MATRIX-FREE FINITE-ELEMENT COMPUTATIONS ON GRAPHICS PROCESSORS WITH ADAPTIVELY REFINED UNSTRUCTURED MESHES

Karl Ljungkvist

Department of Information Technology
Division of Scientific Computing
Uppsala University
Box 337
SE-751 05 Uppsala, Sweden
karl.ljungkvist@it.uu.se

ABSTRACT

This paper concerns efficient matrix-free finite-element algorithms on modern manycore processors such as graphics cards (GPUs) as an alternative to sparse matrix-vector products. In matrix-free finite element algorithms, the assembly and solution phases are merged, yielding a significantly lower memory bandwidth footprint, with a corresponding increase in efficiency on bandwidth limited processors. Additionally, no system matrix must be assembled or stored in memory.

We present a GPU parallelization of the matrix-free method including a novel algorithm for resolving hanging-node constraints on the GPU, capable of simulation on adaptively refined grids. For second-order elements and higher in 3D, our GPU implementation of the adaptive algorithm is between 1.8 and 2.3 times faster than an existing optimized CPU version, on comparable hardware. Compared to a matrix-based implementation using CUSPARSE, we get a speedup of 8 and can solve problems 8 times larger in 3D.

Keywords: finite element methods, GPU, matrix free, adaptive refinement, hanging nodes.

1 INTRODUCTION

The finite element method is a popular choice for numerical simulation due to its capability to easily handle complicated geometries and incorporate adaptive mesh refinement. The conventional procedure for finite-element computations is to first assemble a system of equations, and solve this using an iterative method. However, this two-step approach performs poorly when executed on modern multicore and manycore processors.

The computational core of a finite-element solver is a product between the large and sparse system matrix and a vector, which is performed a large number of times inside the iterative solver. This operation, the sparse-matrix vector product (*SpMV*), needs to fetch an 8-byte double for only every 2 floating point operations (flops) performed, whereas the memory system of most modern CPUs and GPUs can only deliver one double for every 32-64 flops. This means that the performance of the SpMV operation will be solely limited by the available memory bandwidth, and most of the computational hardware will be wasted (see, e.g., Gropp, Kaushik, Keyes, and Smith 1999). In addition to this, the system matrix must be stored in memory which can be a quite severe limit on how large problems can be solved, in particular on GPUs which typically have an

SpringSim-HPC 2017, April 23-26, Virginia Beach, VA, USA
©2017 Society for Modeling & Simulation International (SCS)

order of magnitude smaller memory. Also, the matrix assembly itself can amount to a substantial portion of the total simulation time, especially in applications where frequent reassembly is necessary.

Motivated by these shortcomings a matrix-free approach has been suggested, for the first time already in 1986 for computer systems with limited memory (Carey and Jiang 1986). This idea builds on the important observation that, within the iterative linear solver, the matrix entries are not needed explicitly, but only a recipe for computing the product of the matrix times a vector. The matrix-free multiplication algorithm is especially interesting for finite elements with tensor-product basis functions, such as quadrilateral and hexahedral elements, since for such elements the element-local numerical integration can be performed very efficiently using a sum-factorization approach. As shown by Cantwell, Sherwin, Kirby, and Kelly (2011), the matrix-free approach accesses less data than when using SpMV for elements of order two and higher, whereas the converse is true for first-order elements. Consequently, for order two and higher, the matrix-free algorithm can be expected to perform better on modern bandwidth-limited processors. In Brown (2010), a matrix-free approach is used for a high-order Jacobian combined with an assembled lower-order preconditioner. In Kormann and Kronbichler (2011), the authors present a generic framework for matrix-free computations as part of the open-source finite-element library `deal.II` (Bangerth et al. 2016). The framework uses vectorization, task-based parallelization, and message passing to achieve good performance in a quantum mechanics computation on distributed and shared-memory systems (Kronbichler and Kormann 2012).

Early work on finite-element computations on GPUs focused either on the SpMV operation in the solve phase (Göddeke, Strzodka, and Turek 2005, Dehnavi, Fernandez, and Giannacopoulos 2010), or on the matrix assembly (Cecka, Lew, and Darve 2011, Markall et al. 2013). The matrix-free approach was first used on GPUs in high-order discontinuous Galerkin (DG) or spectral element methods for explicit time stepping of hyperbolic problems. In Klöckner, Warburton, Bridge, and Hesthaven (2009), high-order DG elements are used in a simulation of Maxwell's equations on conservative form. In Komatitsch, Erlebacher, Göddeke, and Michéa (2010), the authors simulate seismic wave propagation in 3D using fourth-order spectral elements on several GPUs. We are not aware of any previous effort conducting finite-element computations with hanging nodes on GPUs.

In a previous paper, we have studied the performance of a matrix-free operator application for a Cartesian mesh where a constant local matrix can be used (Ljungkvist 2014). The present article extends that work to unstructured meshes, and to adaptively refined meshes with hanging nodes. We have implemented the method as a general framework with an underlying GPU parallelization based on CUDA (NVIDIA Corporation 2016), which is in the process of being made officially available as part of `deal.II`. With a unified interface for CPU and GPU backends, which will be the focus of an upcoming publication, we can run the same application code efficiently on both systems, and at the same time offer great flexibility for the application programmer.

The remainder of this paper is structured as follows. In Section 2, we describe the matrix-free algorithm in detail, as well as how we parallelize it for GPUs. In Section 3, our algorithm for resolving hanging-node constraints on the GPU is described. In Section 4, we elaborate on the data structures used. In Section 5, we present benchmark experiments evaluating the performance of our method. Section 6 concludes this work.

2 MATRIX-FREE MULTIPLICATION

The linear system originating from a finite-element discretization of a PDE has a system matrix A which is assembled as a sum of local matrices a_k on all elements k in the mesh. For a thorough introduction to finite-element methods, see, e.g., Brenner and Scott (2002). Inside the linear solver, we want to compute the product of A with a vector u of unknowns, or *degrees of freedom* (DoFs). Now, instead of using a precomputed A, the matrix-free multiplication algorithm is formed by merging the matrix assembly into the multiplication resulting in the following three steps for each element k,

1. Read the local unknowns u_k from the global input vector u
2. Evaluate the local matrix multiplication, $v_k = a_k u_k$
3. Assemble the local contribution v_k into the global result vector v

2.1 Local Matrix Multiplication

In Ljungkvist (2014), we showed that for a Cartesian mesh where all the a_k are equal, using a single precomputed local matrix a gives a competitive algorithm as long as a is small enough to fit in the GPU cache. However, for a general mesh, the distinct a_k would amount to more memory than the corresponding assembled sparse matrix, and thus cannot be favorable neither in terms of storage space nor in terms of reduced bandwidth usage. In this case, a more advanced approach is necessary.

To simplify the discussion, we will now assume that the original PDE is a Poisson equation with variable coefficients. While this is a simple model problem, it serves the purpose of illustrating the method, and can readily be extended to, for instance, a vector valued or non-linear problem. It also appears as part of many more complicated applications, such as in projection-correction methods in computational fluid dynamics.

For the Poisson equation the local matrix is defined by

$$a_{ij}^k = \int_{\Omega_k} \nabla\varphi_i \cdot A(\mathbf{x}) \nabla\varphi_j \mathrm{d}\mathbf{x}, \tag{1}$$

where $A(\mathbf{x})$ is a variable coefficient. If we evaluate the integral by transforming to a reference element and integrating numerically using Gaussian quadrature, we obtain

$$a_{ij}^k = \sum_q \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q)\right) \cdot A(\boldsymbol{\xi}_q) \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q)\right) w_q |\det J_k(\boldsymbol{\xi}_q)|, \tag{2}$$

where the $\boldsymbol{\xi}_q$ and w_q are reference-element quadrature points and weights respectively. Furthermore, J_k^{-1} is the inverse Jacobian of the transformation from element k to the reference element, and $\nabla_{\boldsymbol{\xi}}$ denotes a reference-space gradient. Here, we see that only the variable coefficient $A(\boldsymbol{\xi}_q)$, the inverse Jacobians and Jacobian determinant need to be stored individually for each element. For Q_p elements in d dimensions integrated using $(p+1)^d$ quadrature points, these amount to $(d^2+2)(p+1)^d$ entries which is less than the $(p+1)^d(p+1)^d$ entries of the local matrix already for element degree $p > 1$.

Motivated by this observation, we plug (2) into the local multiplication,

$$v_i = \sum_j \sum_q \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q)\right) \cdot A(\boldsymbol{\xi}_q) \left(J_k^{-1}(\boldsymbol{\xi}_q)\nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q)\right) w_q |\det J_k(\boldsymbol{\xi}_q)| u_j. \tag{3}$$

Rearranging the summations, this operation can be performed by three successive steps. In the first one, we compute the gradients at reference quadrature points,

$$\nabla_{\boldsymbol{\xi}} u_q = \sum_j \nabla_{\boldsymbol{\xi}}\varphi_j(\boldsymbol{\xi}_q) u_j. \tag{4}$$

In the second step, we perform quadrature-point-wise operations; we first transform to the real element, then perform any local operations – in this case multiplying by the coefficient $A(\boldsymbol{\xi}_q)$, transform back to reference element and multiply by quadrature weights and Jacobian determinant,

$$s_q = w_q |\det J_k(\boldsymbol{\xi}_q)| J_k^{-T}(\boldsymbol{\xi}_q) A(\boldsymbol{\xi}_q) J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} u_q. \tag{5}$$

In the last step, we multiply by basis function gradients and integrate,

$$v_i = \sum_q \nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}_q) \cdot s_q. \tag{6}$$

2.2 Evaluation Using Sum-Factorization

As we saw, formulating a matrix-free algorithm leads to a significant reduction in the bandwidth footprint, but this comes at the price of additional operations. In particular, both when evaluating the quadrature point gradients in (4) and when performing the final integration in (6), d matrix-vector products with vectors of size $(p+1)^d$ must be performed. However, for tensor-product elements, such as quadrilateral and hexahedral elements, considerable simplifications can be made.

For such elements, the reference-element shape functions are tensor products of one-dimensional shape functions, i.e. in 3D,

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi)\psi_\nu(\eta)\psi_\sigma(\zeta)\,, \tag{7}$$

where we have introduced a multi index (μ,ν,σ) corresponding to the single index i, with each component running from 1 to $p+1$. For the basis function gradients, we get

$$\nabla_{\boldsymbol{\xi}}\varphi_i(\boldsymbol{\xi}) = \begin{pmatrix} \psi'_\mu(\xi)\psi_\nu(\eta)\psi_\sigma(\zeta) \\ \psi_\mu(\xi)\psi'_\nu(\eta)\psi_\sigma(\zeta) \\ \psi_\mu(\xi)\psi_\nu(\eta)\psi'_\sigma(\zeta) \end{pmatrix}. \tag{8}$$

Similarly, the 3D quadrature points can be defined by a tensor product of the one-dimensional points,

$$\boldsymbol{\xi}_q = (\xi_\alpha, \xi_\beta, \xi_\gamma)\,, \tag{9}$$

where $q \to (\alpha,\beta,\gamma)$ is another multi index. Note that we consistently use μ,ν,σ to index in DoF space, and α,β,γ as indices for quadrature points. Using these new indices and the shorthands $\psi_{\alpha\mu} = \psi_\mu(\xi_\alpha)$ and $\vartheta_{\alpha\mu} = \psi'_\mu(\xi_\alpha)$, we can factorize the sum in (4) and get

$$\nabla_{\boldsymbol{\xi}} u_{\alpha\beta\gamma} = \sum_\mu \begin{pmatrix} \vartheta_{\alpha\mu} \\ \psi_{\alpha\mu} \\ \psi_{\alpha\mu} \end{pmatrix} \sum_\nu \begin{pmatrix} \psi_{\beta\nu} \\ \vartheta_{\beta\nu} \\ \psi_{\beta\nu} \end{pmatrix} \sum_\sigma \begin{pmatrix} \psi_{\gamma\sigma} \\ \psi_{\gamma\sigma} \\ \vartheta_{\gamma\sigma} \end{pmatrix} u_{\mu\nu\sigma}\,, \tag{10}$$

where the vector products are to be understood as element-wise multiplication. The same series of operations are obtained for the integration in (6), but with a summation over the quadrature indices instead. In (10), we have a series of d consecutive $(p+1)^2 \times (p+1)^d$ tensor contractions for each d components, which have an operational complexity of $\mathscr{O}\left(2d^2(p+1)^{d+1}\right)$ altogether, compared to $\mathscr{O}\left(2d(p+1)^{2d}\right)$ for the large matrix-vector products in (4). For more details on the sum-factorization evaluation, see Kronbichler and Kormann (2012).

2.3 Parallelization

The matrix-free operator application algorithm contains several types of parallelism. The most apparent one is that the result is computed as a sum of independent contributions from each element. It is thus natural to parallelize the algorithm over the elements, i.e., divide the list of elements into chunks and compute the contribution from each chunk of elements in parallel.

This leads to fairly coarse-grained parallel work tasks consisting of all the local operations on an element. For each element, we first read the local DoF values into local variables and then evaluate values and/or gradients at quadrature points. Then, quadrature-point-local operations such as multiplication with a variable coefficient need to be performed. Finally, we perform numerical quadrature to obtain DoF values, and write back the local DoF values. Throughout these computations, each thread needs to store the local DoF values

and the values and/or gradients at quadrature points, plus any additional intermediate variables. These can amount to quite a large quantity of memory per thread.

On a multicore CPU, having a coarse-grained parallelization is usually desirable since per-task overhead is often quite substantial. Also, the amount of local data needed is not an issue, since the threads have a relatively large local memory in the form of cache, which can accommodate all the variables that are necessary for the local operations.

On a GPU on the other hand, parallelism is used to hide memory latency by having more threads than cores so that when a high-latency instruction is encountered, instructions from other threads can be executed instead. Therefore, a higher level of parallelism is often sought for when targeting GPUs. In addition, the per-core local memory is relatively limited, so a too high memory requirement per thread will put a limit on the number of threads per core that can be in flight at once. This motivates looking for more fine-grained alternatives to a parallelization over elements.

From Section 2.2, we recall that the sum-factorization resulted in a series of dense tensor contractions, each of which is of the form

$$v_{\alpha\nu\sigma} = \sum_{\mu} \psi_{\alpha\mu} u_{\mu\nu\sigma} \,. \tag{11}$$

Noting that this operation is essentially a matrix-vector product of ψ with each of the "rows" of v along a given coordinate direction, it is clear that the tensor contractions offer another, more fine-grained level of parallelism. We therefore propose to introduce one thread per DoF on each element, and let threads cooperate in computing each tensor contraction. With this approach, the DoF values and gradients are shared between all the threads in a block, and thus the necessary memory per thread is reduced considerably.

Table 1: Number of cells per CUDA block for different elements Q_p in 2D and 3D

	Q_1	Q_2	Q_3	Q_4
2D	32	8	4	4
3D	8	2	1	1

For low-order elements which have few DoFs per cell, a single cell will constitute a very small CUDA block with too few threads for favorable occupancy. In this case, we pack several elements into one block (see Table 1).

When the contributions from different elements should be assembled into the final result vector, many DoFs will be updated by several threads. To avoid race conditions we use a graph coloring approach, where only sets of elements without shared DoFs are processed in parallel. We use the graph coloring algorithm readily available in `deal.II`, which is described in Appendix A of Turcksin, Kronbichler, and Bangerth (2016).

3 TREATMENT OF HANGING NODES

When using a triangular or tetrahedral mesh, adaptive refinement is easily achieved by subdividing the elements flagged for refinement, and also a small number of surrounding elements in order to keep the mesh conforming. When using quadrilateral and hexahedral elements, this cannot be done as straightforwardly, as subdividing these have a greater impact on the surrounding elements. Instead, for such meshes, a non-conforming refinement strategy is often used, which allows for hanging nodes, i.e. nodes on an edge which only belong to one of the elements sharing the edge. In order to enforce the standard smoothness properties of the solution, the unknowns located on these nodes are not actually independent, but must fulfill some linear constraint coupling it to other unknowns. In this paper, we assume the common case that neighboring cells differ in refinement level by at most one, which guarantees that a hanging node is only coupled to

non-hanging nodes. See Section 3.3 in Bangerth, Burstedde, Heister, and Kronbichler (2011) for more details on hanging-node constraints.

For matrix-based methods, it is conceptually simple to first perform the assembly as usual, and then eliminate rows and columns of constrained DoFs from the system. However, it is more efficient to eliminate these on the fly during the assembly itself (see Section 3.3.1 in Bangerth, Burstedde, Heister, and Kronbichler 2011).

With the matrix-free approach, one cannot eliminate the constrained unknowns from the mesh since the sum-factorization evaluation requires each element to have a full set of DoFs. Instead, these constraints must be applied every time the affected unknowns are accessed. In `deal.II`, this is done using a list of local constrained DoFs, which during the access are not simply read from a single global DoF, but computed according to the constraint. This approach has been used successfully for multicore CPUs (Kronbichler and Kormann 2012).

On GPUs, processing the constraints one by one in succession is not compatible with our parallelization with one thread per DoF in each element, since its low parallelism would introduce a serial or almost serial section in the otherwise highly parallel algorithm. Instead, we propose a method which exploits specific properties of hanging-node constraints on tensor-product elements, allowing the threads within an element to cooperate in resolving all constraints at once.

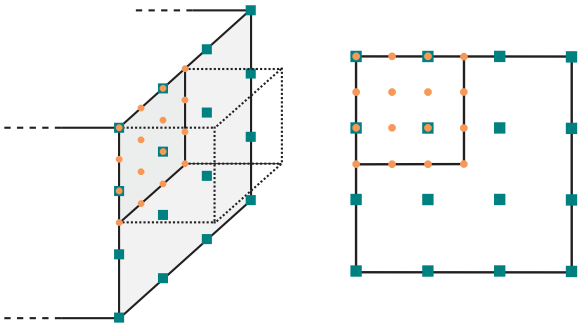


Figure 1: Hanging nodes on a face in 3D, with an extracted 2D view of the face

In Figure 1, we see a typical situation in which constrained DoFs arise in a hexahedral mesh. The DoFs on the fine side (orange dots) are all constrained, and the values they must have to maintain continuity are determined by the value of the basis functions on the coarse side (teal squares). Specifically, the values are computed by evaluating the coarse-side function at the fine points. Since only the basis functions associated with the DoFs on the face are non-zero on the face, only those DoFs will have non-zero weights in the constraint.

For the constrained face shown in Figure 1, each constrained DoF u_i will in general be computed as a unique linear combination of the coarse-side DoFs v_j, weighted by the $(p+1)^2$ shape-function values at that location,

$$u_i = \sum_j \varphi_j(\mathbf{x}_i) v_j \,, \tag{12}$$

where the $\mathbf{x}_i = (x_i, y_i)^T$ are the fine-side DoF locations. This is a relatively large, dense matrix-vector product. However, for the tensor-product elements under consideration, we know that the shape functions factorize into a product of one-dimensional shape functions. This, combined with the similar tensor-product structure of the DoF locations, allows us to make the replacements

$$\varphi_j(\mathbf{x}_i) \to \psi_\alpha(x_\mu)\psi_\beta(y_\nu) \tag{13}$$

where multi-index substitutions $i \to (\mu, \nu)$ and $j \to (\alpha, \beta)$ have been made. If we now insert this into (12), we see that just like in the case of the evaluation of basis functions described in Section 2.3, we can split

this operation up into a series of one-dimensional interpolations performed in succession. Using the same multi-indices, the constrained DoFs can be computed as

$$u_{\mu\nu} = \sum_{\alpha} a_{\mu\alpha} \sum_{\beta} b_{\nu\beta} v_{\alpha\beta} \tag{14}$$

where $a_{\mu\alpha} = \psi_\alpha(x_\mu)$ and $b_{\nu\beta} = \psi_\beta(y_\nu)$. This two-step interpolation approach is illustrated in Figure 2.

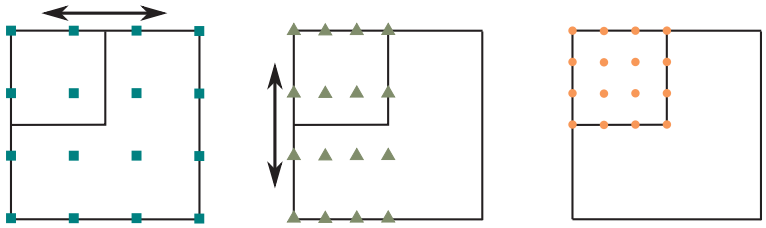


Figure 2: Our method for resolving hanging nodes in a sum-factorization manner; we first interpolate along one coordinate direction, and then the other one.

This approach for the resolution of hanging-node constraints lends itself well to the same fine-grained parallelization as we are using for the basis function evaluation, albeit now restricted to only the DoFs on the face. While this leads to a quite a large number of idle threads, it is still better than the fully serial approach used previously.

In addition to the presently described case of a single constrained face, to which the overwhelming majority of constraints belong, we also include treatment of the exceptional cases where (i) several faces of one element are constrained, and where (ii) only DoFs along an edge of an element are constrained. We do this extending our two-step method to a general three-step process where each step interpolates in one of the coordinate directions, thereby including all applicable constraints on any side or edge at once. In our implementation, we use a 9-bit mask for each element to encode the presence and type of constraints in an efficient and compact manner.

4 DATA STRUCTURES

To optimize the utilization of the GPU memory system, it is very important to choose a data layout which achieves maximum coalescing of memory accesses (NVIDIA Corporation 2016). The memory used in the algorithm can be divided into three types: *per-quadrature data* which is unique to each quadrature point of each element, such as local-to-global DoF index mappings or coefficients; *per-DoF data* such as input and output solution vectors; and *element-independent data*.

With our parallelization with one thread per local DoF, optimal coalescing is achieved if we use an array-of-structure data layout for the *per-quadrature data*, rather than a structure-of-array format which is usually preferred for GPUs. The inverse Jacobian, must be treated in a slightly altered way, since it contains D^2 entries for each quadrature point that are read in succession by the same thread. In this case, an "array-of-structure-of-array" approach is appropriate where for each element we first store the J_{00} components for all quadrature points, then all J_{01} components, etc. In addition, we make sure that memory is properly aligned by padding each chunk of memory to 128 byte boundaries.

For a general mesh, the *per-DoF* solution vectors will be read in a very irregular manner. The fact that many DoFs are shared between elements leads to a conflict of interests where different elements have different opinions on what would be a good ordering of the DoFs, essentially making it impossible to lay out data in a way to coalesce all access. This becomes even more severe for a general unstructured mesh with hanging nodes. The situation becomes somewhat better with higher element order since that increases the portion

of non-shared DoFs. However, the solution vectors amount to much less data than the per-quadrature data, namely $2(pn+1)^d / \left((d^2+2)(p+1)^d n^d\right)$ for a d-dimensional mesh with n elements of order p in each dimension, which is roughly 2 - 20% for the elements under consideration. Therefore, we think this issue is not significant in practice.

For element-independent data, such as the values and gradients of one-dimensional shape functions at quadrature points, we first tried using constant memory to allow for caching in the L1 read-only cache. However, it turned out to be more favorable to explicitly stage relevant parts of these in registers.

5 BENCHMARK EXPERIMENTS

We evaluate the performance of our parallelization by running a number of numerical benchmark experiments. In these, we measure the time to compute a multiplication using our matrix-free method by applying the operation 100 times and computing the average time. We do not include time needed to set up the data structures and the time for transferring data from the CPU to the GPU, as in an iterative solver, all the data would reside on the GPU throughout the whole computation. We are using double-precision numbers throughout, since this is necessary for proper convergence of iterative linear solvers.

The benchmarks are based on the same Poisson problem considered earlier, with a 2D or 3D hyper ball domain, homogeneous Dirichlet conditions, and a variable coefficient defined by $A(\mathbf{x}) = 1/\left(0.05+2\|\mathbf{x}\|^2\right)$. This geometry ensures a mesh with non-Cartesian elements, making it general enough to be representable for more complicated geometries. The relatively coarse base mesh is refined uniformly to create a series of successively finer meshes, which lets us study how performance scales with problem size. To see the performance impact of our treatment of hanging-node constraints, we also run the benchmark on a series of meshes with adaptive refinement. Specifically, the coarsest mesh is refined on a series of non-aligned spherical shells, which simulates adaptive refinement along some wave fronts. An example of such a mesh can be seen in Figure 3. Table 2 lists the largest meshes used for each element configuration.

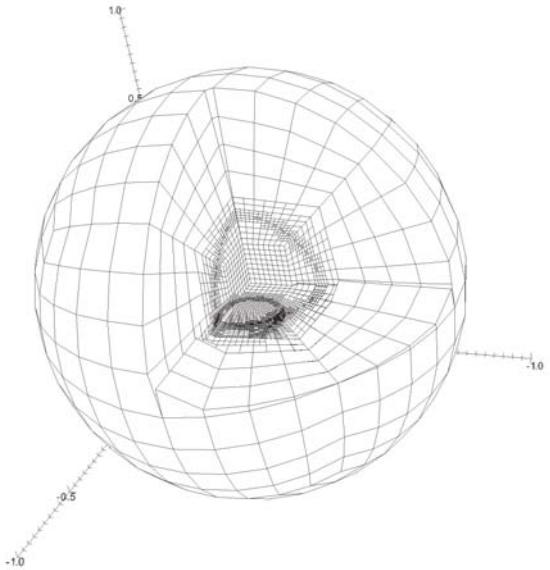


Figure 3: The adaptively refined mesh used to benchmark our method for resolving hanging-node constraints.

We compare our GPU implementation of the matrix-free method with the highly optimized CPU implementation of same matrix-free method by Kronbichler and Kormann (2012), which is readily available in `deal.II`. We also compare against a sparse-matrix-based version for GPU using Nvidias official sparse-matrix library

Table 2: Meshes used in the experiments

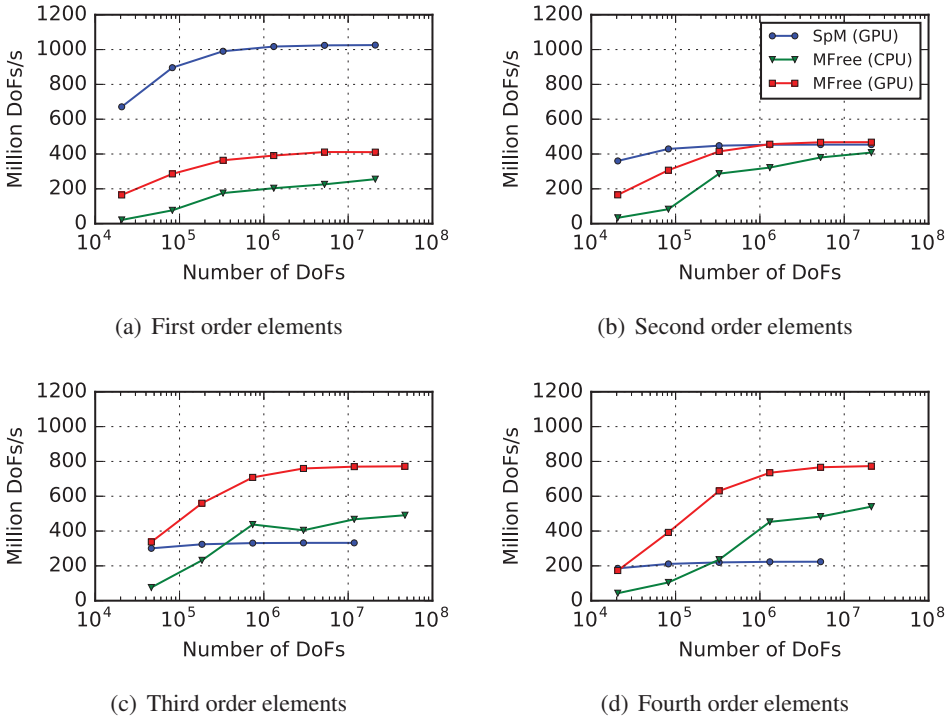
(a) Uniform refinement

d	p	# cells	# DoFs
2	1	20971520	20975617
2	2	5242880	20975617
2	3	5242880	47192065
2	4	1310720	20975617
3	1	1835008	1847617
3	2	1835008	14729857
3	3	229376	6221281
3	4	229376	14729857

(b) Adaptive refinement

d	p	# cells	# DoFs	% HN
2	1	18271217	18572584	3.23
2	2	4847393	19992109	4.51
2	3	4847393	44680720	3.37
2	4	1350302	22356367	4.70
3	1	1223845	1522469	38.3
3	2	1223845	11564225	25.4
3	3	180964	5784259	24.8
3	4	180964	13205632	19.6

CUSPARSE (NVIDIA Corporation 2013). The GPU experiments were run on a server with an Nvidia Tesla K40, an Intel Core i5-3550 quad-core, 16 GB RAM and CUDA 8 RC. The CPU experiments were run on a system with two Intel Xeon E5-2680 eight-core processors, and 64 GB RAM. The K40 GPU has a TDP of 235 W whereas the two Xeon processors has a combined TDP of 260W.



(a) First order elements

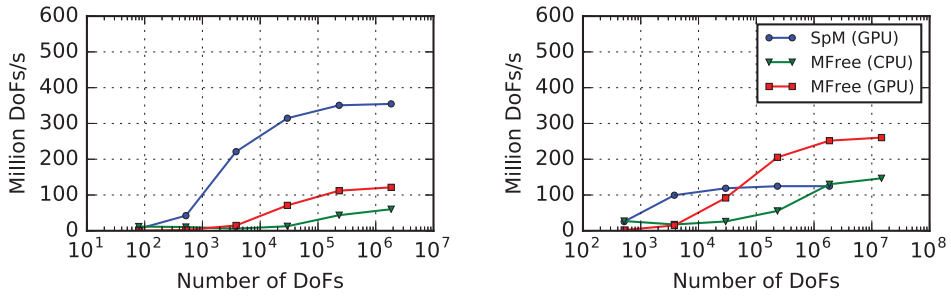
(b) Second order elements

(c) Third order elements

(d) Fourth order elements

Figure 4: Throughput vs problem size for the 2D uniformly refined mesh.

In Figure 4 and 5, the results for the uniformly refined mesh are shown. To facilitate comparison across different problem sizes and implementations with different memory and compute patterns, we present performance as throughput in terms of DoFs processed per second, rather than flop/s or bandwidth. Firstly, we see that, as expected from theory, the matrix-free method is slower than the SpMV version for Q_1 elements, and faster for second order elements and higher. In 2D, the performance gain is very small for second order elements ($\approx 3\%$), but from third order we see speedups of 2.3 - 3.4$\times$. In 3D, we get substantial speedups of 2 - 8.2$\times$ already from second order elements. Secondly, our implementation is consistently faster than the CPU version; between 15% and 61% in 2D, and between 1.8$\times$ and 2.3$\times$ in 3D. Finally, we note that when using a matrix, we get problems fitting it in memory for elements of order 3 and higher in 2D as indicated by



(a) First order elements

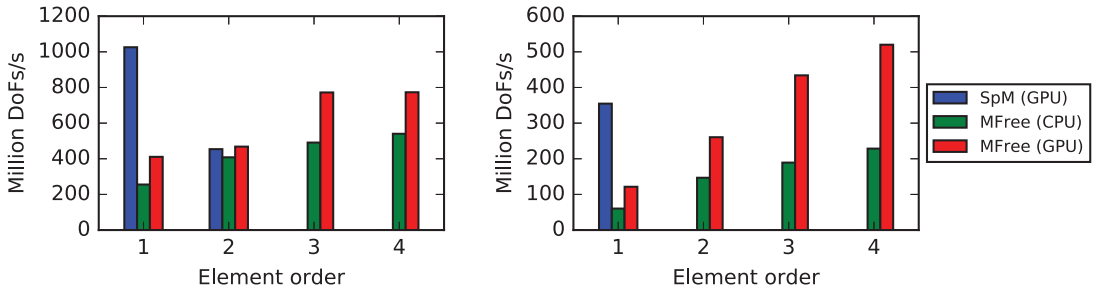
(b) Second order elements

(c) Third order elements

(d) Fourth order elements

Figure 5: Throughput vs problem size for the 3D uniformly refined mesh.

the truncated lines for SpM. In 3D, this happened already from second order elements. In Figure 6, we have summarized the results for the largest problems considered.

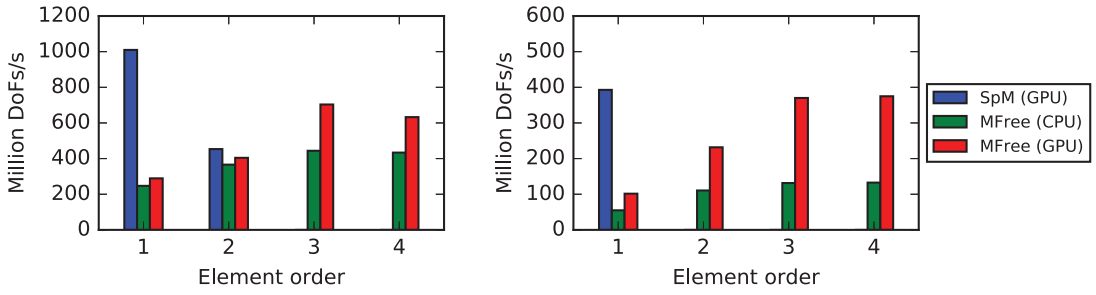


(a) 2D

(b) 3D

Figure 6: Performance for the largest problems solved (uniform refinement). The missing bars for SpM (GPU) indicate that the matrix did not fit in memory.

If we compare Figure 6 with Figure 7, which shows the corresponding results for the adaptively refined mesh, we see that there is a moderate overhead from resolving the hanging nodes. Specifically, the overhead is about 10 - 40% for our GPU implementation, which is lower than the overhead of the CPU version which reaches 70% for some meshes. The matrix-based version eliminates the constrained DoFs once and for all during the assembly, and thus does not see any noticeable overhead. Still, the much better efficiency of our method makes it continue outperforming the matrix-based one from element order 3 in 2D (2 - 2.6×), and from element order 2 in 3D (34% - 4.6×).



(a) 2D (b) 3D

Figure 7: Performance for the largest problems solved on the adaptively refined ball.

Finally, we add that for Q_4 in 3D the matrix-free version achieves 35% bandwidth utilization out of the specified 288 GB/s, compared to 52% when using SpMV. Considering that 13× less memory is accessed, this is a notably small reduction in bandwidth utilization. Still, there is clearly room for further improvements.

6 CONCLUSIONS

We have developed a framework for matrix-free finite-element computations on graphics processors, supporting adaptively refined meshes with hanging nodes. As expected, our method is faster than highly a matrix-based version from elements of order two and higher, reaching speedups of up to 8× over the highly optimized CUSPARSE library. Compared to a state-of-the-art CPU implementation of the same matrix-free algorithm, our implementation for GPUs is 15% - 61% faster in 2D, and 1.8× - 2.3× faster in 3D. Comparing highly optimized implementations of the same algorithm on GPUs and CPUs of similar power consumption, this suggests that GPUs are about 2 times more power efficient than CPUs for this kind of computations in 3D. While our algorithm for resolving hanging node constraints on the GPU introduces some overhead, this overhead is relatively low, both compared to the overhead of the matrix-free CPU implementation, and in the sense that we still outcompete the CUSPARSE version for elements of order 3 and higher in 2D, and for elements 2 and higher in 3D. Finally, we note that the matrix-free method has the additional benefit of eliminating the matrix assembly, and allowing for solution of problems at least 4 times larger in 2D and 8 times larger in 3D, on a given GPU.

Ongoing work includes implementation of a multigrid linear solver to enable competitive full PDE solution on GPUs. Also, we are working on further generalization to systems of equations allowing for simulation of more general applications, such as fluid flow and structural mechanics. Finally, there is an ongoing effort towards an official inclusion of the GPU implementation in `deal.II`.

ACKNOWLEDGMENTS

The author thanks M. Kronbichler, TU Munich, for valuable discussions and advice. All experiments have been executed on hardware provided by the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

REFERENCES

Bangerth, W., C. Burstedde, T. Heister, and M. Kronbichler. 2011. “Algorithms and data structures for massively parallel generic adaptive finite element codes”. *ACM Trans. Math. Softw.* vol. 38, pp. 14/1–28.

Bangerth, W., D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. 2016. "The `deal.II` Library, Version 8.4". *Journal of Numerical Mathematics* vol. 24.

Brenner, S. C., and L. R. Scott. 2002. *The mathematical theory of finite element methods*. Springer.

Brown, J. 2010. "Efficient Nonlinear Solvers for Nodal High-Order Finite Elements in 3D". *Journal of Scientific Computing* vol. 45 (1), pp. 48–63.

Cantwell, C. D., S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly. 2011. "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements". *Computers & Fluids* vol. 43 (1, SI), pp. 23–28.

Carey, G. F., and B.-N. Jiang. 1986. "Element-by-element linear and nonlinear solution schemes". *Communications in Applied Numerical Methods* vol. 2 (2), pp. 145–153.

Cecka, C., A. J. Lew, and E. Darve. 2011. "Assembly of finite element methods on graphics processors". *International Journal for Numerical Methods in Engineering* vol. 85, pp. 640–669.

Dehnavi, M. M., D. M. Fernandez, and D. Giannacopoulos. 2010, AUG. "Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units". *IEEE Transactions on Magnetics* vol. 46 (8), pp. 2982–2985. 17th International Conference on the Computation of Electromagnetic Fields (COMPUMAG 09), Santa Catarina, Brazil, Nov 22-26, 2009.

Göddeke, D., R. Strzodka, and S. Turek. 2005, Sept. "Accelerating Double Precision FEM Simulations with GPUs". In *Proceedings of ASIM 2005 – 18th Symposium on Simulation Technique*, pp. 139–144.

Gropp, W. D., D. K. Kaushik, D. E. Keyes, and B. F. Smith. 1999. "Towards Realistic Performance Bounds for Implicit CFD codes". In *Proceedings of Parallel CFD'99*, Elsevier.

Klöckner, A., T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. "Nodal discontinuous Galerkin methods on graphics processors". *Journal of Computational Physics* vol. 228 (21), pp. 7863–7882.

Komatitsch, D., G. Erlebacher, D. Göddeke, and D. Michéa. 2010. "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster". *Journal of Computatinal Physics* vol. 229 (20), pp. 7692–7714.

Kormann, K., and M. Kronbichler. 2011. "Parallel Finite Element Operator Application: Graph Partitioning and Coloring". In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pp. 332–339.

Kronbichler, M., and K. Kormann. 2012. "A Generic Interface for Parallel Cell-Based Finite Element Operator Application". *Computers & Fluids* vol. 63 (0), pp. 135–147.

Ljungkvist, K. 2014. "Matrix-Free Finite-Element Operator Application on Graphics Processing Units". In *Euro-Par 2014: Parallel Processing Workshops*, Volume 8806 of *Lecture Notes in Computer Science*, pp. 450–461. Springer International Publishing.

Markall, G. R., A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. 2013, JAN 10. "Finite Element Assembly Strategies on Multi-Core and Many-Core Architectures". *International Journal for Numerical Methods in Fluids* vol. 71 (1), pp. 80–97.

NVIDIA Corporation 2013, July. *CUDA CUSPARSE Library*.

NVIDIA Corporation 2016, September. *NVIDIA CUDA C Programming Guide*. Version 8.0.

Turcksin, B., M. Kronbichler, and W. Bangerth. 2016, August. "*WorkStream* – A Design Pattern for Multicore-Enabled Finite Element Computations". *ACM Trans. Math. Softw.* vol. 43 (1), pp. 2:1–2:29.

AUTHOR BIOGRAPHY

KARL LJUNGKVIST is a PhD Student at Uppsala University. His research interests lie in high-performance computing and development of scientific software. His email address is karl.ljungkvist@it.uu.se.