our final GPU parallelizations. At the same time, the systematic use of (sparse) linear algebra leads to easily understandable and maintainable code for the GPU computations.

Outline. The remainder of the article is structured as follows. In section 2 we introduce the mathematical notation and briefly present the finite element discretization. There, we also describe the geometric multigrid method. In section 3, we will discuss accelerators such as GPUs and describe the specifics of the hardware and how they need to be reflected in the algorithms. We also introduce software libraries that

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

facilitate the use of accelerator hardware. Section 4 describes our implementation of the multigrid process based on the cuSPARSE library [39]. Finally, in section 5, we present numerical test problems and discuss the results. We conclude in section 6.

Main contributions. We describe the integration of GPU acceleration into the Gascoigne 3d general-purpose finite element library. As the software did not consider accelerator hardware in its design, the focus is on making the most efficient use of the hardware while minimizing interventions into the existing implementation. For this reason, we base our GPU acceleation on cuSPARSE and require custom kernels only for a small set of operations that cannot be expressed efficiently using linear algebra. Additionally, we present interpolation-based finite element techniques that enable us to formulate certain nonlinear terms with matrix-vector products based on sparse matrices with small stencils. This avoids numerical quadrature and extends the applicability of the cuSPARSE-based implementation to a larger class of problems. Our insight for the GPU acceleration is not specific to Gascoigne 3d and will hence likely generalize to other finite element libraries.

2. Adaptive Finite Element Discretization and Multigrid Solver.

2.1. Finite element discretization. We denote a domain by $\Omega \subset \mathbb{R}^d$, where $d = 2$ or $d = 3$ is the dimension. By Ω_h we denote a finite element mesh consisting of N_h quadrilaterals or generalized (allowing curved faces) hexahedras. The elements $T \in \Omega_h$ all arise from a reference element $\hat{T}$

$$\hat{T}_T : \hat{T} \mapsto T$$

that is the unit square in 2d and the unit cube in 3d. We will consider isoparametric finite elements, where the mapping itself comes from the finite element space. Let

$$Q^r := \{x_1^{\alpha_1} \cdots x_d^{\alpha_d},\ \alpha_i \in \mathbb{N},\ 0 \le \alpha_i \le r\}$$

be the space of polynomials of maximal degree r in each coordinate. Then, $\hat{T}_T \in [Q^r]^d$ and we define the finite element spaces of degree r as

$$V_h^{(r)} := \{\phi \in C(\bar{\Omega}) : \phi \circ T_T^{-1} \in Q^r \quad \forall T \in \Omega\}.$$

We assume shape regularity in the sense that $\|\nabla T_T\| \cdot \|\nabla T_T^{-1}\| \le c$ uniform in $h > 0$; see [44, sec. 4.2]. Structural regularity is relaxed by allowing local mesh refinement with at most 1 hanging node per face; again, see [44, sec 4.2] for the specific realization in the finite element library Gascoigne 3d [12].

Systems of partial differential equations such as the Navier–Stokes equations or elasticity will be discretized with equal-order finite elements, i.e., $\mathbf{V}_h = [V_h^{(r)}]^c$, where $c \in \mathbb{N}$ is the number of components. Gascoigne 3d combines these components locally. Taking the 3d Navier–Stokes equations as example, this means that the vector is represented as a matrix with entries $x_{i,c}$, where i refers to the grid node and c to the component ($c = 1$ pressure, $c = 2, 3, 4$ velocities). The system matrix also has double indexing. Each matrix entry A_{ij} is itself a (dense) matrix $A_{ij} \in \mathbb{R}^{n_c \times n_c}$ with $n_c = 4$ in the case of 3d Navier–Stokes. Likewise, each vector entry v_i is itself a vector $v_i \in \mathbb{R}^{n_c}$. Keeping the solution components together is beneficial in terms of cache efficiency for problems where one obtains large system matrices like Navier–Stokes ($n_c = 4$) or 3d elasticity, where $n_c = 6$ when the velocity and deformation fields are combined [13, 29]. The memory layout of this approach is not easily transferred to GPUs as standard libraries like cuSPARSE do not support such blocked matrices. Section 4.1.1 will give details.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Algorithm 2.1. Geometric multigrid solver.

Given a hierarchy of meshes Ω_l and corresponding finite element spaces $\mathbf{V}_l$ for $l = 0, \dots, L$ and an initial value $x_L^0 \in \mathbf{V}_L$. For $n = 1, \dots, N_{max}$ iterate

$$x_L^{(n)} = GMG(L, x_L^{(n-1)}, b_L)$$

where $GMG(l, x_l, b_l)$ is recursively defined as:

Step 0: Coarse mesh problem	if $l = 0$ return $A_0^{-1} b_0$
Step 1: Presmooth	$x_l' = \mathcal{S}_l(x_l, b_l)$
Step 2: Restrict residual	$d_{l-1} = \mathcal{R}_{l-1}(b_l - A_l x_l')$
Step 3: Recursive coarse mesh correction	$y_{l-1} = GMG(l-1, 0, d_{l-1})$
Step 4: Prolongate update	$x_l'' = x_l' + \mathcal{P}_{l-1} y_{l-1}$
Step 5: Postsmooth	return $\mathcal{S}_l(x_l'', b_l)$

2.2. Adaptivity and geometric multigrid hierarchy. Adaptivity of the computations is realized by means of hierarchical local mesh refinement. If an element $T \in \Omega_h$ is chosen for refinement, it is replaced by 4 elements in 2d and 8 elements in 3d. We allow a level jump of 1 for neighboring elements. The actual refinement can therefore extend further into the domain. Faces and edges on elements with a level jump therefore have inner nodes that are unknowns only on one side. These nodes are called hanging nodes and are replaced by interpolations of their direct neighbors in the finite element approach. In the original Gascoigne, hanging node data used for this interpolation is stored in a sparse structure only covering exactly the hanging nodes. As preparation for GPU parallelization, the data is now represented in a sparse matrix. This matrix $\mathbf{H}_h$ is the diagonal identity in all rows belonging to standard degrees of freedom, whereas in rows belonging to hanging nodes, the matrix contains the interpolation weights such that the matrix vector product $\mathbf{y} = \mathbf{H}_h \mathbf{x}$ performs all interpolations at once. The computational effort is $O(N)$ (N being the number of unknowns), regardless if hanging nodes are present. However, the constant is small and the overhead in relation to a treatment of only the hanging nodes is negligible. The transpose $\mathbf{H}_h^T$ plays an important role in the assembly of the matrix and variational residuals. If, for example, an integral $f_i = (f, \phi_i)$ is calculated for $i = 1, \dots, N$, all test functions ϕ_i, including those in hanging nodes, are processed first. The result vector $\mathbf{f}_h = (f_i)_i$ is then multiplied by $\mathbf{H}_h^T$ so that the test functions belonging to hanging nodes are correctly taken into account.

Mesh refinement can be based either on a priori knowledge or on a posteriori error estimates, the latter usually in the context of the dual weighted residual method [9, 13].

Successive mesh refinement results in a hierarchy of meshes. Instead of using this hierarchy for the setup of the geometric multigrid solver, we start with the finest mesh $\Omega_h =: \Omega_L$ and recursively coarsen it until we reach a coarse mesh $\Omega_H =: \Omega_0$ with small complexity N_H. The mesh Ω_H is either the coarse starting mesh or any finer mesh chosen to give an optimal balance between recursive multigrid performance and fast direct solution of a small problem. By L we denote the resulting number of multigrid layers. In each coarsening step, as many refinements as possible are taken back, yielding a hierarchy $\Omega_h = \Omega_L \succeq \Omega_{L-1} \succeq \dots \Omega_0 = \Omega_H$. To be precise, an element is coarsened, if it belongs to a group of four (in 3d eight) elements on the same mesh level that all arise from splitting the same common father element. Details are

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

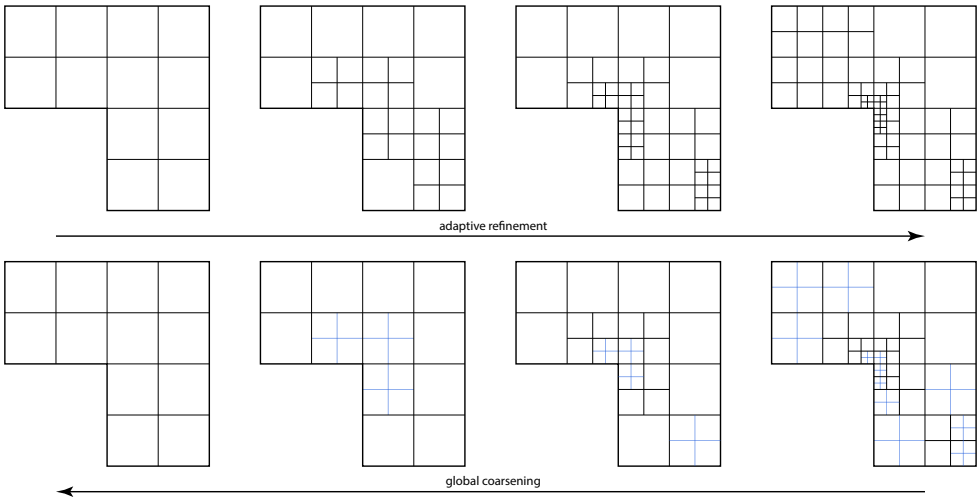


FIG. 1. *Top row: hierarchy of adaptively refined* 2:1 *meshes with hanging nodes. Bottom row: global coarsening resulting in the multigrid hierarchy. In each step as many refinements are taken back as possible (shown in blue) for a rapid decrease of the mesh complexity.*

given in [7]. We follow a global multigrid approach where each intermediate level Ω_l spans the complete domain; see [7]. The main advantage of the global multigrid is its simplicity. An approximation of the entire problem can be created on each grid level. This allows, for example, the introduction of global constraints, e.g., a normalization of the pressure, $\int_\Omega p \, \mathrm{d}x = 0$, in the context of incompressible flows or compliance with material balances $\sum_i c_i = 1$ for chemical reactions; see [8]. The disadvantage of the global approach is the greater effort required, as the multigrid smoother covers the entire grid on each level. However, if one analyses the effort in terms of the accuracy that can be achieved with adaptive grids and not in terms of the number of degrees of freedom, it becomes clear that the global approach has no significant disadvantage in practical applications [7, 14, 13].

Figure 1 provides a sketch of the adaptive mesh refinement and the global coarsening procedure that result in two different hierarchies of meshes.

The multigrid solver is the standard V-cycle. Coarse mesh problems are either solved exactly or approximated using a couple of smoothing steps. The geometric multigrid iteration is either used as linear solver or as preconditioner in a GMRES solver [47]. The GMRES solver is more robust and usually required for problems that are transport dominated or when meshes with large element aspect ratios are used.

3. Data-parallel accelerators. Accelerator coprocessors, such as GPUs from NVIDIA and AMD and TPUs from Google, are used in a wide range of applications, such as computer gaming and neural networks training. Due to their very large compute power, they also play an increasingly important role in scientific computing. The large compute power is achieved through a very high degree of parallelization as well as specialization. For example, NVIDIA's GPUs combine MIMD parallelism with up to 132 streaming multiprocessors with data-parallelism with a logical, pipelined width of 1024 and a hardware width of 32 (and 8 on the latest hardware). Additionally, significant compute power is provided by dedicated matrix-matrix multiplications engines known as tensor cores and introduced for neural network training. Accelerator coprocessors also differ from conventional processors through native hardware support

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

of lower precision data types such as half precision and even 8-bit computations on the latest generation of chips. Accelerator coprocessors as used in this work come with their own, dedicated memory hierarchy starting from a RAM, and with typically two layers of caches. To perform computations on the accelerator, data hence needs to first be transferred to the accelerator RAM through a comparatively slow memory interface.

The programming of accelerator hardware is typically challenging due to the high degree of parallelism and narrow "fast paths" on the hardware (e.g., because of smaller caches compared to CPUs) that are difficult to optimize for and can change from hardware generation to hardware generation. This is compounded by the separate memory hierarchy on the accelerator and the slow interface to CPU RAM that can easily become a bottleneck. Furthermore, native software libraries, such as CUDA and ROCm, are usually vendor specific and not standardized, with often also an incomplete documentation. This makes their use by nonexperts challenging and leads to a high maintenance effort when always the latest features should be used.

Different libraries have been proposed to aid with the development of GPU-accelerated software. OpenACC and OpenMP provide pragma-based access to the compute power of GPUs that is very simple to use but also limited in the potential for optimizations. SYCL [51] and Kokkos [54] provide a middle-ground with more complex usage but also more flexibility. Machine learning libraries such as PyTorch [41] and jax [15] follow a new paradigm for GPU programming where the application code is specified in a high-level language and a backend, such as XLA [48] or ATen [41], generates highly efficient, hardware-specific code. This concept is sometimes also referred to as domain-specific language, e.g., [31]. Recently, Triton [53] was proposed as a complement to existing machine learning libraries. It still allows for highly simplified programming of accelerators compared to CUDA and ROCm but provides more control than PyTorch and jax and hence typically yields more efficient code.

For specific applications, such as linear algebra or discrete Fourier transform, highly optimized libraries also exist. These provide very high performance while, through the targeted use case, also allow for much simpler usage than, e.g., CUDA. In our work, we will build on a linear algebra formulation of our adaptive PDE solvers. This allows us to use the cuBLAS [38] and cuSPARSE libraries [39] that provide high performance while being simple to use with a software interface that is roughly comparable to those of the conventional BLAS library. Furthermore, cuBLAS and cuSPARSE can still be combined with native CUDA code when necessary, i.e., when an operation cannot be expressed efficiently in linear algebra.

4. CUDA multigrid. In this section we discuss our extension of the adaptive finite element library Gascoigne 3d to use GPU accelerators for a wide range of computations. The extensions have been designed with two use cases in mind: first, nonstationary but linear systems of partial differential equations, where a single system matrix is reused in every time step or at least in many time steps, and, second, nonlinear problems that can be formulated in terms of fixed matrices that do not need reassembly. The second case includes a pressure-projection based solvers for the nonstationary nonlinear Navier–Stokes equations but can be extended to more complex flow problems such as the Boussinesq approximation [46]. Both problem classes have in common that matrices are assembled only once and the complete workflow can be formulated in terms of matrix vector products and the solution of linear systems of equations. Algorithms 4.1 and 4.2 show the typical workflow for the problem types. Computations highlighted in blue are completely performed on the GPU, orange marks transfer of data between GPU and CPU.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Algorithm 4.1. Nonstationary Linear Problem.

```
   Basic Initialization;                                  // init
1  Assemble Matrices;                                     // init
2  CopyToGPU(Matrix);                                     // copy
3  for time-iter do
4  |  Assemble Right Hand Side;                           // rhs
5  |  SolveLinearProblem;                                 // solve
6  end
7  CopyFromGPU(Solution);                                 // copy
```

Algorithm 4.2. Explicit Navier–Stokes Pressure Correction.

```
   Basic Initialization
   Assemble Matrices
   CopyToGPU(Matrix);                                     // copy
8  for time-iter do
9  |  Assemble Momentum right Hand Side;                  // mom-rhs
10 |  Explicit Solve Momentum Problem;                    // mom-solve
11 |  Assemble Pressure Right Hand Side;                  // pres-rhs
12 |  for GMRES iteration do
13 |  |  Multigrid Pressure ;                            // pres-solve
14 |  end
15 |  Update Pressure;                                    // pres-up
16 |  Correct Velocity
17 end
18 CopyFromGPU(Solution);                                 // copy
```

It is possible to also port the computations of general nonlinear problems to a GPU. However, keeping the flexibility and support of different discretizations and triangulations is not as easily handled for the ill-structured setup of typical finite element assemblies of system matrices and residuals in the general nonlinear case. We hence defer this to future work.

4.1. General architecture. Our principal approach for the CUDA parallelization of Gascoigne 3d is to formulate all operations in terms of (potentially sparse) matrix-vector and matrix-matrix products and to realize these with the highly optimized cuBLAS [38] and cuSPARSE [39] libraries. This approach differs from the existing implementation in Gascoigne 3d where many operations such as the handling of hanging nodes or the multigrid mesh transfer are based on the local connectivity of the degrees of freedom as function of mesh and discretization. However, this functionality can easily be realized as matrix-vector products, including for the handling of hanging nodes and all mesh transfer operations. This is, e.g., the approach taken from the outset in the deal.II finite element library [5].

All high level operations in Gascoigne 3d are performed on abstract interface classes to vectors and matrices to allow for MPI parallelization [29, 8]. This has been retained in the CUDA-version so that the high-level control flow and complex algorithms (such as Newton, multigrid, GMRES) are unchanged since they only operate on the interfaces. The approach minimizes the changes that were overall required and allows to flexibly retain both the CPU and the GPU backend.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

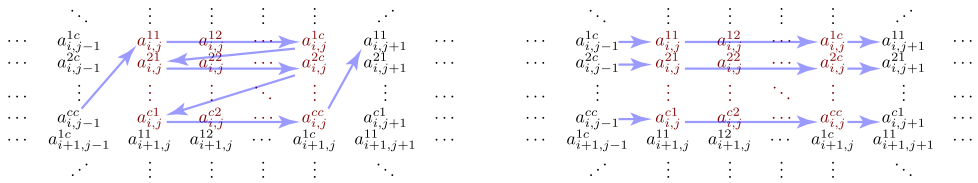


FIG. 2. *Storage of matrix on CPU (left) and GPU (right). On the CPU we employ a block-wise ordering of the matrix entries clustering the $n_c \times n_c$ components of a system of PDEs. On the GPU standard CSR format is used.*

4.1.1. Storage. Gascoigne 3d uses a block-wise memory concept in which the various components of a partial differential equation (e.g., the 3 deformation unknowns in 3d solid mechanics or the pressure and the three velocity components in fluid mechanics) are stored next to each other. Matrices are double-indexed and the outer index (i, j) refers to the mesh node yielding the entry $a_{i,j} \in \mathbb{R}^{n_c \times n_c}$ that stores (aligned in memory) the local couplings between the solution components as a matrix. In general, this setup helps to efficiently use caching [13, 29]. Using cuSPARSE this block-matrix format is not supported. While vectors are directly transferred to the GPU using the same memory layout, the inner matrix blocks must be resolved and matrices are stored in the usual CSR format. Figure 2 shows a visualization of the matrix storage concept on the CPU (left) and the GPU (right).

The representation of a matrix in GPU memory is directly allocated when the matrix is created and assembled on the CPU. This helps to better exploit the asynchronous data transfer. Our implementation is directed at problems where the matrix stays fixed such that this transfer is required just once. For vectors, the data transfer takes place when a `cudaActivate()` function is called on the object owing the data, e.g., a class steering the multigrid solver. Data is transferred back to the CPU when the corresponding `cudaDeactivate()` function is called.

4.1.2. Mapping (sparse) linear algebra to the GPU. The matrix-matrix and matrix-vector multiplications that are the essential building blocks of CUDA-Gascoigne 3d are realized with cuBLAS and cuSPARSE, which provide an interface in close analogy to BLAS. For example, the function `cuSPARSESpMV` performs for a sparse version of BLAS's gemv, i.e., it computes $\alpha\,\mathrm{op}(A)x + \beta y$, where α, β are scalar, x, y are vectors, and A is a matrix. The approach reduces the implementation effort for our GPU-parallelization substantially and ensures it is maintainable, i.e., further developments of Gascoigne 3d can be implemented without substantially efforts also for the GPU version.

4.1.3. Custom CUDA kernels. Almost all operations required for the adaptive finite element solvers of Gascoigne 3d can be expressed efficiently using linear algebra. However, we found some exceptions and performing these calculations on the CPU incurred a very high penalty due to the required data transfer between GPU and CPU. We therefore implemented small parts of the computations in native CUDA, which avoided the extra memory transfers and hence the penalty. This is facilitated by cuBLAS and cuSPARSE operating on raw CUDA pointers in device memory which can directly also be used in native CUDA. Details of the implementation are given in section 5.3.

The flexibility to combine cuBLAS, cuSPARSE and native CUDA is, in our opinion, an important feature to avoid CPU-GPU data transfer and to be able to flexibly

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

implement algorithms with CUDA-Gascoigne-3d. The total number of required custom kernels is very small and serves as guideline for measuring the effort to port further applications to the GPU.

4.1.4. Geometric Multigrid CUDA. As a concrete example, we describe in the following how geometric multigrid algorithm in Algorithm 2.1 is realized in CUDA-Gascoigne-3d. All computations are entirely performed on the GPU. Hence, data transfer overhead is incurred only at the beginning and end of the computations; see also Algorithms 4.1 and 4.2.

All steps of the algorithm involve only elementary operations that can directly be formulated in cuSPARSE or cuBLAS. Mostly, matrix-vector products must be computed, e.g., for prolongation and restriction, but also for the smoother. To avoid memory transfer between CPU and GPU, it is essential that the smoother can be performed completely on the GPU. At the moment we limit ourselves to very simple smoothers of Jacobi or block-Jacobi type that can be written as

$$x^{(l+1)} = x^{(l)} + \omega S\big(b - Ax^{(l)}\big),$$

where $\omega \in \mathbb{R}$ is a damping factor and $S \in \mathbb{R}^{n\times n}$ the smoothing operator, written as fixed matrix. More complex smoothers can, in principle, be implemented using native CUDA but we leave this to future work.

Mesh transfer operations are usually done locally. In the hierarchical setup of finite elements on quadrilateral or hexahedral meshes, prolongation of a solution to the next finer mesh is the usual embedding. To illustrate this, let K be an element of the coarse mesh Ω_l and $K_1, \dots, K_p$ be the resulting fine elements on level Ω_{l+1}, where $p = 2^d$ with d being the spatial dimension. Considering finite elements of degree r, $(r+1)^d$ unknowns are involved on mesh level l and $(2r+1)^d$ on level $l+1$. Prolongation is then by means of

$$u_i^{(l+1)} = \sum_{j=1}^{(r+1)^d} \chi_{ij} u_j^{(l)}, \quad i = 1, \dots, (2r+1)^d. \tag{4.1}$$

The $(r+1)^d \times (2r+1)^d$ coefficients χ_{ij} are the same for each mesh element. (4.1) can be written as one global matrix-vector product

$$u_h^{(l+1)} = P_l u_h^{(l)}$$

with $P_l \in \mathbb{R}^{N_{l+1}\times N_l}$ and its entries given by χ_{ij}. With the number of distinct elements in P_l being very small, the use of the matrix formulation would be sub-optimal on the CPU. On the GPU, however, it is performant and allows us to use cuSPARSE. The multigrid restriction $R_l : \Omega_{l+1} \to \Omega_l$ is the transpose of the prolongation, i.e., $R_l = P_l^T$ and hence also implemented using a sparse matrix in cuSPARSE. Similarly to the mesh transfer, also for hanging nodes we formulate the operations for averaging and distributing the values of the solution vector as sparse matrix-vector products; see section 2.2.

The coarse mesh problem in **Step 0** of Algorithm 2.1 is not solved directly. Instead, we simply apply several steps of the smoothing iteration such that no additional infrastructure for GPU parallelization is required.

4.1.5. GMRES as linear solver on the GPU. For reasons of numerical stability, e.g., on nonuniform meshes, the multigrid iteration can often not be used directly as linear solver. They are then employed as preconditioner in a GMRES iteration. Our GMRES solver follows the approach described in [47, section 6.5.3], using

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

the modified Gram Schmidt algorithm for orthogonalization and Givens rotation for solving the resulting overdetermined linear system. Due to the multigrid preconditioning, we avoid stability issues in the Gram Schmidt orthogonalization and never have to perform more than 5–10 GMRES steps. The Gram Schmidt iteration is also completely run on the GPU using cuSPARSE and cuBLAS operations. As we use a highly efficient geometric multigrid solver as preconditioner, a maximum of $n_G \leq 10$ GMRES steps is usually required. Hence, the resulting overdetermined system for finding the GMRES solution is very small and its step-wise transformation to a diagonal matrix with Givens rotations is performed on the CPU. The cost for transferring these n_G^2 entries between GPU and CPU is small. Details on GMRES are given in [47].

5. Results. In this section, we present several applications where we employ the GPU-accelerated multigrid method presented in the last section. We start with two linear elliptic problems, namely the transport-diffusion equation in 2d and a 3d linear elasticity problem. The discretization of these two equations results directly in a linear system of equations that can be approximated with the geometric multigrid solver and falls into the class of systems handled by Algorithm 4.1.

As a third example, we consider the Navier–Stokes equations. For these, we first derive an explicit pressure-correction method, which can be represented by matrix-vector multiplications and a pressure-Poisson problem, the latter one being solved with the multigrid solver. Special attention is given to the nonlinearity of the Navier–Stokes equations. Through a reformulation, also this will be approximated by a product with a precomputed sparse matrix and can easily be performed on the GPU.

All CPU computations are carried out on an AMD EPYC 7773X. For the relatively small problem sizes we consider, our scaling test in Figure 3 shows that there is only a minor benefit from using a large number of threads. The calculations are dominated by sparse matrix-vector multiplications and the efficiency is therefore limited by the memory bandwidth. This has already been shown in a detailed parallel analysis of fluid-structure interactions based on Gascoigne [20]. A transition to matrix-free finite elements—see, e.g., [30]—would thus offer the greatest potential for improving parallel efficiency. However, this would require a fundamental redesign of the current implementation. We therefore limit the OpenMP based parallel code to 8 threads for the comparisons. The GPU computations are performed on the same machine using an NVIDIA H100 PCIe card with 80 GB of VRAM. When using the GPU to

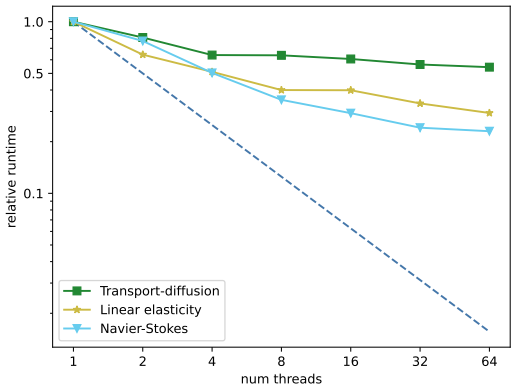


FIG. 3. *Strong scaling of the time-stepping parts of the largest problems described in sections* 5.1*,* 5.2*, and* 5.3*, with up to* 2,000,000 *degrees of freedom on an AMD EPYC* 7773*X CPU.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

accelerate the multigrid solver, those parts of Gascoigne that reside on the CPU still run on 8 parallel threads. We always indicate wall-clock times and separately specify the different contributions of Algorithms 4.1 and 4.2.

5.1. Transport-diffusion equation. Let $\Omega \in (0,1)^2$ and $I = (0,T]$ with $T = 2$. On $I \times \Omega$ we find θ subject to the following transport-diffusion problem

$$\partial_t \theta - \lambda \Delta \theta + (\boldsymbol{b} \cdot \nabla)\theta = f \qquad \text{in } (0,T] \times \Omega, \tag{5.1}$$

$$\theta = \theta_b \qquad \text{on } (0,T] \times \partial\Omega, \tag{5.2}$$

$$\theta(0,x,y) = \theta_0 \qquad \text{in } 0 \times \Omega, \tag{5.3}$$

where $\lambda = 0.01$ and $\boldsymbol{b} = (0,-1)^T$. The source term f, boundary condition θ_b and initial condition θ_0 are chosen such that (5.1)–(5.3) have the exact solution

$$\theta_{ex}(t,x,y) = \exp\left(-\frac{1}{4}(m(t,x)^2 + m(t,y)^2)\right), \quad m(t,z) = \frac{1}{2} + \frac{1}{4}\cos\left(\frac{\pi}{2}t\right) - z.$$

We consider the backward Euler method with a time-step size $\Delta t = 0.02$. Uniform quadrilateral spatial meshes with a range of sizes given by $h = 2^{-n}$ where $n \in \{7,\dots,10\}$ are used to demonstrate the scaling efficiency of the GPU accelerated version. Hence, on the coarsest mesh with $h^{-2} = 2^{2n}$ elements, the problem has $(2^7+1)^2 = 16,641$ degrees of freedom whereas the finest mesh comprises $(2^{10}+1)^2 \approx 1,000,000$ degrees of freedom. In each of the $T/\Delta t = 100$ time steps, a linear system must be solved. To ensure the robustness of the method, we use the GMRES solver, preconditioned with multigrid, to approximate the linear problems, cf. section 4.1.5. Figure 4 shows the wall-clock times comparing the CPU and GPU implementations on the finest mesh level. The labels `init`, `rhs`, and `solve` refer to the initialization (mainly assembly of system matrix), to the computation of the right hand side, and to the actual solution, respectively. Matrix assembly and the computation of the right hand side involve numerical quadrature over the mesh elements which is not ported to the GPU. Hence, no speedup is observed, GPU timings can even be higher, as copying the matrix to the GPU involves some overhead; see also Figure 5. The time for solving the linear problems, however, is reduced from 304.75 s on the CPU to 9.0 s on the GPU, a factor of about 34; see also Table 1 for the raw

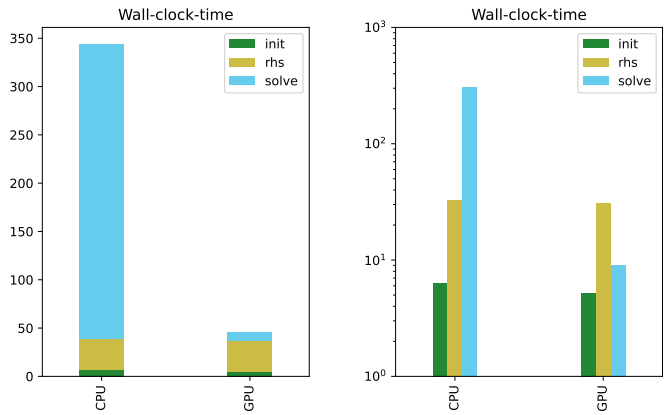


FIG. 4. *Transport-diffusion equation: Wall-clock times CPU vs. GPU on the finest mesh level with* 1 000 000 *degrees of freedom (left: linear scale, right: log scale).*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

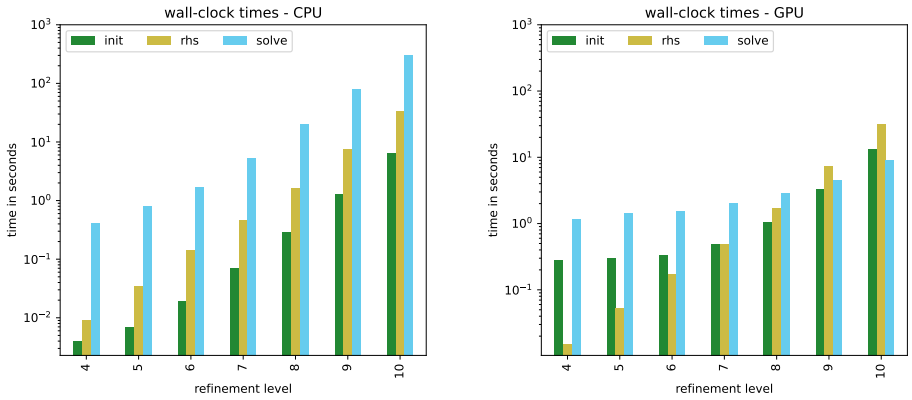


FIG. 5. *Transport-diffusion equation: Wall-clock times CPU vs. GPU on a sequence of uniformly refined meshes.*

TABLE 1
Transport-diffusion problem: wall clock times on CPU and GPU in seconds.

mesh level	DOFs	CPU				GPU			
		`init`	`rhs`	`solve`	`sum`	`init`	`rhs`	`solve`	`sum`
7	16 641	0.07	0.46	5.33	5.87	0.35	0.49	2.05	2.88
8	66 049	0.28	1.66	20.01	21.96	0.54	1.69	2.85	5.08
9	263 169	1.29	7.41	80.63	89.32	1.41	7.2	4.47	13.08
10	1 050 625	6.36	32.9	304.75	344.0	5.21	31.16	9.0	45.37

data. The geometric multigrid solver is robust on all meshes such that the number of linear steps varies only slightly between the mesh levels. For large problems, the nonaccelerated assembly of the right hand side `rhs` hence becomes dominant. In section 5.3 we describe how this term can be efficiently transferred to the GPU by using mass lumping.

In Figure 5 we visually analyze the performance of the CPU (using 8 threads) and GPU version of Gascoigne 3d on the sequence of uniformly refined meshes ranging from $h = 2^{-4}$ to $h = 2^{-10}$. While the scaling of all components `init, rhs` and `solve` is linear on the CPU, the GPU-accelerated linear solver `solve` benefits from larger problems. This is in line with theoretical considerations since the very large degree of parallelism and the very deep pipelining on the GPU requires large problems to fully utilize the computational units.

5.2. Linear elasticity. Second, we consider the linear elasticity equation

$$\partial_t^2 \boldsymbol{u} - \operatorname{div} \boldsymbol{\sigma}(\boldsymbol{u}) = \boldsymbol{f} \qquad \text{in } (0,T] \times \Omega, \tag{5.4}$$

on the domain $\Omega = (0,1)^3$ in the time interval $I = [0, 2.5]$. By $\boldsymbol{u}$ we denote the displacement, $\boldsymbol{f} = (0,-1,0)^T$ is the right hand side vector, the stress tensor is

$$\boldsymbol{\sigma}(\boldsymbol{u}) = \lambda tr(\boldsymbol{\epsilon}(\boldsymbol{u}))I + 2\mu\boldsymbol{\epsilon}(\boldsymbol{u}), \quad \boldsymbol{\epsilon}(u) = \frac{1}{2}(\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T),$$

where I is the identity tensor, tr is the trace operator on a tensor and $\lambda = 8 \cdot 10^4$ and $\mu = 2 \cdot 10^4$ are the Lamé parameters. Equation (5.4) can be written as two coupled PDEs with first order time derivatives,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

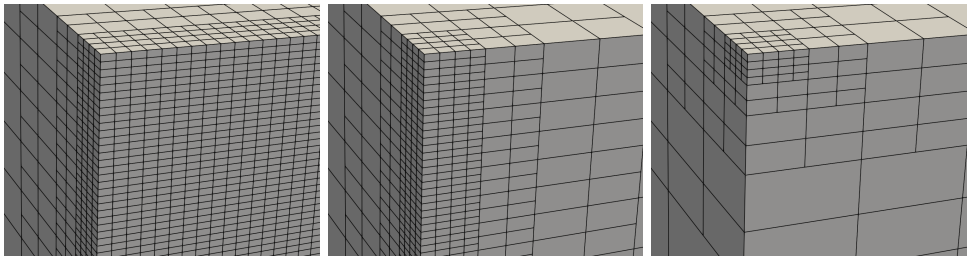


FIG. 6. *Visualization of the adaptive meshes. From left to right: Refinement towards one face, towards an edge, and towards one corner. The fraction of hanging nodes serving as measure for the unstructuredness increases from the left to the right.*

TABLE 2
Elasticity problem: Number of mesh nodes on different mesh levels (first subrow) and fraction of nodes that are hanging nodes (second subrow). From top to bottom we show the different adaptive refinement types; see Fig. 6.

Mesh levels	1	2	3	4	5	6
Face	729	2 925	11 281	43 861	172 505	683 741
	0.00%	4.92%	6.10%	6.33%	6.45%	6.46%
Edge	729	1 881	4 129	8 569	17 393	34 985
	0.00%	7.66%	10.3%	11.4%	11.9%	12.2%
Vertex	729	1 333	1 937	2 541	3 145	3 749
	0.00%	8.10%	11.2%	12.8%	13.7%	14.4%

$$\partial_t \boldsymbol{u} - \boldsymbol{v} = \boldsymbol{0} \qquad \text{in } (0,T] \times \Omega \tag{5.5}$$

$$\partial_t \boldsymbol{v} - \operatorname{div} \boldsymbol{\sigma}(\boldsymbol{u}) = \boldsymbol{f} \qquad \text{in } (0,T] \times \Omega, \tag{5.6}$$

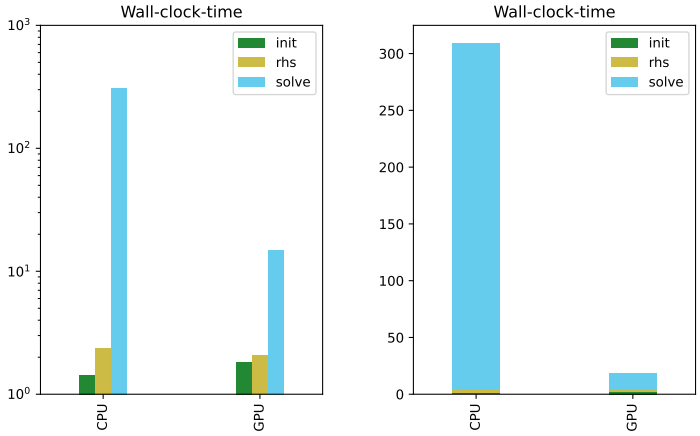
with homogenous Dirichlet boundary conditions and zero initial conditions for $\boldsymbol{u}$ and $\boldsymbol{v}$. We use this test case to illustrate the performance of the CUDA multigrid solver on adaptively refined meshes; see Figure 6 for a visualization of the used mesh. Adaptive refinement is not driven in a problem specific way here but we consider typical cases of adaptive meshing, namely refinement towards a complete face of the box, refinement towards an edge, and refinement towards a vertex. This mimics resolving singularities that have a 2d pattern (face), a 1d pattern (edge) and a 0d pattern (vertex). Table 2 lists the number of mesh nodes and the fraction of mesh nodes that are hanging. The number of degrees of freedom is six times this number of mesh nodes as we have 3 deformation and 3 velocity components. Hanging nodes do not improve the approximation property but instead disrupt the structure of the problem by distorting the sparsity pattern of the matrix.

For time discretization of (5.5)–(5.6) the backward Euler method is used with a step size of $\Delta t = 0.025$ resulting in $2.5/\Delta t = 100$ time steps.

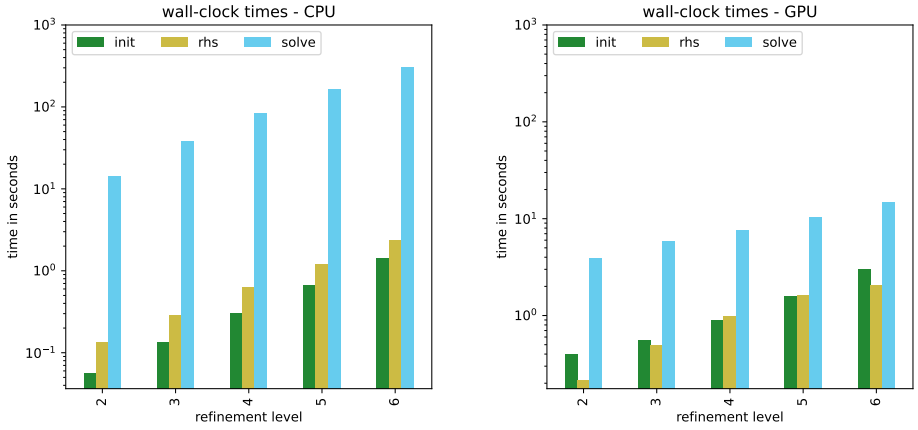
We provide a detailed discussion of the intermediate case, the refinement towards one edge. Figure 7a gives the timings on the finest mesh level (34 985 nodes, hence 209 910 degrees of freedom, about 25 600 of them in hanging nodes) for the CPU version and the GPU version. As for the transport-diffusion equation, times for initialization `init` and right hand side `rhs` do not change, as these parts are not implemented on the GPU. The computation time for the linear solver is, however, drastically reduced; see Table 3b. On the finest mesh, the solution time reduces from 306 s to 15 s, i.e., by a factor of 20. As the time for matrix assembly (`init`) and computing the right hand side (`rhs`) are negligible, the overall speedup is still close to 17 (compared to about 8

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy



(a) Refinement towards edge on mesh level 6, logarithmic scale (left) and linear scale (right).



(b) Refinement towards edge on a sequence of locally refined meshes.

FIG. 7. *Elasticity problem. Wall-clock times on CPU and GPU.*

for the transport-diffusion problem on a much finer mesh). As for the transport-diffusion problem, the initialization (`init`) may require more time on the GPU as this includes the cost for copying the matrix from the CPU to the GPU. Also, since the effort is negligible for this problem, the right hand side has been assembled on the CPU for simplicity. Therefore, the timinigs on the GPU include overhead for copying the data to the GPU.

Figure 7b shows the scaling of the implementation with respect to the refinement. Note that the meshes are refined locally. Therefore, the number of multigrid levels increases in each step but the overall number of unknowns grows only slowly (not by a factor of 8 that would be expected on a 3d hexahedral mesh). Hence more and more unstructured features and hanging nodes appear on higher mesh levels. Due to the slow growth in the number of degrees of freedoms, the run-times on CPU and GPU increase only slowly with each refinement level; see also Table 2. The results show very good efficiency of the GPU multigrid solver on locally refined meshes with no

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

TABLE 3
Elasticity problem: measured values for the wall clock times on CPU and GPU on different refinement types.

mesh level	DOFs	CPU				GPU			
		`init`	`rhs`	`solve`	`sum`	`init`	`rhs`	`solve`	`sum`
2	17 550	0.17	0.22	41.78	42.17	0.45	0.34	7.52	8.32
3	67 686	0.7	0.89	174.96	176.55	0.99	1.27	14.65	16.92
4	263 166	2.99	3.56	1031.13	1037.68	3.45	2.98	38.22	44.64
5	1 035 030	11.91	14.01	4431.94	4457.85	12.56	10.78	122.57	145.9

(a) Refinements towards a face of the domain.

mesh level	DOFs	CPU				GPU			
		`init`	`rhs`	`solve`	`sum`	`init`	`rhs`	`solve`	`sum`
3	24 774	0.13	0.29	38.22	38.64	0.43	0.49	5.93	6.85
4	51 414	0.3	0.64	82.92	83.86	0.61	0.99	7.64	9.24
5	104 358	0.66	1.19	165.29	167.14	1.0	1.62	10.38	13.01
6	209 910	1.42	2.38	305.58	309.39	1.82	2.07	14.7	18.59

(b) Refinement towards an edge of the domain.

mesh level	DOFs	CPU				GPU			
		`init`	`rhs`	`solve`	`sum`	`init`	`rhs`	`solve`	`sum`
3	11 622	0.07	0.13	13.04	13.24	0.37	0.22	4.73	5.33
4	15 246	0.11	0.17	17.68	17.96	0.42	0.31	6.12	6.85
5	18 870	0.16	0.21	22.7	23.07	0.48	0.36	7.09	7.94
6	22 494	0.21	0.25	28.21	28.67	0.56	0.42	7.9	8.88

(c) Refinement towards a vertex.

negative impact of having more and more hanging nodes and less regular structure in the problem.

Next we compare the performance of the GPU implementation on the different types of adaptive mesh shown in Figure 6 with number of unknowns indicated in Table 2. The complexity of these mesh types is very different. While the number of nodes increases like 4^l (l is the mesh level) for face refinement, which is typical for 2d problems, and 2^l for the edge refinement, the typical behavior for 1d problems, the scaling is much slower for the vertex-refinement.

Figure 8 shows that the efficiency of the GPU parallelization is robust. On the finest mesh level belonging to the vertex-case in Table 3c, the computational time of the solver only drops from 29 s on the CPU to 8 s on the GPU, a factor of about 3.5. However, this problem has just 25 000 unknowns and is too small to properly utilize a GPU. It is important to note that the GPU parallelization does not lead to an overhead at any point, which negatively affects the entire computing time.

5.3. Navier–Stokes equations. As third example, we consider the time-dependent Navier–Stokes equations in their dimensionless form

$$\partial_t \boldsymbol{u} - \frac{1}{\mathrm{Re}}\Delta \boldsymbol{u} + \operatorname{div}(\boldsymbol{u} \otimes \boldsymbol{u}) + \nabla p = 0 \qquad \text{in } (0,T] \times \Omega, \tag{5.7}$$

$$\operatorname{div} \boldsymbol{u} = 0 \qquad \text{in } (0,T] \times \Omega. \tag{5.8}$$

By $\boldsymbol{u} : (0,T] \times \Omega \to \mathbb{R}^3$ we denote the velocity field and by $p : (0,T] \times \Omega \to \mathbb{R}$ the scalar pressure. In our numerical experiment, we use the driven cavity benchmark problem on the time interval $I = (0, 12]$ and on the three dimensional domain $\Omega = (0,1) \times (0,1) \times (0,2)$. We choose the Reynolds number $\mathrm{Re} = 10^3$ and the boundary conditions

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

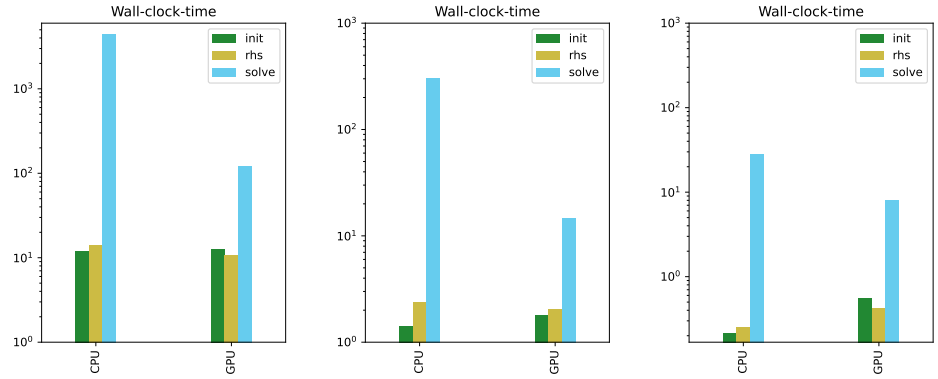


FIG. 8. *Comparison of the refinement types (from left to right):* 5 *refinements towards one face (left),* 6 *refinements towards one edge (middle),* 6 *refinements towards one vertex (right).*

$$\boldsymbol{u}(t,x,y,z) = \begin{cases} (0,1,0)^T & \text{if } x = 1, \\ (0,0,0)^T & \text{otherwise.} \end{cases}$$

Homogenous initial condition $\boldsymbol{u}(0,x,y,z) = \boldsymbol{u}_0 := \mathbf{0}$ hold at time $t = 0$. We set $\boldsymbol{V} = [H_0^1(\Omega)]^d$ and $Q := \{q \in L^2(\Omega) : \int_\Omega q \,\mathrm{d}x = 0\}$. The weak formulation of the time-dependent Navier–Stokes equations in its semidiscrete form is

$$(\partial_t \boldsymbol{u}, \boldsymbol{\chi}) + \frac{1}{\mathrm{Re}}(\nabla \boldsymbol{u}, \nabla \boldsymbol{\chi}) - (\boldsymbol{u} \otimes \boldsymbol{u}, \nabla \boldsymbol{\chi}) - (p, \operatorname{div} \boldsymbol{\chi}) = (\boldsymbol{f}, \boldsymbol{\chi}) \qquad \forall \boldsymbol{\chi} \in \boldsymbol{V}, \tag{5.9}$$

$$(\operatorname{div} \boldsymbol{u}, \xi) = 0 \qquad \forall \xi \in Q. \tag{5.10}$$

Here, we used the divergence form of the convective term once integrated, i.e.,

$$\big(\boldsymbol{u} \cdot \nabla \boldsymbol{u}, \boldsymbol{\chi}\big) = \big(\operatorname{div}(\boldsymbol{u} \otimes \boldsymbol{u}), \boldsymbol{\chi}\big) = -\big(\boldsymbol{u} \otimes \boldsymbol{u}, \nabla \boldsymbol{\chi}\big).$$

This representation will be crucial for an efficient realization on the GPU, since in the discrete setting $(\boldsymbol{u} \otimes \boldsymbol{u}, \nabla \boldsymbol{\chi})$ can be approximated by sparse matrix vector products, as will be explained shortly.

For the spatial discretization of (5.9) and (5.10), we employ finite elements with the inf-sup stable $\mathbb{Q}_1$-iso-$\mathbb{Q}_2/\mathbb{Q}_1$ pair. We consider a shape regular mesh Ω_h of the domain and its equidistant refinement $\Omega_{h/2}$. The discrete spaces $\boldsymbol{V}_h \subset \boldsymbol{V}$ and $Q_h \subset Q$ consist of $\mathbb{Q}_1$ polynomials in each cell of $\Omega_{h/2}$ and Ω_h, respectively. The semidiscrete problem is obtained with $\boldsymbol{u}_h, \boldsymbol{\chi}_h \in \boldsymbol{V}_h$ and $p_h, \xi_h \in Q_h$. Application of an implicit time discretization for this semidiscrete problem requires fixed point iterations due to the nonlinear transport term. Each iteration step for linearization requires thereby to resolve a nonsymmetric, indefinite system matrix owing to the transport term and the saddle point structure that arises from velocity pressure coupling. Here, we follow an semi-implicit approach where the momentum equation is discretized explicitly.

Due to better stability and the advantageous diagonal structure, we employ a lumped mass matrix, i.e., we approximate

$$(\boldsymbol{v}_h, \phi_i)_{i=1}^N \approx M_h^l \boldsymbol{v}, \tag{5.11}$$

where $\phi_i^h : \Omega \to \mathbb{R}$ are the $\mathbb{Q}_1$ basis functions, $\boldsymbol{v} \in \mathbb{R}^{N_{nodes} \times 4}$ is the coefficient vector, and M_h denotes the lumped mass matrix given by

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

$$M_h^l = \mathrm{diag}(m_1, \ldots, m_{N_{dof}}), \quad m_i = \sum_j (\phi_j, \phi_i).$$

Moreover, we apply a mass lumping approach also in the nonlinear term, which enables us to implement the convective term as a matrix-vector product with a pre-assembled sparse matrix; see also [28]. In particular, let $\mathcal{I}_h : \boldsymbol{V} \to \boldsymbol{V}_h$ be the nodal interpolation operator. For each $\boldsymbol{v}_h$, $\boldsymbol{\chi}_h \in \boldsymbol{V}_h$, we approximate

$$\begin{aligned}(\mathrm{div}\,(\boldsymbol{v}_h \otimes \boldsymbol{v}_h), \boldsymbol{\chi}_h) &= -(\boldsymbol{v}_h \otimes \boldsymbol{v}_h, \nabla \boldsymbol{\chi}_h) \\ &\approx -(\mathcal{I}_h(\boldsymbol{v}_h \otimes \boldsymbol{v}_h), \nabla \boldsymbol{\chi}_h) =: n(\boldsymbol{v}_h \otimes \boldsymbol{v}_h, \boldsymbol{\chi}_h).\end{aligned} \tag{5.12}$$

In finite element notation, this discrete approximation amounts to

$$\mathcal{I}_h(\boldsymbol{v}_h \otimes \boldsymbol{v}_h) = \sum_i (\boldsymbol{v}_i \otimes \boldsymbol{v}_i)\phi_i^h \approx \left(\sum_i \boldsymbol{v}_i \phi_i^h\right) \otimes \left(\sum_j \boldsymbol{v}_j \phi_j^h\right).$$

Note that in the interpolation nodes it holds that

$$\mathcal{I}_h(\boldsymbol{v}_h \otimes \boldsymbol{v}_h)(\boldsymbol{x}_k) = \boldsymbol{v}_h(\boldsymbol{x}_k) \otimes \boldsymbol{v}_h(\boldsymbol{x}_k).$$

The error caused by mass lumping in the zeroth-order term (5.11) and the convective term (5.12) are of the same order as the polynomial approximation error and therefore do not affect the error asymptotics. This approach is sometimes denoted as 'fully practical finite element method' [6] as it allows for error estimates that reflect the full error including numerical quadrature. We have recently published the complete error analysis for an explicit pressure correction method based on this approximation [28]. Both theoretical analysis and numerical examples show that the explicit treatment based on the interpolation of the nonlinearity does not lead to any additional error.

For the convective term we assemble three sparse matrices

$$C_{h,\{x,y,z\}} = (\phi_j, \partial_{\{x,y,z\}} \phi_i)_{i,j=1}^{N_{nodes}}$$

for the three components x, y, z and compute the three vectors

$$\boldsymbol{v}_{i,c}^1 = \boldsymbol{u}_{i,1}\boldsymbol{u}_{i,c},\ \boldsymbol{v}_{i,c}^2 = \boldsymbol{u}_{i,2}\boldsymbol{u}_{i,c},\ \boldsymbol{v}_{i,c}^3 = \boldsymbol{u}_{i,3}\boldsymbol{u}_{i,c} \tag{5.13}$$

for $c = 1, 2, 3,\ i = 1, \ldots, N_{nodes}$. The residual can then be evaluated using matrix-vector products as

$$-\left(\boldsymbol{u}_h \otimes \boldsymbol{u}_j, \nabla \boldsymbol{\chi}\right)_{i=1}^{N_{nodes}} \approx -C_{h,x}\boldsymbol{v}^1 - C_{h,x}\boldsymbol{v}^2 - C_{h,x}\boldsymbol{v}^3. \tag{5.14}$$

The computation of the component-wise products in 5.13 is not a standard operation that can be expressed in linear algebra. We will therefore require custom kernels to assemble these vectors efficiently on the GPU.

5.3.1. Explicit projection solver. Let $k > 0$ be a time step and $N = T/k$. We set $t_n := n \cdot k$ for $0 \le n \le N$ and

$$A((\boldsymbol{u}, p); (\boldsymbol{\chi}, \xi)) := -\nu(\nabla \boldsymbol{u}, \nabla \boldsymbol{\chi}) + n(\boldsymbol{u} \otimes \boldsymbol{u}, \boldsymbol{\chi}) - (p, \mathrm{div}\,\boldsymbol{\chi}) + (\xi, \mathrm{div}\,\boldsymbol{u}).$$

The fully discrete solution algorithm we employ solves the momentum equation explicitly and updates the pressure field by solving a Poisson problem (see Alg. 5.1). Hence, we introduce the solution space $S = H^1(\Omega)$ and its discrete counterpart S_h that consists of $\mathbb{Q}_1$ polynomials on each cell T of Ω_h. This type of predictor-corrector methods for approximating incompressible flows requires an initial pressure field which must be calculated from the Poisson equation if the right hand side and initial velocity are not zero

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

$$(\nabla p_h^0, \nabla\varphi) = -(f^0, \nabla\varphi) + ((\boldsymbol{u}_0 \cdot \nabla)\boldsymbol{u}_0, \nabla\varphi) + (\nu\Delta\boldsymbol{u}_0, \varphi)_\Gamma \quad \forall\varphi \in S_h. \tag{5.15}$$

For justification and well-posedness of this equation, see [27]. Moreover, note that the velocity solutions $\boldsymbol{u}_h^m$ are so-called predictor solutions. The corrector velocities were eliminated as described in [24]. The complete error analysis for this fully explicit variant of a pressure correction scheme is found in [28] and its extension to the Boussinesq approximation is described in [46].

We set $\Delta t = 10^{-4}$ and consider $(N_x,\ N_y,\ N_z) = (32, 32, 64)$ elements in each coordinate direction. The mesh coordinates $(x_i,\ y_j,\ z_k)$ for $i \in \{0, \cdots, N_x\}$, $j \in \{0, \cdots, N_y\}$ and $k \in \{0, \cdots, N_z\}$ are defined as

$$\begin{aligned} x_i &= \frac{1}{2}\left(1 + \cos\left(\frac{i \cdot \pi}{N_x}\right)\right), && i \in \{0, \cdots, N_x\}, \\ y_j &= \frac{1}{2}\left(1 + \cos\left(\frac{j \cdot \pi}{N_y}\right)\right), && j \in \{0, \cdots, N_y\}, \\ z_k &= 1 + \sin\left(\frac{(k - N_z) \cdot \pi}{2N_z}\right), && k \in \{0, \cdots, N_z\}. \end{aligned}$$

The meshes contain anisotropic elements. Therefore, to obtain better robustness, we again embed the geometric multigrid solver as preconditioner in the GMRES method.

5.3.2. GPU realization. The explicit pressure correction scheme in Algorithm 5.1 consists largely of the same parts that have already been used for the nonstationary linear problems that have been presented above. The multigrid method is fully run on the GPU as well as all parts of the GMRES solver that work with global matrices and vectors scaling with the dimension of the finite element space. **Step 1** and **Step 3** are explicit and only require sparse matrix vector products as well as inversion of the lumped mass matrix. This however is already stored as a diagonal matrix containing the inverse elements such that standard cuSPARSE methods can be used.

An operation that is not easily expressed in cuBLAS or cuSPARSE is the local assembly of the outer products $\mathbf{u}_i \otimes \mathbf{u}_i \in \mathbb{R}^{3\times 3}$ in (5.13). To understand the effect this has on the performance, the numerical results below show two versions: one, simply denoted by (GPU), where a custom CUDA kernel is used to evaluate (5.13) directly on

---

Algorithm 5.1. The fully discrete solution procedure for the Navier–Stokes equations.

Given $\boldsymbol{u}_0$. Set $q_h^0 := 0$ and choose $p_h^0 \in Q_h$ s.t. (5.15) is fulfilled. For $m = 1, \ldots, N$ calculate

Step 1: Find $\boldsymbol{u}_h^m \in \boldsymbol{V}_h$ such that

$$\tfrac{1}{k}(\boldsymbol{u}_h^m, \boldsymbol{\chi})_* = (\boldsymbol{f}^{m-1} + \tfrac{1}{k}\boldsymbol{u}_h^{m-1}, \boldsymbol{\chi})_* - A((\boldsymbol{u}_h^{m-1}, p_h^{m-1} + q_h^{m-1}); (\boldsymbol{\chi}, 0)) \quad \forall\boldsymbol{\chi} \in \boldsymbol{V}_h.$$

Step 2: Find $q_h^m \in S_h$ such that

$$(\nabla q_h^m, \nabla\varphi) = -\frac{1}{k}(\operatorname{div}\boldsymbol{u}_h^m, \varphi) \quad \forall\varphi \in S_h.$$

Step 3: Find $p_h^m \in Q_h$ such that

$$(p_h^m, \varphi)_* = (p_h^{m-1} + q_h^m, \varphi)_* - (\nu \operatorname{div}\boldsymbol{u}_h^m, \varphi) \quad \forall\varphi \in Q_h.$$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

the GPU and an intermediate version, denoted as (GPU*), where we compute these products on the CPU. This requires transferring the solution vector back and forth whenever this matrix-vector product is required. We add this intermediate results to demonstrate the importance of minimizing memory transfers.

We also use custom kernels to efficiently compute the products with the rectangular matrices corresponding to the discretization of the gradient of the pressure and the divergence of the velocity that is required in preprocessing. Appendix A gives details on the implementation and in particular on the required changes in the user code to employ GPU parallelization and on the integration of custom kernels.

5.3.3. Numerical Results. Figure 9 and Table 4 show the overall wall-clock times for the three implementations: CPU using 8 parallel threads, GPU* solely with cuSPARSE, and the GPU version with custom kernels as described in section 5.3.2.

We split the timings into several components. `momentum` corresponds to **Step 1** of Algorithm 5.1. Here, `mom-rhs` is the assembly of the right hand side, which we further split into `mom-rhs-nonlin`, the multiplication with C_h, see (5.14), into `mom-rhs-p` which is a multiplication with a rectangular matrix acting on the pressure and finally into `mom-rhs-visc` that covers the matrix-vector product representing the viscous term. By `mom-solve` we denote the multiplication with the inverse mass matrix. The assembly of the right hand side combined in `mom-rhs` is the most costly part in the solution of the momentum equation. Porting it to the GPU reduces the computational cost by a factor of 6.

The column denoted as GPU* is purely based on cuSPARSE and no custom kernels are used. In particular, cuSPARSE does not provide functionality to compute the node-wise products (5.13) which are therefore still computed on the CPU. This causes the substantial costs for the memory transfer listed as `copy`. By introducing a custom CUDA kernel to directly compute (5.13) on the GPU, a substantial further reduction of computational time is achieved; see also section 5.3.2. A further custom kernels is used to optimize the transfer between the scalar pressure space and the vector-valued velocity space required in `mom-rhs-p`. The column labeled GPU lists the timings based on these additional custom kernels. The costs for memory transfer and hence the overall runtime is hereby substantially reduced giving a speedup (for the momentum part `momentum`) by a factor of nearly 40; see Table 4.

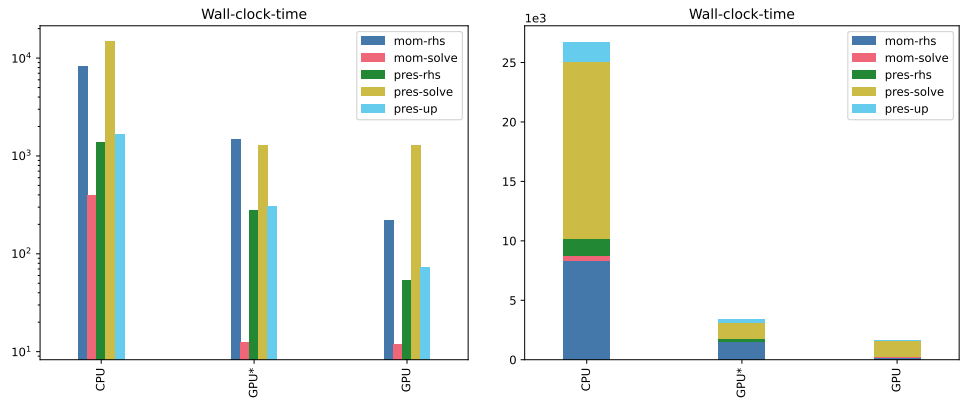


FIG. 9. *Navier–Stokes problem: Wall-clock times CPU vs. GPU (GPU* is without custom kernels), log scale (left) and linear scale (right). Table 4 gives the raw numbers and it also depicts the time required for transfer of data between CPU and GPU.*

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

TABLE 4

Navier–Stokes problem: Wall-clock times (in seconds) shown in Fig. 9. CPU is the times on a CPU using 8 threads. GPU is the result of the implementation using cuSPARSE without further custom kernel code. GPU includes further optimizations described in section 5.3.2 using custom kernels. For both GPU timings, the code remaining on the CPU is run using 8 parallel threads. The timings for Step* **1** *(momentum equation),* ***Step* 2** *(pressure Poisson problem) and* ***Step* 3** *(pressure update) are split into their contributions. Furthermore we indicate the times for copying data between CPU and GPU. These times are already included in* ***Step* 1**, ***Step* 2** *and* ***Step* 3**.

	CPU	GPU*	GPU
`mom-rhs-nonlin`	5255.3 |	1133.0 |	143.1 |
`mom-rhs-p`	1648.0 |	327.0 |	37.3 |
`mom-rhs-visc`	1278.6↓	36.4↓	37.3↓
`mom-rhs`	8331.4 |	1497.5 |	218.9 |
`mom-solve`	401.9↓	12.4↓	11.8↓
`momentum`	**8733.7**	**1510.3**	**231.1**
`pres-rhs`	1385.8 |	279.1 |	53.2 |
`pres-solve`	14967.4↓	1297.6↓	1303.4↓
`pres`	**16353.5**	**1576.8**	**1356.7**
`pres-up.rhs`	1403.6 |	285.3 |	49.3 |
`pres-up.solve`	234.2↓	19.2↓	22.5↓
`pres-up`	**1667.0**	**305.0**	**72.2**
`sum`	**26754.2**	**3392.0**	**1660.0**
`copy`	–	1437.6	7.8

`pres` covers **Step 2** of Algorithm 5.1, the pressure Poisson problem. Again, we split the timings into `pres-rhs` for the assemble of the right hand side, a sparse matrix vector product computing the divergence of the velocity prediction and `pres-solve` which is the actual inversion of the Laplace problem using the GMRES multigrid solver. `pres-up` finally corresponds to **Step 3**, the explicit update of the pressure. Here, `pres-up-rhs` is the assemble of the right hand side and `pres-up-solve` the inversion of the lumped pressure space mass matrix.

Both `pres-rhs` and `pres-up.rhs` require matrix-vector multiplications with rectangular matrices. The timings listed under GPU* require a memory transfer to the CPU to map between pressure and velocity space (similar to `mom-rhs-p` described above). Acceleration is achieved by again using a custom kernel to directly work on the GPU.

Memory is only transferred for initialization (first copy of matrices and vectors) as well as for the GMRES solver. Here however, only single floats, e.g., results of scalar products or short matrices used in the GMRES orthogonalization must be copied. Their dimension is in the order of number of GMRES steps (always less than 10). By combining all optimizations the overall runtime for the $8/\Delta t = 40\,000$ time steps is reduced from about 26 800 s (about 7.5 h) using 8 CPU cores to about 1 700 s (about 30 min) with the GPU, i.e., we obtain a speedup by a factor of approximately 16.

5.4. Limitations. In order to achieve a significant speedup with the GPU, all the required data needs to be kept on the device. Because of the smaller size of GPU-RAM compared to CPU-RAM, the maximum problem size that can be solved with the GPU implementation is more limited. The device memory usage of the presented examples is shown in Figure 10. On an H100 GPU, the limit for the 3d Navier–Stokes problem is of the order 10^7. One potential direction to address this is a multi-GPU system. A challenge is thereby to load-balance between the GPUs with nonuniform mesh refinement. We plan to investigate this in more detail in future work.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

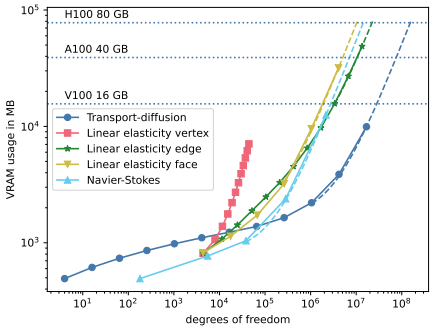


FIG. 10. *GPU memory usage depending on the degrees of freedom of the solution. The problem size is increased by repeated mesh refinement as described in sections* 5.1 *and* 5.2. *For Navier–Stokes, the mesh is also refined uniformly. Dashed lines are linear extrapolations from the respective largest three sizes. Dotted lines indicate the available memory on* 3 *generations of data center GPUs.*

For speed, the bottleneck of the GPU implementation is the memory bandwidth. The increase in bandwidth by a factor of 10 from the DDR4 RAM accessed by the CPU to the HBMe2 memory utilized on the H100 GPU (2TB/s) is complemented by a similar increase in potential compute throughput. An analysis using NVIDIA's Nsight profiler shows that roughly 2/3 of the GPU cycles are spent on sparse matrix-vector products, which have an arithmetic intensity of 0.39FLOP/byte (0.15FLOP/byte for the transposed variant) and therefore achieve just $\frac{1}{32}$ ($\frac{1}{83}$) of the peak double-precision performance. As part of cuSPARSE, the kernels are already highly optimized with a memory throughput of 91% (82%) and the low arithmetic intensity is inherent to the computational problem. Major improvements would therefore require fusing some of the roughly 3500 kernel calls needed for a single time-step together in order to reuse the data. This is not practical with the current approach that heavily leverages the cuSPARSE library. All the custom kernels together use less than 3% of the GPU time. While some of these could be eliminated by assembling rectangular block matrices in Gascoigne 3d, the speedup would therefore be negligible. The most promising direction for further optimizations is therefore a switch to single precision, since this effectively doubles the arithmetic intensity.

6. Conclusions. In this paper, we presented a GPU parallelization of the adaptive finite element library Gascoigne 3d. Our implementation uses primarily the cuBLAS and cuSPARSE libraries, which directly map dense and sparse linear algebra operations that arise as part of the adaptive finite element computations to the GPU. The use of cuBLAS and cuSPARSE covers most cases of relevance and leads to code with only small differences between CPU and GPU versions. We also demonstrated that custom CUDA kernels can provide significant speedups, e.g., for the assembly of terms such as right-hand-sides. Data transfer to and from the GPU is encapsulated by developing custom implementations for Gascoigne 3d's data interfaces. Combining these features, we achieved that large parts are consecutively computed on the GPU, so that only infrequent data transfers between CPU and GPU are necessary and these are not significant bottlenecks.

Our approach is conceptually simple since largely only existing linear algebra operations are mapped to the GPU, and requires only limited CUDA expertise. Correspondingly, also the implementation effort for the GPU-parallelization is limited and also substantially simplifies to simultaneously support a CPU and a GPU backend in the code base in the future. The option to directly also integrated custom

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

CUDA kernels provides at the same time great flexibility and has helped us in to substantially reduce the data transfer between CPU and GPU, that otherwise easily become a bottleneck.

We are planning several steps for further optimization: A major limitation of the current GPU parallelization is that the matrices are assembled on the CPU and then transferred statically to the GPU. This severely restricts the applicability to nonlinear problems. The obvious option is to assemble the matrices directly on the GPU. However, this is complicated by a large number of options in the choice and control of the discretization and the use of adaptive grids. Alternatively, we plan to use the GPU and CPU in hybrid mode, so that for nonlinear problems the Jacobi matrix of the Newton solver is always built in the background on the CPU and then transferred to the GPU. This can be seamlessly integrated in the usual inexact Newton algorithm, which only reassembles the Jacobian when the convergence rate deteriorates. A further important step is the GPU parallelization of more powerful multigrid smoothers than the block Jacobi iteration. The realization of a Vanka smoother can be easily implemented with custom kernels and then allows the direct solution of saddle point problems.

An interesting alternative to native CUDA is the use of the high-performance linear algebra libraries that are the backends of machine learning frameworks such as torch and jax, for example torch inductor or XLA. These provide flexible support for a range of accelerators, e.g., also TPUs, and also compilers that optimize the computations for the available compute hardware. Triton is also an interesting intermediate ground between native CUDA and higher level libraries.

Appendix A. Details on the implementation. To describe the necessary changes to the source code for using GPU acceleration and also to explain the role of the custom kernels we give details on the implementation of the interpolated nonlinearity in (5.14). We start with the basic CPU Gascoigne version:

Assembling the nonlinearity (CPU)

```
void nonlinear(Vector& nl, const Vector& u) {
#pragma omp parallel for
    for (size_t i = 0; i < n; ++i) {
      nl(i,0) = u(i,0)*u(i,0); nl(i,1) = u(i,0)*u(i,1);
      nl(i,2) = u(i,0)*u(i,2); nl(i,3) = u(i,1)*u(i,1);
      nl(i,4) = u(i,1)*u(i,2); nl(i,5) = u(i,2)*u(i,2); }
}
void Nonlinear(Vector& f, const Vector& u) {
    nonlin(nl, u); // assembles node-wise product
    GetSolver()->vmulteq(C, f, nl, -1.); // f = -C*nl
}
```

The function `nonlinear(...)` has no native implementation in cuSPARSE. Hence, in the GPU* version, where only sparse matrix vector operations are run on the GPU, the following minimal modification of the function `Nonlinear(...)` is needed:

Assembling the nonlinearity (GPU*)

```
void Nonlinear(Vector& f, const Vector& u) {
    DeactivateCuda(u); // copies vectors to CPU
    nonlin(nl, u); // assembles node-wise product
    ActivateCuda(nl); // copies result to GPU
    GetSolver()->vmulteq(C, f, nl, -1.); // f = -C*nl
}
```

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

The two functions `ActivateCuda(...)` and `DeactivateCuda(...)` set internal flags to activate or deactivate matrix-vector handling on the GPU and they transfer the data of the corresponding vectors such that the function `nonlin(...)` can be processed on the CPU. To avoid this data transfer we must shift this function to a custom kernel.

Assembling the nonlinearity (GPU)

```
__global__ void
nonlin_device(size_t n, double* nl, double* u) {
  if (threadIdx.x + blockIdx.x * blockDim.x >= n)
    return;
  nl[id*6+0]=u[id*3+0]*u[id*3+0];
  nl[id*6+1]=u[id*3+0]*u[id*3+1];
  nl[id*6+2]=u[id*3+0]*u[id*3+2];
  nl[id*6+3]=u[id*3+1]*u[id*3+1];
  nl[id*6+4]=u[id*3+1]*u[id*3+2];
  nl[id*6+5]=u[id*3+2]*u[id*3+2];
}

void nonlin(IndexType n, double* dest, double* src) {
  size_t tpb = 1024;
  size_t blocks = max(1, ceil(n / tpb))
  nonlin_device<<<blocks, tpb>>>(n, dest, src);
}
void Nonlinear(Vector& f, const Vector& u) {
    nonlin(u.n(), nl, u); // node-wise product on GPU
    GetSolver()->vmulteq(C, f, nl, -1.); // f = -C*nl
}
```

Reproducibility of computational results. This paper has been awarded the "SIAM Reproducibility Badge: Code and data available" as a recognition that the authors have followed reproducibility principles valued by SISC and the scientific computing community. Code and data that allow readers to reproduce the results in this paper are available at https://doi.org/10.5281/zenodo.15582537.

Code availability. The source code of the Gascoigne library with CUDA support is found in the Zenodo repository https://zenodo.org/records/13891228 [32]. This repository includes all scripts required to reproduce the examples discussed in this manuscript.

REFERENCES

[1] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, *A survey of numerical linear algebra methods utilizing mixed-precision arithmetic*, Int. J. High Perform. Comput. Appl., 35 (2021), pp. 344–369, https://doi.org/10.1177/10943420211003313.

[2] J. Andrej, N. Atallah, J.-P. Bäcker, J.-S. Camier, D. Copeland, V. Dobrev, Y. Dudouit, T. Duswald, B. Keith, D. Kim, T. Kolev, B. Lazarov, K. Mittal, W. Pazner, S. Petrides, S. Shiraiwa, M. Stowell, and V. Tomov, *High-performance finite elements with mfem*, Int. J. High Perform. Comput. Appl., (2024), https://doi.org/10.1177/10943420241261981.

[3] O. Antepara, S. Williams, H. Johansen, and M. Hall, *High-performance, scalable geometric multigrid via fine-grain data blocking for GPUs*, SC24-W: Workshops of the Interna-

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

tional Conference for High Performance Computing, Networking, Storage and Analysis, 2024, pp. 1177–1191, https://doi.org/10.1109/SCW63240.2024.00159.

[4] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, *Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing*, ACM Trans. Math. Software, 48 (2022), pp. 2:1–2:33, https://doi.org/10.1145/3480935 (accessed 2022-02-17).

[5] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.ii—a general-purpose object-oriented finite element library*, ACM Trans. Math. Softw., 33 (2007), 24–es, https://doi.org/10.1145/1268776.1268779.

[6] J. W. Barrett, J. F. Blowey, and H. Garcke, *On fully practical finite element approximations of degenerate Cahn-Hilliard systems*, ESAIM: M2AN, 35 (2001), pp. 713–748, https://doi.org/10.1051/m2an:2001133.

[7] R. Becker and M. Braack, *Multigrid techniques for finite elements on locally refined meshes*, Numer. Linear Algebr., 7 (2000), pp. 363–379, https://doi.org/10.1002/1099-1506(200009)7:6⟨363::AID-NLA202⟩3.0.CO;2-V, Special Issue.

[8] R. Becker, M. Braack, and T. Richter, *Parallel multigrid on locally refined meshes*, in Reactive Flows, Diffusion and Transport, W. Jäger, R. Rannacher, and J. Warnatz, eds., Springer, Berlin Heidelberg, 2006, pp. 77–92, https://doi.org/10.1007/978-3-540-28396-6_4.

[9] R. Becker and R. Rannacher, *An optimal control approach to a posteriori error estimation in finite element methods*, in Acta Numeritca 2001, A. Iserles ed., 37, Cambridge University Press, 2001, pp. 1–225.

[10] K. Bi, L. Xie, H. Zhang, X. Chen, X. Gu, and Q. Tian, *Accurate medium-range global weather forecasting with 3D neural networks*, Nature, 619 (2023), pp. 533–538, https://doi.org/10.1038/s41586-023-06185-3.

[11] S. Boukhris, A. Napov, and Y. Notay, *Algebraic multigrid using a stencil–CSR hybrid format on GPUs*, SIAM J. Sci. Comput., 45 (2023), pp. C154–C178, https://doi.org/10.1137/22M1480938.

[12] M. Braack, R. Becker, D. Meidner, T. Richter, and B. Vexler, *The finite element toolkit Gascoigne*, 2021, https://doi.org/10.5281/zenodo.5574969.

[13] M. Braack and T. Richter, *Solutions of 3D Navier–Stokes benchmark problems with adaptive finite elements*, Computers and Fluids, 35 (2006), pp. 372–392, https://doi.org/10.1016/j.compfluid.2005.02.001.

[14] M. Braack and T. Richter, *Stabilized finite elements for 3-d reactive flows*, Int. J. Numer. Meth. Fluids, 51 (2006), pp. 981–999, https://doi.org/10.1002/fld.1160.

[15] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: Composable transformations of Python+NumPy programs*, 2018, http://github.com/google/jax.

[16] D. Castillo, U. Kaya, and T. Richter, *A Explicit Time Integration Method for the Boussinesq Approximation*, preprint, 2024.

[17] C. Cui, P. Grosse-Bley, G. Kanschat, and R. Strzodka, *An implementation of tensor product patch smoothers on GPUs*, SIAM J. Sci. Comput., 47 (2025), pp. B280–B307, https://doi.org/10.1137/24M1642706.

[18] C. Cui and G. Kanschat, *Multigrid methods for the stokes problem on GPU systems*, Comput. Fluids, 299 (2025), 106703, https://doi.org/10.1016/j.compfluid.2025.106703.

[19] C. Cui and G. Kanschat, *Multilevel interior penalty methods on GPUs*, ACM Trans. Math. Software, 51 (2025), pp. 1–27, https://doi.org/10.1145/3765616.

[20] L. Failer and T. Richter, *A parallel newton multigrid framework for monolithic fluid-structure interactions*, J. Sci. Comput., 82 (2020), https://doi.org/10.1007/s10915-019-01113-y.

[21] M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek, *Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses*, Comput. Fluids, 80 (2013), pp. 327–332, https://doi.org/10.1016/j.compfluid.2012.01.025.

[22] D. Goddeke, S. H. Buijssen, H. Wobker, and S. Turek, *GPU acceleration of an unmodified parallel finite element Navier–Stokes solver*, in 2009 International Conference on High Performance Computing & Simulation, 2009, pp. 12–21, https://doi.org/10.1109/HPCSIM.2009.5191718.

[23] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, *Using GPUs to improve multigrid solver performance on a cluster*, Int. J. Comput. Sci. Eng., 4 (2008), pp. 36–55, https://doi.org/10.1504/IJCSE.2008.021111.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[24] J.-L. Guermond, *Some implementations of projection methods for Navier–Stokes equations*, ESAIM: Math. Model. Numer. Anal., 30 (1996), pp. 637–667, https://doi.org/10.1051/m2an/1996300506371.

[25] W. Hackbusch, *Multi-Grid Methods and Applications*, Springer, Berlin Heidelberg, 1985, https://doi.org/10.1007/978-3-662-02427-0.

[26] V. Heuveline, D. Lukarski, N. Trost, and J.-P. Weiss, *Parallel Smoothers for Matrix-Based Geometric Multigrid Methods on Locally Refined Meshes Using Multicore CPUs and GPUs*, Springer, Berlin Heidelberg, 2012, pp. 158–171, https://doi.org/10.1007/978-3-642-30397-5_14.

[27] J. G. Heywood and R. Rannacher, *Finite element approximation of the nonstationary Navier–Stokes problem. i. regularity of solutions and second-order error estimates for spatial discretization*, SIAM J. Numer. Anal., 19 (1982), pp. 275–311, https://doi.org/10.1137/0719018.

[28] U. Kaya and T. Richter, *Error analysis of a pressure correction method with explicit time stepping*, Int. J. Numer. Meth. Fluids, 97 (2025), pp. 1363–1378, https://doi.org/10.1002/fld.70000.

[29] M. Kimmritz and T. Richter, *Parallel multigrid method for finite element simulations of complex flow problems on locally refined meshes*, Numer. Linear Algebr., 18 (2011), pp. 615–636, https://doi.org/10.1002/nla.744.

[30] M. Kronbichler and K. Kormann, *Fast matrix-free evaluation of discontinuous Galerkin finite element operators*, ACM Trans. Math. Softw., 45 (2019), pp. 1–40, https://doi.org/10.1145/3325864.

[31] B. N. Lawrence, M. Rezny, R. Budich, P. Bauer, J. Behrens, M. Carter, W. Deconinck, R. Ford, C. Maynard, S. Mullerworth, C. Osuna, A. Porter, K. Serradell, S. Valcke, N. Wedi, and S. Wilson, *Crossing the chasm: How to develop weather and climate models for next generation computers?*, Geosci. Model Dev., 11 (2018), pp. 1799–1821, https://doi.org/10.5194/gmd-11-1799-2018, https://gmd.copernicus.org/articles/11/1799/2018/.

[32] M. Liebchen, R. Jendersie, U. Kaya, C. Lessig, and T. Richter, *GPU parallelization of the finite element toolkit Gascoigne 3d*, 2025, https://doi.org/10.5281/zenodo.13891227.

[33] K. Ljungkvist, *Matrix-free finite-element computations on graphics processors with adaptively refined unstructured meshes*, in Proceedings of the 25th High Performance Computing Symposium, HPC '17, San Diego, CA, Society for Computer Simulation International, 2017.

[34] N. Margenberg, R. Jendersie, C. Lessig, and T. Richter, *DNN-MG: A hybrid neural network/finite element method with applications to 3d simulations of the Navier–Stokes equations*, Comput. Method. Appl. Mech. Engrg.., 420 (2024), 116692, https://doi.org/10.1016/j.cma.2023.116692.

[35] R. T. Mills, M. F. Adams, S. Balay, J. Brown, J. Faibussowitsch, T. Isaac, M. G. Knepley, T. Munson, H. Suh, S. Zampini, H. Zhang, and J. Zhang, *Petsc/tao developments for GPU-based early exascale systems*, Int. J. High Perform. Comput. Appl., 39 (2025), pp. 306–325, https://doi.org/10.1177/10943420241303710.

[36] P. Munch, T. Heister, L. Prieto Saavedra, and M. Kronbichler, *Efficient distributed matrix-free multigrid methods on locally refined meshes for fem computations*, ACM Trans. Parallel Comput., 10 (2023), pp. 1–38, https://doi.org/10.1145/3580314.

[37] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM J. Sci. Comput., 37 (2015), pp. S602–S626, https://doi.org/10.1137/140980260.

[38] NVIDIA, *cublas api reference guide*. Accessed: 2024-04-09. https://docs.nvidia.com/cuda/cublas.

[39] NVIDIA, cuSPARSE library documentation, 2022, https://docs.nvidia.com/cuda/cusparse/index.html.

[40] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux, *Multi-GPU implementation of the lattice Boltzmann method*, Comput. Math. Appl., 65 (2013), pp. 252–261, https://doi.org/10.1016/j.camwa.2011.02.020. Special Issue on Mesoscopic Methods in Engineering and Science (ICMMES-2010, Edmonton, Canada).

[41] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, *Automatic differentiation in pytorch*, in NIPS 2017 Autodiff Workshop, 2017.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.

[42] D. Ribbrock, M. Geveler, D. Göddeke, and S. Turek, *Performance and accuracy of Lattice-Boltzmann kernels on multi- and manycore architectures*, Procedia Comput. Sci., 1 (2010), pp. 239–247, https://doi.org/10.1016/j.procs.2010.04.027.
[43] C. N. Richardson, I. A. Baratta, J. P. Dean, A. Jackson, and G. N. Wells, *An efficient multigrid solver for finite element methods on multi-GPU systems*, Procedia Comput. Sci., 267 (2025), pp. 82–91, https://doi.org/10.1016/j.procs.2025.08.235.
[44] T. Richter, *Fluid-structure Interactions. Models, Analysis and Finite Elements*, Lecture Notes in Computational Science and Engineering 118, Springer, 2017.
[45] D. Ruda, S. Turek, D. Ribbrock, and P. Zajac, *Very fast finite element Poisson solvers on lower precision accelerator hardware: A proof of concept study for Nvidia Tesla V*100, Int. J. High Perform. Comput. Appl., 36 (2022), pp. 459–474, https://doi.org/10.1177/10943420221084657.
[46] D. Rueda, U. Kaya, and T. Richter, *An explicit time integration method for boussinesq approximation*, Proc. Appl. Math. Mech., 24 (2024), e202400050, https://doi.org/10.1002/pamm.202400050.
[47] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996.
[48] A. Sabne, *Xla : Compiling Machine Learning for Peak Performance*, 2020.
[49] N. Sakharnykh, *High-performance geometric multi-grid with GPU acceleration*, NVIDIA Technical Blog, 2016, https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/.
[50] R. Stevenson, *An optimal adaptive finite element method*, SNA, 42 (2005), pp. 2188–2217, https://doi.org/10.1137/S0036142903425082.
[51] I. The Khronos Group, *SYCL: A Cross-Platform Abstraction Layer for Heterogeneous Computing*, 2021. Accessed: 2024-05-06, https://www.khronos.org/sycl.
[52] S. Thomas and A. H. Baker, *Efficient gmres+amg on GPUs: Composite smoothers and mixed, $\boldsymbol{V}$-cycles*, SIAM J. Sci. Comput., 46 (2024), pp. S246–S270, https://doi.org/10.1137/23M1578632.
[53] P. Tillet, H. T. Kung, and D. Cox, *Triton: An intermediate language and compiler for tiled neural network computations*, in Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, New York, NY, Association for Computing Machinery, 2019, pp. 10–19, https://doi.org/10.1145/3315508.3329973.
[54] C. R. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, *Kokkos 3: Programming model extensions for the exascale era*, IEEE Trans. Parall. Distr. Syst., 33 (2022), pp. 805–817, https://doi.org/10.1109/TPDS.2021.3097283.
[55] Y.-H. M. Tsai, N. Beams, and H. Anzt, *Three-precision algebraic multigrid on GPUs*, Future Gener. Comput. Syst., 149 (2023), pp. 280–293, https://doi.org/10.1016/j.future.2023.07.024.
[56] T. Xu, R. P. Li, and D. Osei-Kuffuor, *A two-level GPU-accelerated incomplete lu preconditioner for general sparse linear systems*, Int. J. High Perform. Comput. Appl., 39 (2025), pp. 424–442, https://doi.org/10.1177/10943420251319334.

Downloaded 11/24/26 to 141.44.23.5 . Redistribution subject to SIAM license or copyright; see https://epubs.siam.org/terms-privacy

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited.