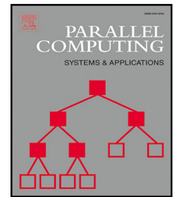
Parallel Computing 118 (2023) 103051

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco



Targeting performance and user-friendliness: GPU-accelerated finite element computation with automated code generation in FEniCS

James D. Trotter [a,*], Johannes Langguth [a,b], Xing Cai [a,c]

[a] Simula Research Laboratory, Kristian Augusts gate 23, Oslo, 0164, Norway
[b] Department of Informatics, University of Bergen, P.O. Box 7803, Bergen, 5020, Norway
[c] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, Oslo, 0316, Norway

ARTICLE INFO

Keywords:
Finite element method
Automated code generation
GPU computing
CUDA
Unstructured mesh

ABSTRACT

This paper studies the use of automated code generation to provide user-friendly GPU acceleration for solving partial differential equations (PDEs) with finite element methods. By extending the FEniCS framework and its automated compiler, we have achieved that a high-level description of finite element computations written in the Unified Form Language is auto-translated to parallelised CUDA C++ code. The auto-generated code provides GPU offloading for the finite element assembly of linear equation systems which are then solved by a GPU-supported linear algebra backend.

Specifically, we explore several auto-generated optimisations of the resulting CUDA C++ code. Numerical experiments show that GPU-based linear system assembly for a typical PDE with first-order elements can benefit from using a lookup table to avoid repeatedly carrying out numerous binary searches, and that further performance gains can be obtained by assembling a sparse matrix row by row. More importantly, the extended FEniCS compiler is able to seamlessly couple the assembly and solution phases for GPU acceleration, so that all unnecessary CPU–GPU data transfers are eliminated. Detailed experiments are used to quantify the negative impact of these data transfers, which can entirely destroy the potential of GPU acceleration if the assembly and solution phases are offloaded to GPU separately. Finally, a complete, auto-generated GPU-based PDE solver for a nonlinear solid mechanics application is used to demonstrate a substantial speedup over running on dual-socket multi-core CPUs, including GPU acceleration of algebraic multigrid as the preconditioner.

1. Introduction

Numerically solving partial differential equations (PDEs) is one of the most important tasks in scientific computing, and the finite element method (FEM) is among the most powerful tools for this task. Finite element codes normally require hand-written computational kernels, particularly for assembling systems of linear equations as the result of finite element discretisation. However, achieving good performance for these kernels requires considerable effort through manual optimisation and careful tuning of the code. Moreover, such efforts may have to be repeated when the PDE or details concerning the discretisation change, or simply if the code is to be run on different hardware. As a result, writing performant FEM codes is already challenging for experts in high-performance computing, and it may be out of reach for domain scientists or others lacking specialist knowledge or time to carry out low-level performance tuning.

A more friendly alternative is to offer a high-level, domain-specific language for describing finite element-based computations, together with a specialised compiler that automatically translates such descriptions to low-level, optimised code for problem-specific kernels. This strategy is used by some open-source frameworks such as FEniCS [1] and Firedrake [2], thus providing high-performance, parallel PDE solvers in a user-friendly manner for real-world applications running on clusters of multi-core CPUs.

While developing finite element codes for efficient use of multi-core CPUs is difficult, porting them to accelerators such as GPUs can be even harder. Nevertheless, the performance and energy efficiency of modern accelerators are very appealing for such codes, which has led to increasing support for GPUs in projects such as libCEED [3], PETSc [4], MFEM [5] and deal.II [6]. Considerable effort has also been devoted to GPU-accelerated linear solvers [7–11]. Moreover, matrix-free and high-order finite element methods, particularly in the case of tensor-product elements, are advocated due to increased arithmetic intensity and reduced memory footprint [3], which enables more efficient use of

* Corresponding author.
E-mail addresses: james@simula.no (J.D. Trotter), langguth@simula.no (J. Langguth), xingca@simula.no (X. Cai).

https://doi.org/10.1016/j.parco.2023.103051
Received 27 December 2022; Received in revised form 13 September 2023; Accepted 21 September 2023
Available online 6 October 2023
0167-8191/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

GPU hardware. For example, MFEM [5] and deal.II [6] offload matrix-free methods (see [5, Section 6.3] and [6, Section 3.7]) to GPUs, where linear systems are not assembled explicitly, but finite element integrals are instead evaluated and used on-the-fly as they are needed.

In this paper, we focus on the frequently encountered case of low-order tetrahedral elements, for which it is still common practice to explicitly assemble a global linear system. Currently, this must be done on the CPU in most finite element libraries. The assembled matrix and right-hand side vector must be subsequently transferred to the GPU for offloading the linear system solver. This paper describes how we have extended FEniCS with automated generation of GPU code for finite element assembly. In addition, coupled with a GPU-supported linear algebra backend, we use FEniCS to deliver finite element solvers with fully GPU-accelerated linear system assembly and solution. Moreover, we present a quantitative study of the impact of various CUDA optimisations applicable to auto-generated finite element assembly code. Our work is fully compatible with all the other features of FEniCS, which means that nonlinearity, vector PDEs, unstructured meshes, complicated boundary conditions, and so on, pose no hindrance to GPU acceleration. The scope of the current paper is offloading to a single GPU, which is an important step towards the multi-GPU acceleration of finite element methods with automated code generation.

This paper presents a *seamless* GPU offloading of both the assembly and solution phases in a high-level, user-friendly finite element framework supported by automated code generation. This approach avoids the unnecessary CPU–GPU data transfers, which can severely harm the overall performance. Our main contributions are as follows:

- automatic generation of CUDA kernels for offloading per-element computations to NVIDIA GPUs, together with a practical implementation in FEniCS;
- a comparison of different offloading strategies for the entire finite element assembly procedure, along with additional performance optimisations;
- detailed microbenchmarks and profiling, which provide insight into application- and kernel-level performance bottlenecks, including a careful review of data transfers between CPU and GPU;
- a rigorous performance comparison with state-of-the-art, automated finite element solvers on multicore CPUs;
- demonstration of a complete, nonlinear PDE solver using automated code generation and GPU-offloading on NVIDIA A100.

The rest of the paper is organised as follows. First, we provide some background on finite element assembly, automated code generation and the FEniCS PDE solver framework in Section 2. In Section 3, we describe our GPU-offloaded finite element assembly algorithm, including how we extended the built-in automated code generation features of the FEniCS form compiler. We also discuss some optimisations that can be applied to improve the performance of the fully offloaded assembly. Then, in Section 4, we present numerical experiments and results to illustrate key parts of the development and optimisation process, including the effectiveness of various optimisations and the performance of the final GPU-based assembly algorithm. This is followed by a discussion of related work in Section 5 and concluding remarks in Section 6.

2. Finite element assembly and automated code generation

Roughly speaking, finite element solvers may be broken down into two major computational phases, where the first phase consists of *assembling systems of linear equations* and the second phase concerns *solving them*. Since solving a linear system is a generic procedure used in many types of numerical computations, its GPU acceleration is a well studied subject [9]. On the other hand, the phase of linear system assembly, which discretises a PDE following the finite element principle, is a less studied area with respect to automated code generation and, particularly, GPU acceleration. In this section, we give a brief description of the assembly phase. The purpose is to show the main computational steps involved and explain how automated code generation can be applied.

2.1. Finite element assembly

The finite element assembly procedure begins with a decomposition of the problem domain into a mesh of non-overlapping cells, called *elements*. There are many choices of the element type, such as triangles in 2D and tetrahedra in 3D. In any case, the numerical solution of the PDE under consideration is sought in each element as a combination of prescribed *shape functions*, which are typically piecewise polynomials. The goal of the assembly procedure is to create a global linear system, $Ax = b$, where the solution vector x will contain the weights, often referred to as *degrees of freedom*, for combining the shape functions into the overall numerical solution.

The global matrix A and right-hand side vector b are obtained by adding together contributions from individual elements. Since the shape functions can span across neighbouring elements, each non-zero value in A or b is normally a sum of values from several elements. Furthermore, the matrix A is always sparse, because each shape function is only non-zero in a small number of elements. In addition, the non-zero values will be irregularly placed in A when using an unstructured computational mesh.

Algorithm 1 shows a high-level pseudocode for the case of assembling a global matrix. The overall procedure loops over each element, T, to compute a small, dense element matrix A_T. As a rule, some form of numerical quadrature is usually involved, but the precise details surrounding how to form element matrices and vectors depend on the choice of shape functions as well as the so-called *variational form* of the PDE in question (an example will be given for Poisson's equation in Section 2.2).

For each element, step 1 is to obtain the vertex coordinates of the mesh cell, which must be read from memory in an irregular fashion when an unstructured mesh is used. Depending on the PDE being solved, it may also be necessary to similarly fetch some problem-dependent coefficient values from memory. Then, an element matrix (or vector) is computed in step 2, making use of the vertex coordinates and coefficients, as well as details specific to the target PDE and shape functions being used. At this point, some additional work (step 3) may be needed to correctly enforce boundary conditions, e.g., Dirichlet boundary conditions. We omit the details, but note that in the case of inhomogeneous Dirichlet boundary conditions, certain element matrix values also contribute to the assembled right-hand side vector.

Finally, once an element vector or matrix has been computed, its values are added to the global vector or matrix. To this end, step 4 requires a map of the global degrees of freedom that each element contributes to. For matrix assembly, the map designates rows and columns in the global matrix where element matrix values must be added. If the global matrix is stored in a sparse format, such as compressed sparse row (CSR), an additional step is needed to locate the correct position in the 1D array of non-zero global matrix values where an element matrix contribution should be added. A common strategy is to perform a binary search in the list of column indices associated with the non-zero values of the appropriate row in the sparse matrix.

2.2. Element matrices and vectors

To provide a simple example of computing the element matrix and vector, we consider Poisson's equation on a polygonal domain $\Omega \subset \mathbf{R}^d$ with a boundary $\partial\Omega$ and a very simple boundary condition,

$$-\kappa\nabla^2 u = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial\Omega, \tag{1}$$

Algorithm 1 Pseudocode for elementwise assembly of a global matrix A.

Data: elements $T_1, T_2, \dots, T_N$, vertices $x_1, x_2, \dots, x_M$, and degrees of freedom for each element: $\mu^{T_1}, \mu^{T_2}, \dots, \mu^{T_N}$
for $i = 1, 2, \dots, N$ **do**
  Let $T_i = [x_{i_1}\ x_{i_2}\ \dots\ x_{i_m}]$ and $\mu^{T_i} = \{k_1, k_2, \dots, k_n\}$
  step 1: gather vertex coordinates $x_{i_1}, \dots, x_{i_m}$
  step 2: compute element matrix A_{T_i}
  step 3: modify rows and columns of A_{T_i} in case of Dirichlet boundary conditions
  step 4: add A_{T_i} to global matrix A (i.e., $A_{k_p,k_q} \leftarrow (A_{T_i})_{p,q}$ for $0 \le p, q < n$)
end for

where u is the solution to be found, $\kappa > 0$ is a constant and f is a given source term. The corresponding variational form has two parts (a bilinear form and a linear form) as follows:

$$a(v, u) = \int_\Omega \kappa \nabla v \cdot \nabla u \, \mathrm{d}x, \quad L(v) = \int_\Omega f v \, \mathrm{d}x. \tag{2}$$

Here v denotes a test function belonging to an appropriate function space depending on the element type and shape functions. See, for example, [12], for further details.

On a mesh cell T, the element matrix A_T is computed by evaluating integrals over T (for each pair of p, q),

$$\left(A_T\right)_{p,q} = \int_T \kappa \nabla \psi_p^T \cdot \nabla \phi_q^T \, \mathrm{d}x, \tag{3}$$

where ψ_p^T and ϕ_q^T are non-zero local shape functions on T (also called local test and trial functions). Each value in the element vector b_T is calculated by $\int_T f \psi_p^T \, \mathrm{d}x$.

To implement a complete finite element assembly procedure, one must program the computational kernels for computing element matrices and vectors for the particular problem at hand. Such kernels can either be hand-written for each variational form, or be automatically generated from a high-level description.

2.3. Unified Form Language

The *Unified Form Language* (UFL) [13] is a high-level language for specifying variational forms in a manner close to their mathematical description. It is used by multiple projects, including FEniCS, to enable user-defined variational forms expressed as integrals over cells, interior faces, and boundary faces of a mesh. The user indicates whether an integral is to be taken over the whole mesh or some user-defined regions. Furthermore, the user can freely choose the desired element type and shape function [14] for the test and trial functions, as well as prescribing any coefficients that may be involved.

UFL is a domain-specific language embedded in Python, so users can take advantage of Python's features when writing variational forms. This becomes particularly useful in more advanced PDE solver applications. The code in Algorithm 2 shows UFL descriptions of the standard variational form for Poisson's equation (1). Here, linear Lagrange elements on tetrahedral mesh cells are prescribed for the trial and test functions `u` and `v` and for the coefficient (source term) `f`. The bilinear and linear forms from Eq. (2) are spelt out using `grad` and `inner` to indicate the gradient operator and dot product, whereas `dx` signifies integration over the domain.

2.4. Compiling variational forms

The *FEniCS form compiler* (FFC) is responsible for translating variational forms written in UFL into kernels needed for the finite element assembly. As usual for a compiler, FFC internally constructs an abstract syntax tree, which provides the possibility of program transformations and optimisations (if needed) before FFC finally translates the result to plain C code. This approach allows for runtime code generation and just-in-time (JIT) compilation by later invoking a C compiler at runtime.

Algorithm 2 Variational form for Poisson's equation in UFL.

```
cell = tetrahedron
element = FiniteElement("Lagrange", cell, 1)
coords = VectorElement("Lagrange", cell, 1)
mesh = Mesh(coords)
V = FunctionSpace(mesh, element)
u = TrialFunction(V)
v = TestFunction(V)
f = Coefficient(V)
kappa = Constant(mesh)
a = kappa * inner(grad(u),grad(v))*dx
L = inner(f,v)*dx
```

In addition to its form language and compiler, FEniCS also internally uses a high-level C++ library, called DOLFIN. The objective is to tie the automated code generation together with all the other components needed in finite element solvers, such as handling unstructured, computational meshes, carrying out finite element assembly, and solving the resulting linear equation systems.

The C code generated by FFC follows a predefined template and includes functions for mapping between arbitrary mesh cells and reference cells, evaluating shape functions and their derivatives, as well as performing numerical quadrature needed to compute element matrices and vectors. In FEniCS's terminology, the generated functions for computing element vectors and matrices are referred to as *tabulate tensor* kernels. Although such kernels usually involve numerical integration over a reference element, FEniCS frequently employs an optimisation technique based on tensor contractions to factor out terms that are independent of element geometry [15]. In particular, for affine meshes and variational forms with constant coefficients, integrals are precomputed exactly during code generation and there is no need to perform numerical integration at runtime. In cases where quadrature is still needed (e.g., variable coefficients), FEniCS will generate quadrature loops after automatically selecting a suitable quadrature scheme and degree. For example, the quadrature schemes described in [16] or the collapsed Gauss schemes are used for triangles and tetrahedra, whereas tensor product quadrature rules are used for tensor-product elements.

Various other code generation and optimisation techniques have been proposed for FFC, including simplifying expressions through constant folding and common subexpression elimination [17,18], as well as various loop optimisations like loop hoisting, fusion or splitting [19–21]. Further details are found in the FEniCS book [1] or in references given in Section 5. Nevertheless, the main optimisation currently employed by FFC involves precomputing certain integrals, if possible. Otherwise, FFC performs a fairly straightforward translation to C code, with the assumption that the C compiler is capable of further optimising the generated code to a sufficient degree.

3. GPU implementation of finite element assembly

In this section, we explain how we have enabled FEniCS to offload finite element assembly to GPUs, including different offloading strategies and optimisations that have been incorporated into our extended versions of FFC and the DOLFIN library used inside FEniCS.

Our approach is based on using CUDA [22], which is the most popular method of GPU programming as of now. CUDA extends C++ to allow offloading subprograms, called CUDA kernels, from a host CPU by launching them on a device (i.e., GPU). Each CUDA kernel is executed on the device by a large number of concurrent threads, often requiring fine-grain data parallelism to make use of all the available hardware resources. This fine-grain parallelism is a good match for the finite element assembly phase, and also the solution phase to a large extent. Moreover, like many sparse computations, the performance of finite element solvers is mostly limited by memory bandwidth (see, e.g., [23,24]). As a result, the high-bandwidth memory commonly employed in GPU accelerators (e.g., [25]) should give an advantage compared to executing the same computations on the host CPU.

3.1. Auto-generating CUDA kernels

Recall that the form compiler, FFC, is responsible for translating variational forms written in UFL into tabulate tensor kernels needed for computing element vectors and matrices, i.e., step 2 in Algorithm 1. Our first step towards offloading to a CUDA device is to extend FFC to emit tabulate tensor kernels compatible with CUDA C++. This is accomplished by first automatically annotating generated functions with a `__global__` execution space specifier, thus instructing the CUDA C++ compiler that the function must be callable from the host in the form of a CUDA kernel launch [26]. Second, we address various minor incompatibilities, such as letting the extended FFC replace the C99 keyword `restrict` with the `__restrict__` keyword, since the latter is supported in CUDA C++. These automatically inserted keywords in the generated code mitigate pointer aliasing issues that otherwise prevent critical compiler optimisations, such as reordering and common subexpression elimination. Finally, the extended FEniCS framework has taken care to avoid including external header files, which would otherwise complicate runtime code compilation that takes place later (see, e.g., [27]). We apply these changes to FFC by modifying the predefined template that is used to emit the final, auto-generated code.

The generated CUDA kernels are written such that a single CUDA thread computes an entire element vector or matrix. More fine-grained data parallelism is possible, especially in the case of higher-order elements (see, e.g., [28–30]). Furthermore, there may be room for other improvements by tuning the generated CUDA kernels. However, we find that the CUDA code produced by the extended FFC, combined with optimisations of the CUDA C++ compiler, are already sufficient to provide good performance for offloading assembly to GPUs. We therefore defer additional performance tuning of the generated CUDA kernels to a future work and instead focus on strategies and performance considerations with respect to offloading the overall finite element assembly algorithm.

3.2. Runtime compilation of CUDA C++

Since the PDE-specific variational forms are usually provided at runtime, we use NVIDIA's runtime compilation API (NVRTC) [31] to process the auto-generated CUDA code and thereby obtain CUDA kernels that can be launched on a GPU. Given a string of CUDA C++ source code, NVRTC produces compiled and optimised assembly language code for the PTX instruction set architecture. The PTX assembly is then loaded using the CUDA driver API [22] by calling the function `cuModuleLoadDataEx`, which further compiles PTX assembly to CUDA device code for a specific GPU model. Once a module is loaded, the host may offload computations to a CUDA device by launching CUDA kernels for any functions that are annotated with the `__global__` keyword.

3.3. Transferring data to GPU memory

Prior to offloading, the required input data must be copied from host memory to device memory, including degrees of freedom for each element, mapping info, vertex coordinates, coefficients, as well as boundary markers for Dirichlet boundary conditions (if any). One option is to make use of CUDA's unified memory model [26], which provides a common address space and automatically manages data transfers between host and device memory. However, transferring data between host and device is prone to become a performance bottleneck, so we choose to manage data transfers within the extended FEniCS framework explicitly using `cudaMemcpy`. Consequently, we augment various classes and data structures in DOLFIN (the accompanying C++ library of FEniCS) to mirror and synchronise data that must be present in both host- and device-side memory.

Besides the data needed for assembly, the resulting linear system may also need to reside in GPU memory if a subsequent linear solver runs on the same device. At least in the case of offloading the entire assembly procedure, as discussed in Section 3.5, there is no need to copy the matrix or vector back to the host. For FEniCS and other codes that use PETSc [32] to perform sparse linear algebra, special care must be taken to use `MatSeqAIJCUSPARSEGetArray` and `MatSeqAIJCUSPARSERestoreArray`,[1] which provide direct access to sparse matrix data on the device. Thus, the extended FEniCS framework internally creates a device-side pointer to the non-zero matrix values, performs assembly on the GPU, and passes the assembled matrix directly to a GPU-enabled linear solver without transferring any data to the host. Numerical experiments in Section 4 are provided to demonstrate the importance of this issue.

3.4. Partial offloading

Section 3.1 has described how FFC is extended to generate CUDA compatible code for step 2 in Algorithm 1, also called tabulate tensor kernels. GPU offloading of the other steps of Algorithm 1 requires further extensions of FEniCS. First, we note that naively launching a CUDA kernel for each element incurs too much overhead and would prevent any acceleration. The solution is instead to devise higher-level CUDA kernels that loop over every element in parallel, calling the auto-generated tabulate tensor kernels to compute element matrices or vectors. Assigning one thread per element ensures an ample amount of parallelism. For example, tens of thousands of elements or more can be processed in parallel on an NVIDIA V100 GPU, which has 80 streaming multiprocessors (SMs) and typically employs up to 2048 threads per SM.

Returning to Algorithm 1, offloading steps 1–3 is straightforward, whereas the final step of adding element matrices to the global matrix requires some subtlety, as will be discussed in Section 3.5. Algorithm 3 shows an example of an auto-generated CUDA kernel for a "partial offloading" of the assembly, i.e., only steps 1–3 of Algorithm 1. This is for the purpose of comparing with a "full offloading" approach to be described in the following subsections. The "partial offloading" approach is easier to code, but suffers from lower performance.

For the purpose of illustration, the example in Algorithm 3 (and subsequent examples in Algorithms 4 and 5) was generated from a bilinear form based on first-order elements on a tetrahedral mesh, and it therefore involves 4-by-4 element matrices. This can be seen in the various constants appearing in array sizes and loop ranges throughout the generated code. For other variational forms, element types and polynomial degrees, the size of the element matrices may be different and the form compiler will generate code accordingly.

The main CUDA kernel of Algorithm 3 is a *grid-stride loop*, where thread i computes element vectors or matrices for cells i, $i+N$,

[1] These functions were added in PETSc 3.16.4.

$i + 2N$, and so on, with N being the total number of threads. The cell vertex coordinates are stored in per-thread arrays with automatic storage duration, which means that the CUDA compiler will try to place the data in registers. If that is not possible, local memory is used, potentially transferring to and from GPU device memory. Each call to `tabulate_tensor` computes an element matrix or vector (see Section 2.4), placing the computed values in device memory. Moreover, element matrix values are adjusted whenever Dirichlet boundary conditions apply. Afterwards, all the element matrices are transferred from device memory to host memory, where each element matrix is added to the global matrix by calling the PETSc function `MatSetValues`.

Algorithm 3 Auto-generated CUDA C++ code for the "partial offloading" approach. This example uses 4×4 element matrices, corresponding to first-order elements on a tetrahedral mesh.

```
1  void __global__ cuda_local_assembly(
2    int num_active_cells, const int * active_cells,
3    const int * vertices_per_cell,
4    const double * vertex_coords,
5    int num_coeffs_per_cell, const double * coeffs,
6    const double * constants,
7    const int * dofmap0, const int * dofmap1,
8    const char * bc0, const char * bc1,
9    double * values)
10 {
11   for (int i=blockIdx.x*blockDim.x+threadIdx.x;
12        i < num_active_cells;
13        i += blockDim.x * gridDim.x) {
14     // Set element matrix values to zero
15     double* Ae = &values[i*4*4];
16     for (int j = 0; j < 4*4; j++) Ae[j] = 0.0;
17
18     // Gather cell vertex coords/coefficients
19     int c = active_cells[i];
20     double cell_vertex_coords[4*3];
21     for (int j = 0; j < 4; j++) {
22       int vertex = vertices_per_cell[c*4+j];
23       for (int k = 0; k < 3; k++)
24         cell_vertex_coords[j*3+k] =
25           vertex_coords[vertex*3+k];
26     }
27     const double * cell_coeffs = &coeffs[
28       c*num_coeffs_per_cell];
29
30     // Compute element matrix
31     tabulate_tensor(Ae, cell_coeffs, constants,
32         cell_vertex_coords);
33
34     // Handle Dirichlet boundary conditions (...)
35   }
36 }
```

3.5. Full offloading

To offload the entire assembly, we next consider how step 4 in Algorithm 1 can also be moved from the host to a GPU. First, it is necessary to maintain a copy of the sparse matrix data structure on the CUDA device. If we assume that the global matrix is stored in the CSR format, then the row pointers and column indices of non-zero matrix entries are copied to device memory prior to assembly. This is done only once, since the matrix structure usually does not change. After the assembly is finished, the newly computed global matrix values can always be transferred from device memory to host memory, if needed. On the other hand, transferring the data is costly and it can be a great advantage to avoid doing so unnecessarily, for example, in cases where the solver is also offloaded to the GPU. Note that the amount of data transferred depends on the connectivity of the mesh and the particular discretisation used, but it is always much less than the partial offloading approach considered in the previous section.

Algorithm 4 Auto-generated CUDA C++ code for global matrix assembly in the "full offloading" approach. Some parts of this CUDA kernel are identical to Algorithm 3 and have therefore been abbreviated.

```
1  void __global__ cuda_global_assembly(
2    (...) // Input arguments (see Algorithm 3)
3    const int * rowptr,
4    const int * colidx,
5    double * values)
6  {
7    for (int i=blockIdx.x*blockDim.x+threadIdx.x;
8      i < num_active_cells;
9      i += blockDim.x * gridDim.x)
10   {
11     (...) // Set element matrix to zero
12     (...) // Gather vertex coords/coefficients
13     (...) // Compute element matrix
14
15     // Add values to global matrix, skipping
16     // degrees of freedom subject to
17     // Dirichlet boundary conditions
18     for (int j = 0; j < 4; j++) {
19       int row = dofmap0[c*4+j];
20       if (bc0 && bc0[row]) continue;
21       for (int k = 0; k < 4; k++) {
22         int column = dofmap1[c*4+k];
23         if (bc1 && bc1[column]) continue;
24         int r = binary_search(
25           rowptr[row+1] - rowptr[row],
26           &colidx[rowptr[row]], column);
27         r += rowptr[row];
28         atomicAdd(&values[r], Ae[j*4+k]);
29       }
30     }
31   }
32 }
```

The CUDA kernel for the "full offloading" approach is shown in Algorithm 4, which is added to DOLFIN in the extended FEniCS framework. There are two main differences compared with Algorithm 3. First, element matrices are now stored in per-thread arrays with automatic storage duration, meaning that the compiler will use registers or local memory. Second, for each cell, a final step is carried out, where a binary search is performed for each entry of the element matrix to find the corresponding position in the array of global matrix values. Once the correct location is found, the element matrix value is added to the global matrix using an atomic operation, which is necessary to avoid race conditions between threads that may attempt to update the same value simultaneously.

3.6. Lookup table for global matrix values

Because sparse matrix storage formats (e.g., CSR) usually require a search when adding element matrix values to a global matrix, our fully offloaded assembly algorithm becomes prone to branch divergence. The result is a severe slowdown whenever threads in a warp are led to execute different code paths during searches. Closer examination using the NVIDIA Visual Profiler [33] confirms that certain source code lines of the binary search procedure in Algorithm 4 are associated with as much as 90% branch divergence. Moreover, the profiler reveals that for 95% of program counter samples retrieved during execution, threads are stalled waiting on memory dependencies, meaning that performance is limited by memory latency rather than compute capacity or memory bandwidth.

To alleviate this performance bottleneck, we investigate the effect of using a lookup table to replace the costly binary searches needed in Algorithm 4. The same concept proved highly beneficial also for CPU-based assembly in previous work [24]. The idea is to perform binary searches for element matrix entries only once as a pre-computation,

and the results are stored in a lookup table. The CUDA kernel in Algorithm 5—which is also discussed further in the next section—illustrates how the lookup table (`nonzero_locations`) is consulted during assembly to directly obtain the locations of global matrix non-zeros to which an element contributes.

Generally, the lookup table can anyway be obtained at little or no extra cost while setting up the matrix sparsity pattern prior to assembly. The savings can therefore be substantial, especially when repeatedly assembling a global matrix with the same sparsity pattern, for example, during every time step in a time-dependent problem or every iteration in a nonlinear solver.

The lookup table itself requires an additional $4Mn^2$ bytes to be read from global memory, where M is the number of mesh cells and n is the number of rows and columns of an element matrix. (Each entry in the lookup table is an integer of 4 bytes.) On the other hand, it is no longer necessary to read degree-of-freedom maps (`dofmap0` and `dofmap1`, markers for Dirichlet boundary conditions (`bc0` and `bc1`), row pointers (`rowptr`) or column indices (`colidx`) of the global matrix, which amount to $8Mn$, $2N$, $4(N + 1)$, and $4K$ bytes, respectively, where N is the number of rows and columns and K is the number of non-zeros of the global matrix. Moreover, accesses to the lookup table are easily coalesced and therefore enjoy the high bandwidth of the device memory.

3.7. Rowwise assembly

Even after eliminating branch divergence, there are some challenges remaining with respect to the fully offloaded assembly procedure. First, there is some contention caused by separate threads writing to the same cache line in the atomic add operation (i.e., line 28 in Algorithm 4). Second, large parts of the data, including vertex coordinates, coefficients, and especially the matrix non-zeros, are accessed in an irregular fashion. Both of these problems may lead to poor use of the memory subsystem.

We therefore include an alternative algorithm, previously proposed by Cecka, Lew and Darve [34], and also considered in our previous work on CPU-based assembly [24], that involves assembling the global matrix row by row. The main advantage of this method is that accesses to the global matrix become more regular compared to elementwise assembly, reducing the overall memory traffic as well as cache line contention associated with the atomic add. On the other hand, some redundant work is performed if a thread uses the auto-generated `tabulate_tensor` kernel to compute an entire element matrix, but only the values of a single row are needed. In the context of our automated CUDA code generation, the benefit of rowwise assembly is thus unclear, but the numerical experiments in Section 4.2 show that it may indeed be worthwhile.

The resulting algorithm, shown in Algorithm 5, requires a mapping from the global degrees of freedom of the test space to the mesh cells that contain them (`cells_per_dof`). Since the number of cells per degree of freedom varies, a compressed row format is used to indicate the first and last cell for each degree of freedom (`cells_per_dof_ptr`). In addition, there is an auxiliary array (`element_matrix_rows`) that is used to look up which row in the element matrix is needed for a given global degree of freedom and mesh cell. More specifically, after computing the element matrix, we loop over the rows of the element matrix, and the conditional statement on line 22 in Algorithm 5 uses `element_matrix_rows` to select a single row of the element matrix that is added to the global matrix.

Recall that the outer loop in Algorithms 3 and 4 ranges over the cells of the mesh. In contrast, the outer loop in Algorithm 5 ranges simultaneously over every matrix row of the global matrix and every mesh cell that contains the degree of freedom corresponding to that row. As a result, every degree of freedom in every mesh cell is assigned to a thread. This particular work division requires us to perform atomic updates of the global matrix to avoid race conditions, just like in

Algorithm 5 Auto-generated CUDA C++ code for rowwise assembly of a global matrix. Some parts of this CUDA kernel are identical to Algorithm 3 and have therefore been abbreviated.

```
1  void __global__ cuda_rowwise_assembly(
2    (...) // Input arguments (see Algorithm 3)
3    const int * cells_per_dof_ptr,
4    const int * cells_per_dof,
5    const int * nonzero_locations, // Lookup table
6    const int * element_matrix_rows,
7    int num_rows,
8    double * values)
9  {
10   for (int p=blockIdx.x*blockDim.x+threadIdx.x;
11        p < cells_per_dof_ptr[num_rows];
12        p += blockDim.x * gridDim.x)
13   {
14     (...) // Set element matrix to zero
15     (...) // Gather vertex coords/coefficients
16     (...) // Compute element matrix
17
18     // Add values to global matrix, skipping
19     // degrees of freedom subject to
20     // Dirichlet boundary conditions
21     for (int j = 0; j < 4; j++) {
22       if (j != element_matrix_rows[p]) continue;
23       for (int k = 0; k < 4; k++) {
24         int l = ((p/warpSize)*4+k)*warpSize +
25           p % warpSize;
26         int r = nonzero_locations[l];
27         if (r < 0) continue;
28         atomicAdd(&values[r], Ae[j*4+k]);
29       }
30     }
31   }
32 }
```

Algorithm 4, because global degrees of freedom that belong to more than one mesh cell may be shared by different threads. It is possible to instead assign an entire row of the global matrix to a single thread and thus avoid the need for atomic operations. But the cost of these atomic operations on the GPU appears to be small, and the scheme we have chosen instead leads to good load balancing even if there is a large variation in the number of cells from one degree of freedom to another.

While the rowwise assembly shown in Algorithm 5 invokes the `tabulate_tensor` kernel to compute the entire element matrix every time, we also explored an alternative version with a modified `tabulate_tensor` kernel that computes only the single element matrix row that is needed. As a result, fewer redundant computations are needed overall. These two rowwise methods are compared in Section 4.2.

4. Numerical experiments

In this section, we measure our GPU-based finite element assembly implementations to highlight strengths and weaknesses of various approaches and the effectiveness of the proposed optimisations. We also validate our GPU-enabled form compiler using a nonlinear solid mechanics problem and a detailed comparison of CPU and GPU performance.

4.1. Experimental setup

Our GPU experiments were carried out on NVIDIA V100 and NVIDIA A100 GPUs. In addition, some experiments were carried out on three different dual-socket, multi-core CPU systems featuring Intel Xeon Gold 6130, AMD Epyc 7601 (“Naples”) and AMD Epyc 7763 (“Milan”) CPUs. An overview is found in Table 1, including measurements of achievable bandwidth based on BabelStream [35] for the NVIDIA GPUs, and the STREAM benchmark [36] for the CPUs. Throughout the benchmarks presented here, we use CUDA 10.1 and 11.7 on NVIDIA

Table 1
Hardware used in our experiments.

	NVIDIA V100	NVIDIA A100	Intel Xeon Gold 6130	AMD Epyc 7601	AMD Epyc 7763
Microarchitecture	Volta	Ampere	Skylake	Zen	Zen 3
SMs/cores	80	108	32	64	128
Frequency	1.46 GHz	1.41 GHz	1.9–3.6 GHz	2.7–3.2 GHz	2.5–3.5 GHz
FP64 performance	7450 Gflop/s	9750 Gflop/s	1946 Gflop/s	1382 Gflop/s	5120 Gflop/s
Memory bandwidth	898 GB/s	2039 GB/s	256 GB/s	342 GB/s	410 GB/s
STREAM triad	887 GB/s	1771 GB/s	147.1 GB/s	161.4 GB/s	256.5 GB/s

Table 2
Computational meshes used in numerical experiments.

Mesh	Vertices	Cells
Uniform mesh 1	226 981	1 296 000
Uniform mesh 2	531 441	3 072 000
Uniform mesh 3	1 030 301	6 000 000
Uniform mesh 4	1 771 561	10 368 000
Uniform mesh 5	2 803 221	16 464 000
Uniform mesh 6	4 173 281	24 576 000
Uniform mesh 7	5 929 741	34 992 000
Uniform mesh 8	8 120 601	48 000 000
Cardiac mesh 1	1 255 775	6 735 654
Cardiac mesh 2	1 958 816	10 697 116
Cardiac mesh 3	2 226 802	12 255 517
Cardiac mesh 4	3 019 809	16 907 270

V100 and A100, respectively. Furthermore, GCC 11.2.0 was used with the compiler flags `-O3 -march=native`. Finally, our implementation that supports GPU offloading was carried out based on a development version of FEniCSx, an upcoming, redesign of the FEniCS framework. All the codes are archived and made available in Zenodo [37].

The following experiments use two sets of unstructured, tetrahedral meshes with up to 48 million tetrahedral cells, as shown in Table 2. The first set consists of standard, uniform discretisations of the unit cube subdivided into 60^3 up to 200^3 equal-sized, smaller cubes, each further subdivided into six tetrahedra. The remaining meshes are from a cardiac modelling application [38], based on patient data from a Danish study on cardiac disease [39]. These meshes are more representative of real-world applications involving unstructured meshes.

4.2. Offloading finite element assembly to a GPU

In this section, we consider the performance of different GPU-offloaded assembly algorithms for Poisson's equation (see Section 2). For this, we use the cardiac meshes and one of the mid-sized uniform meshes. The main results are shown in Table 3, which compares the performance of assembling a matrix for Poisson's equation using different CUDA-based assembly algorithms on NVIDIA V100. Performance is reported as the number of degrees-of-freedom per second, in millions, or Mdof/s. In the following, we have also used the NVIDIA Visual Profiler to measure register usage, occupancy, achieved memory bandwidth, thread divergence, and so on.

For comparison, the performance of FEniCS's MPI-parallel, CPU-based assembly on Xeon and Epyc (Naples) is also shown. Note, however, that the current CPU version of FEniCS has not been fully optimised. Therefore, we also show the performance of a standalone, optimised benchmark that implements an OpenMP-parallel, CPU-based assembler for the Poisson problem with first-order elements. This CPU benchmark uses both the lookup table and rowwise assembly optimisations, which yield significantly improved performance. Further details can be found in [24]. Using 64 cores on Naples, the best performance for assembling a matrix for *Cardiac mesh 4* on the CPU is about 70 Mdof/s. Nonetheless, the best version of GPU-based assembly on NVIDIA V100 is about four times faster at 286 Mdof/s.

The ineffectiveness of the partial offloading approach is very clear. Its poor performance is a result of expensive data transfers between CPU and GPU and the fact that only a single core of the host CPU is used when adding element matrices to the global matrix. We are limited to a single core in this case because there is currently no easy way of extending the CPU-based global assembly code in FEniCS to use more cores with shared memory parallelism. The alternative is to use one MPI process per core, possibly allowing processes to share the same GPU. We plan to extend our GPU-accelerated approach to support distributed-memory parallelism through MPI in the future, but the partial offloading approach would still suffer from CPU-GPU transfers.

In contrast, a tremendous acceleration is achieved by fully offloading the assembly. The achieved performance varies from about 85 to 230 Mdof/s, which shows that the size and structure of the computational mesh has a significant performance impact. Detailed profiling confirms that larger, unstructured meshes, such as *Cardiac mesh 2*, *3* and *4*, benefit less from caching on NVIDIA V100 when accessing vertex coordinates and global matrix data. Profiling also reveals that performance is mostly limited by memory latency, since the achieved throughput is nowhere near the available bandwidth and the floating-point and other functional units are far from saturated.

As discussed in Section 3.6, the latency observed for the CUDA kernel in the "full offloading" approach is closely related to the binary search procedure that is employed, which leads to a considerable amount of divergence among threads in a warp. Table 3 shows that replacing binary searches with a lookup table leads to a notable speedup of about 1.7 to 1.9× for the larger meshes. In addition to almost entirely eliminating thread divergence, the number of registers per thread is reduced from 82 to 64, and the occupancy (i.e., the number of active threads) is thus increased from 31% (640 threads) to 50% (1024 threads). In other words, there are more threads available to hide potential memory latencies. At this point, the achieved memory bandwidth is about 350 GB/s, which is nearly 40% of the 900 GB/s theoretical maximum. Improved coalescing of global memory accesses could possibly improve performance further, but it is not clear how this can be achieved for the irregular memory accesses involved in reading vertex coordinates.

Finally, a further speedup of up to 70% is gained by employing the rowwise assembly algorithm for our benchmark problem. The motivation for employing the rowwise algorithm was primarily its improved data locality for writing to the global matrix. This behaviour is indeed confirmed by profiling, which shows, for example, that cellwise assembly for *Cardiac mesh 4* results in more than 2 GiB of data written from the NVIDIA V100's L2 cache to device memory, whereas the rowwise method results in only a quarter of that at 500 MiB. Furthermore, the rowwise approach appears to use even fewer registers, only 55 per thread, and thus reaches a slightly higher occupancy of 56% (1152 threads). The achieved memory bandwidth for this CUDA kernel is about 510 GB/s, or 57% of the theoretical maximum, which should be considered a high degree of bandwidth usage bearing in mind the significant amount of irregular memory accesses involved.

One of the remaining challenges in the rowwise CUDA kernel in Algorithm 5 is the conditional statement on line 22, which is used to select a single row of the element matrix that should be added to the thread's current row of the global matrix. As one might expect, this leads to some thread divergence and is therefore a potential performance issue. We have also briefly experimented with an alternative version of the rowwise method by modifying the `tabulate_tensor` kernel to compute only the single element matrix row

Table 3
Performance (in Mdof/s) of matrix assembly for Poisson's equation with linear (P1) elements on dual-socket Intel Xeon Gold 6130 and AMD Epyc "Naples" 7601 CPUs, and an NVIDIA V100 GPU.

P1 elements	Xeon Gold 6130 (CPU)		Epyc "Naples" 7601 (CPU)		NVIDIA V100 (GPU)			
Mesh	FEniCS	Optimised	FEniCS	Optimised	Partial offload	Full offload	Lookup table	Rowwise
Uniform mesh 3	7.06	58.83	10.03	76.96	0.64	188.97	229.57	279.37
Cardiac mesh 1	6.60	64.38	10.46	71.48	0.39	229.78	220.57	320.67
Cardiac mesh 2	6.41	56.82	10.27	64.11	0.37	97.73	180.11	309.89
Cardiac mesh 3	6.38	59.05	10.67	69.61	0.35	85.34	165.31	292.08
Cardiac mesh 4	6.18	58.83	10.36	70.23	0.38	104.04	178.00	286.03

Table 4
Performance (in Mdof/s) of matrix assembly for Poisson's equation with quadratic (P2) elements on dual-socket AMD Epyc "Naples" 7601 and NVIDIA V100.

P2 elements	Naples (CPU)	NVIDIA V100 (GPU)		
Mesh	FEniCS	Full offload	Lookup table	Rowwise
Uniform mesh 3	11.5	61.1	13.1	1.5
Cardiac mesh 1	12.1	51.3	13.2	1.6
Cardiac mesh 2	12.1	46.0	13.0	1.5
Cardiac mesh 3	12.1	47.7	12.9	1.5
Cardiac mesh 4	12.0	48.8	13.0	1.5

that is needed, and the problem of thread divergence is avoided. Using the NVIDIA profiler, we can confirm that thread divergence is eliminated, and this alternative implementation yields a small improvement for *Cardiac mesh 4*, where the performance reaches 295 Mdof/s, and the achieved bandwidth is 560 GB/s. Although these results indicate that a slight improvement may be achieved, implementing this version of the rowwise assembly approach in an automated way would require further changes to the FEniCS form compiler. This is not easily achievable within the scope of this paper, but it may be worth pursuing in the future.

Finally, we compare the performance of our GPU-accelerated assembly in FEniCS with GPU-accelerated matrix assembly provided by the MFEM library [5]. For this comparison, we use the `ex1p` example code included in MFEM to assemble a matrix for Poisson's equation on a unit cube mesh consisting of 1 million linear, hexahedral elements. Using MFEM's "full assembly" mode and the CUDA backend, the overall matrix assembly takes 1.42 seconds on an NVIDIA V100 GPU. For a fairer comparison, we choose to exclude the time of MFEM spent on copying data between host and device and some additional processing. Profiling reveals that 234 ms is spent in the CUDA kernel `EADiffusionAssemble3D`, which computes element matrices for every element. The GPU-accelerated MFEM performance for this particular example thus corresponds to 4.27 Mdof/s, which lies somewhere between the "partial offload" and "full offload" approaches reported in Table 3. According to the NVIDIA profiler, this CUDA kernel of MFEM requires the maximum amount of 255 registers per thread and thus suffers from register spilling and low occupancy.

4.3. Quadratic elements

Table 4 compares CPU and GPU performance of matrix assembly for Poisson's equation when using quadratic (P2) elements. On the Naples CPU, performance is about 12 Mdof/s, so the assembly time per degree-of-freedom remains almost the same as for linear elements. On the NVIDIA V100 GPU, the cost per degree-of-freedom is higher than for linear elements, and the "full offload" strategy yields a performance of 46 to 61 Mdof/s. Thus, a speedup of about 4–5× is achieved by GPU offloading. Unlike linear elements, however, the lookup table and rowwise strategies result in lower performance. While all the three CUDA versions suffer from high register pressure, the NVRTC compiler reports 9.5 KiB of spill loads and stores for the lookup table and rowwise variants, which is about 7 times more than the "full offload" counterpart. Moreover, due to the increased element matrix size, the rowwise kernel performs considerably more redundant work compared

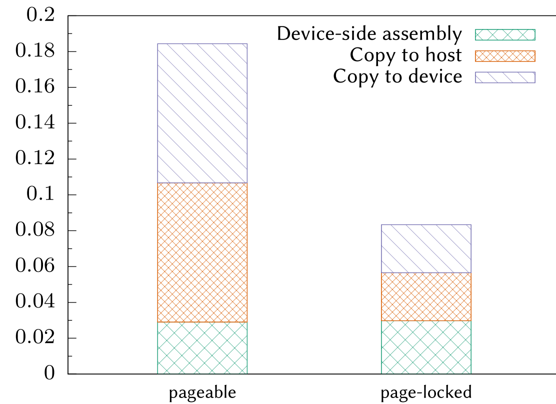


Fig. 1. Time (in seconds) spent in assembly and subsequent data copies between GPU device memory and main memory for the 3D Poisson's equation with *Cardiac mesh 4* on NVIDIA V100.

to the case of linear elements, and is thus not worthwhile for quadratic elements.

Note that as the polynomial order increases, the issue of register pressure becomes even more problematic. Improved code generation may alleviate the issue, but other strategies like matrix-free methods are better suited for reducing the overall memory traffic. Moreover, higher-order elements require more memory usage, which becomes a limiting factor when using a single GPU.

4.4. Impact of CPU–GPU data transfers

Fig. 1 compares the time spent on performing assembly with the time required to subsequently transfer data between host and device, as explained in Section 3.3. Measurements obtained with the NVIDIA profiler indicate that the effective bandwidths of these transfers are 4.6 and 13.0 GB/s for pageable and page-locked memory, respectively. Thus, even in the case of page-locked memory, where the effective bandwidth is quite close to the theoretical maximum bandwidth of 16 GB/s, data transfers nearly triple the overall global assembly time. Even worse, for the optimised, rowwise assembly, transferring global matrix values back and forth would increase the overall assembly time by a factor of about five. These results clearly stress the importance of avoiding unnecessarily transferring data between the CPU and GPU. In this case, for the assembled linear system to remain on the GPU, a fast GPU-based linear solver is needed.

4.5. GPU-accelerating a nonlinear hyperelasticity solver

Finally, we employ our automated code generation and GPU-offloading capabilities to investigate the performance of GPU-accelerated assembly and solution of a more challenging, nonlinear solid mechanics problem described by Ølgaard and Wells [40]. We note that the problem features a nonlinear, vector PDE with variable coefficients, a mixture of boundary conditions, and an unstructured

3D computational mesh. This illustrates the generality of the implemented form compiler and the automated code generation approach for GPU-based assembly and solution for finite element methods.

The physical problem is posed as finding a 3D displacement field $u : \Omega \to \mathbf{R}^3$ for a solid body by minimising the total potential energy of a hyperelastic material model:

$$\Pi(u) = \int_\Omega \psi(u)\,\mathrm{d}x - \int_\Omega B \cdot u\,\mathrm{d}x - \int_{\partial\Omega} T \cdot u\,\mathrm{d}s, \tag{4}$$

where B is a body force and T is a traction force on the boundary. Also, ψ is the stored strain energy density function, which, in this case, is based on a Neo-Hookean elastic stored energy model,

$$\psi = \frac{\mu}{2}(I_c - 3) - \mu \ln(J) + \frac{\lambda}{2} \ln(J)^2. \tag{5}$$

Here μ and λ are Lamé parameters that depend on the material, $J = \det(F)$, $I_c = \mathrm{trace}(C)$, $C = F^T F$ is the right Cauchy–Green tensor, and $F = I + \nabla u$ is the deformation gradient.

The variational formulation of the above minimisation problem is obtained from directional derivatives of the energy functional Π, leading to the following pair of forms:

$$L(u; v) = D_v \Pi = \lim_{\epsilon \to 0} \frac{d\Pi(u + \epsilon v)}{d\epsilon} \tag{6}$$

$$a(u; \delta u, v) = D_{\delta u} L = \lim_{\epsilon \to 0} \frac{dL(u + \epsilon \delta u; v)}{d\epsilon}. \tag{7}$$

For further details, we refer the reader to [40].

Newton's method is used to solve the nonlinear hyperelasticity problem. The solver converges when the L2 norm of the (absolute) difference in the approximate solutions from one iteration to the previous iteration falls below a tolerance of 10^{-12}. The linear system that arises during each Newton iteration is symmetric and positive definite, and we therefore solve it using the conjugate gradient (CG) method, with two different preconditioners and a convergence tolerance of 10^{-5} for the L2 norm of the relative residual. The first preconditioner is a simple diagonal scaling (Jacobi) method, whereas the second is a more advanced algebraic multigrid (AMG) method. The CG solver is PETSc's native implementation, which employs PETSc's own functions for sparse matrix–vector multiply, dot product and vector addition when running on the CPU. When the PETSc CG solver runs on the GPU, it uses NVIDIA's cuSPARSE [41] and cuBLAS [42] libraries to offload sparse matrix and vector operations. Furthermore, PETSc's native AMG preconditioner is used when running on the CPU, whereas Hypre's BoomerAMG [10] is used for AMG preconditioning on the GPU.

For the GPU-accelerated assembly, we employ the fully offloaded assembly algorithm with a lookup table, since it outperforms rowwise assembly for the current problem. Detailed profiling with the NVIDIA profiler reveals that the CUDA kernel for rowwise assembly requires higher register usage, which in turn leads to reduced occupancy and lower performance.

Table 5 shows the time and performance of assembly on NVIDIA A100 and on up to four dual-socket AMD Epyc "Milan" CPU nodes. Assembly performance is reported as the number of million degrees-of-freedom per second per Newton iteration. Table 6 shows the solver performance for both Jacobi and AMG preconditioning, reported as the number of million degrees-of-freedom per second per CG iteration.

Regarding assembly, a single NVIDIA A100 yields a speedup of 2.3–3.0× compared to four CPU nodes. With respect to the linear solver, performance depends much more on the problem size and the amount of cache reuse. In the case of Jacobi preconditioning, doubling the number of CPU nodes more than doubles performance as the total amount of cache memory increases and more data is read from cache rather than main memory. For example, *Cardiac mesh 4* involves about 9 million degrees of freedom, and the entire solution vector occupies approximately 70 MiB. Moreover, if we ignore the sparse matrix, a typical CG procedure requires the right-hand side, solution and three auxiliary vectors, yielding a memory footprint of about 350 MiB. Since each Milan node has 512 MiB of L3 cache and 64 MiB of L2 cache, the effective bandwidth of the CPUs thus greatly increases as a larger portion of the vector data resides in L2 cache.

With Jacobi preconditioning, NVIDIA A100 outperforms 2 CPU nodes by up to 58% for the largest mesh, while yielding a slowdown of about 30% for some of the smallest meshes. Although the Jacobi solver is accelerated considerably by moving to a GPU, it provides poor preconditioning and generally requires a large number of iterations. Table 6 shows that the convergence rate and overall performance for both CPU and GPU is greatly improved by using AMG preconditioning. For the largest uniform meshes, a single NVIDIA A100 GPU provides comparable performance to a single CPU node, whereas for the cardiac meshes, the NVIDIA A100 yields a speedup of about 1.7× compared to a single CPU node and reaches about 85% of the performance of 2 CPU nodes.

While this example shows that a state-of-the-art preconditioned linear solver can be accelerated by offloading to a GPU, the topic of fast and effective GPU-accelerated preconditioners is still an active research topic (see, e.g., [8,9,43–45]). We expect to see improvements in this area in the future.

5. Related work

The implementation of FEniCS is described in the FEniCS book [1], alongside several applications to solving PDEs. During the course of the development of FFC and other, related form compilers, numerous strategies have been explored with regards to generating and optimising code for computing element vectors and matrices. For example, a tensor representation [15,46,47] or computer algebra [17,18] can sometimes be used to simplify or compute integrals exactly. Other approaches include optimising tabulate tensor kernels based on numerical integration [48], various low-level loop optimisations [19–21], or explicit vectorisation [49]. In addition, [28,30] were able to optimise GPU kernels for computing element matrices through well-considered assignment of work to different threads and memory layout for data in global memory, as well as the use of shared memory to speed up certain memory accesses. Another approach, also based on code generation, is taken by libCEED [3], which allows user-defined, problem-specific functions to be defined at quadrature points, and uses just-in-time (JIT) compilation and kernel fusion to avoid unnecessarily moving intermediate results to and from device memory.

The translation from Unified Form Language to GPU kernels and a prototype compiler was discussed by Markall et al. [50]. However, prior work in the context of automated code generation only considered the "partial offloading" strategy [51,52]. This approach requires transferring large amounts of intermediate data between GPU and CPU for the element matrices and vectors. Moreover, partial offloading is rarely justified when using low-order finite elements, since calculating element matrices is relatively cheap. Instead, most of the time is spent in the final stage of adding values to the global matrix, which does not benefit from offloading in this approach. These issues can be somewhat mitigated, at least for higher-order elements, by resorting to *matrix-free* methods [5,6,29,53] and variants thereof [51], where global matrix assembly is avoided entirely. A matrix-free linear solver must then be used, and it must also be offloaded to the GPU. Although this may be a suitable approach in some scenarios, there are other situations where assembling a global matrix cannot be avoided, such as when using direct solvers or factorisation-based preconditioners.

Cecka, Lew and Darve [34] studied several different GPU-based algorithms for global assembly for the Poisson problem, including the usual cellwise algorithm, a rowwise algorithm, which we adopted in Section 3.7, and a third algorithm that assigns each non-zero of the global matrix to a separate thread. Experiments show that the third method can be worthwhile, though there are challenges related to limited availability of shared memory, load imbalance, and a large number of redundant operations being carried out.

Table 5
Total time (in seconds), number of Newton iterations, and performance (in Mdof/s/Newton iteration) of assembly for a 3D nonlinear hyperelasticity problem on an NVIDIA A100 GPU and up to 4 dual-socket AMD Epyc "Milan" 7763 CPU nodes.

		NVIDIA A100 (GPU)		1 Milan CPU node		2 Milan CPU nodes		4 Milan CPU nodes	
Mesh	Iterations	Time [s]	Mdof/s/it	Time [s]	Mdof/s/it	Time [s]	Mdof/s/it	Time [s]	Mdof/s/it
Uniform mesh 1	5	0.05	68.9	0.45	7.6	0.31	10.9	0.24	13.9
Uniform mesh 2	5	0.11	75.2	1.23	6.5	0.67	11.9	0.40	20.0
Uniform mesh 3	5	0.19	83.4	1.53	10.1	0.87	17.9	0.53	28.9
Uniform mesh 4	5	0.32	84.1	2.64	10.0	1.48	18.0	0.85	31.3
Uniform mesh 5	5	0.50	84.0	4.18	10.1	2.24	18.7	1.29	32.6
Uniform mesh 6	5	0.75	83.4	5.21	12.0	3.41	18.3	1.71	36.6
Uniform mesh 7	5	1.07	83.3	7.15	12.4	3.92	22.7	2.48	35.9
Uniform mesh 8	5	1.64	74.2	9.56	12.7	5.34	22.8	3.04	40.1
Cardiac mesh 1	6	0.26	88.1	2.04	11.1	1.12	20.1	0.70	32.2
Cardiac mesh 2	7	0.49	84.5	3.80	10.8	1.95	21.1	1.14	36.0
Cardiac mesh 3	7	0.57	82.5	4.10	11.4	2.23	21.0	1.29	36.4
Cardiac mesh 4	8	0.87	83.3	6.31	11.5	3.40	21.3	1.96	37.1

Table 6
Total time (in seconds), total number of CG iterations, and performance (in Mdof/s/CG iteration) for a CG solver with Jacobi or AMG preconditioning with respect to a 3D nonlinear hyperelasticity problem on an NVIDIA A100 GPU and up to 4 dual-socket AMD Epyc "Milan" 7763 CPU nodes. The CG solver with AMG preconditioning runs out of memory for "Uniform mesh 8" on NVIDIA A100.

	NVIDIA A100 (GPU)			1 Milan CPU node			2 Milan CPU nodes			4 Milan CPU nodes		
Mesh	Time [s]	Iterations	Mdof/s/it	Time [s]	Iterations	Mdof/s/it	Time [s]	Iterations	Mdof/s/it	Time [s]	Iterations	Mdof/s/it
CG/Jacobi												
Uniform mesh 1	2.5	4 616	1239.8	3.3	4 611	962.4	3.5	4 608	901.5	3.5	4 612	905.3
Uniform mesh 2	6.8	6 077	1433.2	9.9	6 072	981.3	8.6	6 077	1127.5	6.2	5 995	1533.6
Uniform mesh 3	13.2	7 259	1703.1	26.7	7 257	841.2	8.9	7 256	2527.1	5.2	7 256	4312.6
Uniform mesh 4	27.7	9 091	1742.5	67.7	9 088	713.3	38.7	9 090	1248.1	10.8	9 086	4453.1
Uniform mesh 5	49.3	10 571	1802.5	144.4	10 566	615.4	73.6	10 562	1207.1	37.0	10 569	2400.8
Uniform mesh 6	82.5	12 064	1830.3	241.2	12 061	626.0	128.5	12 061	1175.2	65.1	12 064	2321.2
Uniform mesh 7	133.3	13 552	1808.2	388.3	13 554	621.0	202.9	13 553	1188.5	109.1	13 555	2211.1
Uniform mesh 8	194.9	14 870	1858.7	612.7	14 874	591.3	307.0	14 873	1180.2	165.0	14 875	2196.4
Cardiac mesh 1	44.0	22 041	1888.5	91.1	21 886	904.8	31.7	21 996	2615.5	11.2	21 923	7344.2
Cardiac mesh 2	79.2	26 230	1945.5	193.0	26 277	800.3	82.0	26 225	1879.2	23.6	26 167	6508.5
Cardiac mesh 3	98.1	28 599	1947.2	241.5	28 597	791.2	107.4	28 601	1779.3	33.6	28 601	5686.9
Cardiac mesh 4	145.4	31 058	1934.5	378.8	31 330	749.3	181.1	31 160	1559.1	67.4	31 014	4166.5
CG/AMG												
Uniform mesh 1	2.6	182	47.3	1.9	147	52.7	2.0	150	52.5	2.2	145	45.4
Uniform mesh 2	5.3	175	53.1	4.4	153	55.2	2.7	153	90.9	3.0	151	80.0
Uniform mesh 3	9.5	178	58.2	8.6	159	57.3	5.4	156	90.0	4.0	159	122.7
Uniform mesh 4	17.3	180	55.4	15.6	162	55.3	8.3	161	102.9	5.2	164	168.0
Uniform mesh 5	27.5	183	56.0	25.2	171	57.0	14.0	168	100.6	8.5	170	168.7
Uniform mesh 6	36.9	183	62.0	36.9	168	57.0	20.4	172	105.8	11.8	175	185.4
Uniform mesh 7	52.9	187	62.9	53.3	176	58.8	29.1	177	108.3	15.7	174	197.1
Uniform mesh 8	—	—	—	74.6	180	58.7	40.2	180	109.1	22.0	182	201.9
Cardiac mesh 1	11.3	445	149.0	18.2	360	74.7	9.2	364	149.9	4.7	347	281.1
Cardiac mesh 2	18.9	493	153.0	32.4	399	72.3	16.3	398	143.1	11.6	383	193.5
Cardiac mesh 3	21.5	466	144.5	37.9	407	71.7	18.3	391	143.0	10.1	387	255.3
Cardiac mesh 4	33.7	557	150.0	56.8	435	69.3	28.3	433	138.4	17.3	442	231.5

For a time-dependent, nonlinear diffusion equation, Pichler and Haase [54] offload assembly to a GPU using the CUDA C++ template library Thrust [55], whose high-level interface is used to avoid writing CUDA kernels. Moreover, global assembly is carried out using a precomputed lookup table to add element matrix values to a global matrix, in the same way that is described in Section 3.6.

An example of GPU acceleration of both assembly and solution of linear systems for a finite element problem is given by Fu et al. [56], including a more advanced linear solver based on the conjugate gradient method with an algebraic multigrid preconditioner. Reguly and Giles [23] thoroughly investigate GPU acceleration for both assembly and solution of linear systems for the Poisson problem, comparing global assembly using different sparse matrix formats to matrix-free methods. Roughly speaking, the linear solver benefits from global assembly whenever first-order elements are used, while other strategies may be advantageous for second-, third- and fourth-order elements. Further research on matrix-free methods has been conducted by Ljungkvist [53] and Kronbichler and Ljungkvist [57] as well as in the context of libCEED [3,58].

6. Conclusion

We have enabled an automated and seamless GPU offloading for both the assembly and solution phases of finite element computations prescribed in the high-level UFL format. This is achieved by extending the FEniCS framework's form compiler, FFC, to automatically generate CUDA code, as well as extending the DOLFIN library with supporting CUDA kernels for offloading the entire finite element assembly procedure. Our experiments show that GPU acceleration is achieved for linear system assembly in the case of 1st and 2nd-order elements on affine, tetrahedral meshes. Moreover, GPU offloading shows the most promise for problems where assembly accounts for a significant part of the execution time, for example, if assembly is performed repeatedly or a fast GPU-accelerated linear solver can be used. As an example, we offload an advanced, nonlinear PDE solver to illustrate the potential for GPU acceleration while still retaining the ease of use offered by UFL as a high-level, domain-specific language for finite element problems.

Furthermore, the GPU-offloaded assembly can be further sped up by replacing costly binary searches with the use of a precomputed lookup

table. Performing the assembly row by row is also shown to improve performance, at least for Poisson's equation with first-order elements.

We stress the need for paying careful attention to CPU–GPU data transfers to successfully accelerate finite element computations. Otherwise, the cost of transferring data can easily outweigh the performance improvements achieved by offloading in the first place. From this perspective, our strategy is different from that chosen by many other libraries, which are based on separately offloading the assembly and solution phases to a GPU. Indeed, a high-level view of the entire PDE solving procedure is necessary to benefit from seamlessly integrated GPU acceleration.

Users of the FEniCS framework can now benefit from GPU-offloading without needing to sacrifice the other advanced features of FEniCS. In a wider context, our approach should readily extend to also support AMD GPUs through the HIP programming model. Thus, for automated finite element codes, easy and efficient use of yet more exascale-class GPUs is undoubtedly within reach. In the future, we plan to extend our current work to the use of multiple GPUs in a distributed setting, thus enabling GPU-acceleration for much larger problems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

This work was supported by the Research Council of Norway under contract 251186 and by the European High-Performance Computing Joint Undertaking grant agreement 956213 (SparCity). Also, the research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

[1] A. Logg, K.-A. Mardal, G.N. Wells (Eds.), Automated Solution of Differential Equations by the Finite Element Method, Springer, Berlin, 2012, http://dx.doi.org/10.1007/978-3-642-23099-8.

[2] F. Rathgeber, D.A. Ham, L. Mitchell, M. Lange, F. Luporini, A.T.T. Mcrae, G.-T. Bercea, G.R. Markall, P.H.J. Kelly, Firedrake: Automating the finite element method by composing abstractions, ACM Trans. Math. Software 43 (3) (2016) http://dx.doi.org/10.1145/2998441.

[3] A. Abdelfattah, V. Barra, N. Beams, R. Bleile, J. Brown, J.-S. Camier, R. Carson, N. Chalmers, V. Dobrev, Y. Dudouit, P. Fischer, A. Karakus, S. Kerkemeier, T. Kolev, Y.-H. Lan, E. Merzari, M. Min, M. Phillips, T. Rathnayake, R. Rieben, T. Stitt, A. Tomboulides, S. Tomov, V. Tomov, A. Vargas, T. Warburton, K. Weiss, GPU algorithms for efficient exascale discretizations, Parallel Comput. 108 (2021) http://dx.doi.org/10.1016/j.parco.2021.102841.

[4] R.T. Mills, M.F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S.E. Kruger, H. Morgan, T. Munson, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, J. Zhang, Toward performance-portable PETSc for GPU-based exascale systems, Parallel Comput. 108 (2021) http://dx.doi.org/10.1016/j.parco.2021.102831.

[5] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, S. Zampini, MFEM: A modular finite element methods library, Comput. Math. Appl. 81 (2021) 42–74, http://dx.doi.org/10.1016/j.camwa.2020.06.009.

[6] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The deal.II finite element library: Design, features, and insights, Comput. Math. Appl. 81 (2021) 407–422, http://dx.doi.org/10.1016/j.camwa.2020.02.022.

[7] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, R. Strzodka, AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods, SIAM J. Sci. Comput. 37 (5) (2015) S602–S626, http://dx.doi.org/10.1137/140980260, arXiv:https://doi.org/10.1137/140980260.

[8] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, M. Köhler, Preconditioned Krylov solvers on GPUs, Parallel Comput. 68 (2017) 32–44, http://dx.doi.org/10.1016/j.parco.2017.05.006.

[9] H. Anzt, E. Boman, R. Falgout, P. Ghysels, M. Heroux, X. Li, L.C. McInnes, R.T. Mills, S. Rajamanickam, K. Rupp, B. Smith, I. Yamazaki, U.M. Yang, Preparing sparse solvers for exascale computing, Phil. Trans. R. Soc. A 378 (2166) (2020) http://dx.doi.org/10.1098/rsta.2019.0053.

[10] R.D. Falgout, R. Li, B. Sjögreen, L. Wang, U.M. Yang, Porting hypre to heterogeneous computer architectures: Strategies and experiences, Parallel Comput. 108 (2021) http://dx.doi.org/10.1016/j.parco.2021.102840.

[11] X.S. Li, P. Lin, Y. Liu, P. Sao, Newly released capabilities in the distributed-memory SuperLU sparse direct solver, ACM Trans. Math. Software 49 (1) (2023) http://dx.doi.org/10.1145/3577197.

[12] P.G. Ciarlet, The Finite Element Method for Elliptic Problems, in: Classics in Applied Mathematics, Society for Industrial and Applied Mathematics, Philadelphia, 2002.

[13] M.S. Alnæs, A. Logg, K.B. Ølgaard, M.B. Rognes, G.N. Wells, Unified form language: A domain-specific language for weak formulations of partial differential equations, ACM Trans. Math. Software 40 (2) (2014) http://dx.doi.org/10.1145/2566630.

[14] D.N. Arnold, A. Logg, Periodic table of the finite elements, SIAM News 47 (2014) URL http://www.femtable.org/.

[15] R.C. Kirby, A. Logg, A compiler for variational forms, ACM Trans. Math. Software 32 (3) (2006) 417–444, http://dx.doi.org/10.1145/1163641.1163644.

[16] P. Keast, Moderate-degree tetrahedral quadrature formulas, Comput. Methods Appl. Mech. Engrg. 55 (3) (1986) 339–348, http://dx.doi.org/10.1016/0045-7825(86)90059-9.

[17] M.S. Alnæs, K.-A. Mardal, On the efficiency of symbolic computations combined with code generation for finite element methods, ACM Trans. Math. Software 37 (1) (2010) http://dx.doi.org/10.1145/1644001.1644007.

[18] F.P. Russell, P.H.J. Kelly, Optimized code generation for finite element local assembly using symbolic manipulation, ACM Trans. Math. Software 39 (4) (2013) http://dx.doi.org/10.1145/2491491.2491496.

[19] F. Luporini, A.L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D.A. Ham, P.H.J. Kelly, Cross-loop optimization of arithmetic intensity for finite element local assembly, ACM Trans. Archit. Code Optim. 11 (4) (2015) http://dx.doi.org/10.1145/2687415.

[20] F. Luporini, D.A. Ham, P.H.J. Kelly, An algorithm for the optimization of finite element integration loops, ACM Trans. Math. Software 44 (1) (2017) http://dx.doi.org/10.1145/3054944.

[21] M. Homolya, L. Mitchell, F. Luporini, D.A. Ham, TSFC: A structure-preserving form compiler, SIAM J. Sci. Comput. 40 (3) (2018) 401–428, http://dx.doi.org/10.1137/17M1130642.

[22] NVIDIA Corporation, CUDA driver API: API reference manual, 2019, URL https://docs.nvidia.com/cuda/cuda-driver-api/.

[23] I.Z. Reguly, M.B. Giles, Finite element algorithms and data structures on graphical processing units, Int. J. Parallel Program. 43 (2) (2015) 203–239, http://dx.doi.org/10.1007/s10766-013-0301-6.

[24] J.D. Trotter, X. Cai, S.W. Funke, On memory traffic and optimisations for low-order finite element assembly algorithms on multi-core CPUs, ACM Trans. Math. Software 48 (2) (2022) http://dx.doi.org/10.1145/3503925.

[25] NVIDIA Corporation, NVIDIA Tesla V100 GPU architecture, 2017, URL https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[26] NVIDIA Corporation, CUDA C++ programming guide, 2022, URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[27] B. Barsdell, K. Clark, Jitify: CUDA C++ runtime compilation made easy, in: GPU Technology Conference 2017, San Jose, CA, 2017.

[28] M.G. Knepley, A.R. Terrel, Finite element integration on GPUs, ACM Trans. Math. Software 39 (2) (2013) http://dx.doi.org/10.1145/2427023.2427027.

[29] K. Świrydowicz, N. Chalmers, A. Karakus, T. Warburton, Acceleration of tensor-product operations for high-order finite element methods, Int. J. High Perform. Comput. Appl. 33 (4) (2019) 735–757, http://dx.doi.org/10.1177/1094342018816368.

[30] K. Banaś, P. Płaszewski, P. Macioł, Numerical integration on GPUs for higher order finite elements, Comput. Math. Appl. 67 (6) (2014) 1319–1344, http://dx.doi.org/10.1016/j.camwa.2014.01.021.

[31] NVIDIA Corporation, NVRTC – CUDA runtime compilation user guide, 2019, URL https://docs.nvidia.com/cuda/nvrtc/.

[32] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W.D. Gropp, D. Karpeyev, D. Kaushik, M.G. Knepley, D.A. May, L.C. McInnes, R.T. Mills, T. Munson, K. Rupp, P. Sanan, B.F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, 2019, URL https://www.mcs.anl.gov/petsc.

[33] NVIDIA Corporation, CUDA toolkit, 2022, URL https://developer.nvidia.com/cuda-toolkit.
[34] C. Cecka, A.J. Lew, E. Darve, Assembly of finite element methods on graphics processors, Internat. J. Numer. Methods Engrg. 85 (5) (2010) 640–669, http://dx.doi.org/10.1002/nme.2989.
[35] T. Deakin, J. Price, M. Martineau, S. McIntosh-Smith, Evaluating attainable memory bandwidth of parallel programming models via BabelStream, Int. J. Comput. Sci. Eng. 17 (3) (2018) 247–262, http://dx.doi.org/10.1504/IJCSE.2018.095847.
[36] J.D. McCalpin, STREAM: Sustainable memory bandwidth in high performance computers, 2013, URL https://www.cs.virginia.edu/stream/.
[37] J.D. Trotter, Software for "Targeting performance and user-friendliness: GPU-accelerated finite element computation with automated code generation in FEniCS", 2023, http://dx.doi.org/10.5281/zenodo.7854931.
[38] M. Marciniak, H. Arevalo, J. Tfelt-Hansen, T. Jespersen, R. Jabbari, C. Glinge, K.A. Ahtarovski, N. Vejlstrup, T. Engstrom, M.M. Maleckar, K. McLeod, From CMR image to patient-specific simulation and population-based analysis: Tutorial for an openly available image-processing pipeline, in: T. Mansi, K. McLeod, M. Pop, K. Rhode, M. Sermesant, A. Young (Eds.), STACOM 2016: Statistical Atlases and Computational Models of the Heart. Imaging and Modelling Challenges, Springer International Publishing, Cham, 2017, pp. 106–117, http://dx.doi.org/10.1007/978-3-319-52718-5_12.
[39] R. Jabbari, T. Engstrøm, C. Glinge, B. Risgaard, J. Jabbari, B.G. Winkel, C.J. Terkelsen, H.-H. Tilsted, L.O. Jensen, M. Hougaard, S.E. Chiuve, F. Pedersen, J.H. Svendsen, S. Haunsø, C.M. Albert, J. Tfelt-Hansen, Incidence and risk factors of ventricular fibrillation before primary angioplasty in patients with first ST-elevation myocardial infarction: a nationwide study in Denmark, J. Am. Heart Assoc. 4 (1) (2015) http://dx.doi.org/10.1161/JAHA.114.001399.
[40] K.B. Ølgaard, G.N. Wells, Applications in solid mechanics, in: A. Logg, K.-A. Mardal, G.N. Wells (Eds.), Automated Solution of Differential Equations by the Finite Element Method, Springer, Berlin, 2012, pp. 505–526, http://dx.doi.org/10.1007/978-3-642-23099-8, Ch. 26.
[41] NVIDIA Corporation, cuSPARSE library, 2022, URL https://docs.nvidia.com/cuda/cusparse/index.html.
[42] NVIDIA Corporation, cuBLAS library user guide, 2022, URL https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.
[43] H. Liu, B. Yang, Z. Chen, Accelerating algebraic multigrid solvers on NVIDIA GPUs, Comput. Math. Appl. 70 (5) (2015) 1162–1181, http://dx.doi.org/10.1016/j.camwa.2015.07.005.
[44] Y. Chen, X. Tian, H.a. Liu, Parallel ILU preconditioners in GPU computation, Soft Comput. 22 (2018) 8187–8205, http://dx.doi.org/10.1007/s00500-017-2764-7.
[45] J.I. Aliaga, E. Dufrechou, P. Ezzatti, E.S. Quintana-Ortí, An efficient GPU version of the preconditioned GMRES method, J. Supercomput. 75 (2019) 1455–1469, http://dx.doi.org/10.1007/s11227-018-2658-1.
[46] R.C. Kirby, M. Knepley, A. Logg, L.R. Scott, Optimizing the evaluation of finite element matrices, SIAM J. Sci. Comput. 27 (3) (2005) 741–758, http://dx.doi.org/10.1137/040607824.
[47] M.E. Rognes, R.C. Kirby, A. Logg, Efficient assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ conforming finite elements, SIAM J. Sci. Comput. 31 (6) (2009) 4130–4151, http://dx.doi.org/10.1137/08073901X.
[48] K.B. Ølgaard, G.N. Wells, Optimizations for quadrature representations of finite element tensors through automated code generation, ACM Trans. Math. Software 37 (1) (2010) http://dx.doi.org/10.1145/1644001.1644009.
[49] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D.A. Ham, P.H. Kelly, A study of vectorization for matrix-free finite element methods, Int. J. High Perform. Comput. Appl. (2020) http://dx.doi.org/10.1177/1094342020945005.
[50] G.R. Markall, D.A. Ham, P.H. Kelly, Towards generating optimised finite element solvers for GPUs from high-level specifications, Procedia Comput. Sci. 1 (1) (2010) 1815–1823, http://dx.doi.org/10.1016/j.procs.2010.04.203, ICCS 2010.
[51] G.R. Markall, A. Slemmer, D.A. Ham, P.H.J. Kelly, C.D. Cantwell, S.J. Sherwin, Finite element assembly strategies on multi-core and many-core architectures, Internat. J. Numer. Methods Fluids 71 (1) (2013) 80–97, http://dx.doi.org/10.1002/fld.3648.
[52] G.R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D.A. Ham, P.H. Kelly, Performance-portable finite element assembly using PyOP2 and FEniCS, in: J.M. Kunkel, T. Ludwig, H.W. Meuer (Eds.), 28th International Supercomputing Conference, ISC 2013, in: Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2013, pp. 279–289, http://dx.doi.org/10.1007/978-3-642-38750-0_21.
[53] K. Ljungkvist, Matrix-free finite-element operator application on graphics processing units, in: L. Lopes, J. Žilinskas, A. Costan, R.G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S.L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, M. Alexander (Eds.), Euro-Par 2014: Parallel Processing Workshops, Springer International Publishing, Cham, 2014, pp. 450–461, http://dx.doi.org/10.1007/978-3-319-14313-2_38.
[54] F. Pichler, G. Haase, Finite element method completely implemented for graphic processor units using parallel algorithm libraries, Int. J. High Perform. Comput. Appl. 33 (1) (2017) 53–66, http://dx.doi.org/10.1177/1094342017694703.
[55] NVIDIA Corporation, Thrust quick start guide, 2020, URL https://docs.nvidia.com/cuda/thrust/.
[56] Z. Fu, T.J. Lewis, R.M. Kirby, R.T. Whitaker, Architecting the finite element method pipeline for the GPU, J. Comput. Appl. Math. 257 (2014) 195–211, http://dx.doi.org/10.1016/j.cam.2013.09.001.
[57] M. Kronbichler, K. Ljungkvist, Multigrid for matrix-free high-order finite element computations on graphics processors, ACM Trans. Parallel Comput. 6 (1) (2019) http://dx.doi.org/10.1145/3322813.
[58] J. Brown, V. Barra, N. Beams, L. Ghaffari, M. Knepley, W. Moses, R. Shakeri, K. Stengel, J.L. Thompson, J. Zhang, Performance portable solid mechanics via matrix-free p-multigrid, 2022, http://dx.doi.org/10.48550/arXiv.2204.01722, arXiv:2204.01722.