Matrix-Free Finite-Element Operator Application on Graphics Processing Units

Karl Ljungkvist

Department of Information Technology, Uppsala University, Sweden
karl.ljungkvist@it.uu.se

Abstract. In this paper, methods for efficient utilization of modern accelerator-based hardware for performing high-order finite-element computations are studied. We have implemented several versions of a matrix-free finite-element stiffness operator targeting graphics processors. Two different techniques for handling the issue of conflicting updates are investigated; one approach based on CUDA atomics, and a more advanced approach using mesh coloring. These are contrasted to a number of matrix-free CPU-based implementations. A comparison to standard matrix-based implementations for CPU and GPU is also made. The performance of the different approaches are evaluated through a series of benchmarks corresponding to a Poisson model problem. Depending on dimensionality and polynomial order, the best GPU-based implementations performed between four and ten times faster than the fastest CPU-based implementation.

1 Introduction

For applications where the geometry can be expected to be very complicated, methods based on completely unstructured grids, such as finite-element methods, are popular because of their ability to fully capture the geometry. On the other hand, in application fields where solutions also posses a high level of smoothness, such as in micro-scale simulation of viscous fluid, or linear wave propagation in an elastic medium, using a high-order numerical method can give high accuracy and efficiency. However, computational challenges limit the numerical order of a conventional matrix-based finite element-method.

Traditionally, the finite element method, *FEM*, has been seen as consisting of two distinct parts; an assembly of a linear system of equations, and a solution of this system. The system of equations is then typically represented as a sparse matrix, and the solution is found using an iterative Krylov subspace method. However, if high-order basis functions are used, in particular in 3D, the system matrix becomes increasingly less sparse. In order to accurately simulate realistic problems in three dimensions, millions or even billions of degrees of freedom can be required. In such cases, the system matrix can simply be too large to store explicitly in memory, even if a sparse representation is used.

In addition to the problem of storage, an equally important problem is that of memory bandwidth. In most iterative methods, most time is typically spent

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 450–461, 2014.

performing sparse matrix-vector products, *SpMV*, with the system matrix [1]. The sparse matrix-vector product has a relatively poor ratio of computations per memory access. On modern computer systems, even the most optimized implementations of this operation will not utilize the computation resources fully and is effectively bound by the memory bandwidth [2].

Matrix-free finite-element methods avoid these issues by merging the assembly and SpMV phases into a single operator application step, thereby removing the need for storing the system matrix explicitly. Since the large system matrix no longer has to be read, the bandwidth footprint is reduced radically. On the other hand, this is traded for additional computations, since the assembly needs to be recomputed at each operator application. For non-linear and time-dependent problems, this is not an issue since reassembly is necessary anyway. In [3], Cantwell et al. perform a comparison of different matrix-based and matrix-free approaches to high-order FEM, concluding that for order one elements, sparse matrices are most efficient, while for orders two and higher, a matrix-free approach yields the best performance. In [4], Kronbichler and Kormann propose a general framework for matrix-free finite element methods.

Due to the increased computational intensity of the matrix-free approach [4], it makes a good candidate for execution on throughput-oriented hardware such as graphics processors. Work on porting high-order FEM code to GPUs include the work by Cecka et al. [5], which compares different methods for performing the assembly of an explicit FEM matrix on GPUs. In [6], Klöckner et al. proposed a GPU implementation of a Discontinuous Galerkin method, which in many ways is similar to finite-element methods. However, there hyperbolic conservation laws were studied, which allows for an explicit time stepping without the need to solve a linear system. In [7], Komatitsch et al. port an earthquake code based on the related spectral element method to GPUs. Also here, the seismic wave equation being studied is hyperbolic and can be integrated explicitly in time.

In this paper, we propose a matrix-free GPU implementation of a finite-element stiffness operator based on CUDA, for future use in a solver for possibly non-linear elliptic and parabolic PDEs. An issue in performing the operator application is how to avoid race conditions when writing partial results to the output. We present two different techniques to handle this; one which uses the intrinsic atomic instruction of CUDA to protect the writes, and a more advanced technique based on mesh coloring to avoid the conflicts. We evaluate the two techniques in benchmarks based on a simple model problem, namely Poisson's equation on a Cartesian mesh in 2D and 3D, for polynomial degrees one to four.

2 A Matrix-Free Finite-Element Method

In the following discussion, the Poisson equation with homogeneous boundary conditions,

$$\nabla^2 u = f \text{ on } \Omega, \tag{1}$$

$$u = 0 \text{ on } \partial\Omega, \tag{2}$$

in two dimensions is studied. This is a simple model problem, however it is still representative of more complex problems as it shares most of their properties. If the equation involves other differential operators than ∇^2, they are typically treated in a similar way. It is readily extensible to three or higher dimensions. If there is a time dependency, a similar time-independent equation is solved at each time step. If the equation is non-linear, it is linearized and a similar linear problem is solved, e.g. throughout a Newton iteration procedure. Non-homogeneous Dirichlet boundary conditions can easily be transformed to homogeneous ones, and the treatment of Neumann conditions or more general Robin conditions leads to similar end results.

By multiplying (1) by a test function v and integrating by parts, the weak form

$$\int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}V = \int_\Omega v f \,\mathrm{d}V \tag{3}$$

is obtained, where v belongs to the function space V which is chosen to satisfy the boundary conditions (2).

Now, let $\mathcal{K}$ be a quadrilateralization of Ω, i.e. a partitioning of Ω into a set of non-overlapping quadrilaterals Ω_k. Also, let V_h be the finite-dimensional space of all functions v, bi-polynomial of degree p within each element Ω_k, continuous between neighboring elements, and, once again, fulfilling the boundary condition. To find a basis for V_h, we begin by noting that in order to span the space of all p'th order bi-polynomials of an element, $(p+1)^2$ basis functions are needed for that element. To uniquely determine the coefficients of these $(p+1)^2$ element-local basis functions, $(p+1)^2$ *degrees of freedom*, (*DoFs*) are needed, which are introduced as the function values at $(p+1)^2$ node points on each element. Note that node points on edges and corners will be shared between several elements. The basis is then comprised of the p'th-degree bi-polynomials $\{\psi_i\}_{i=1}^{N_p}$, where basis function ψ_i is equal to unity at precisely node $j = i$, and zero at all other nodes $j \neq i$.

Expanding the solution in this space, $u = \sum_{i=1}^{N} u_i \psi_i$, and substituting ψ_j as the test functions v, we get

$$\sum_{i=1}^{N} A_{i,j} u_i = b_j \,, \text{for } j = 1, \ldots, N \,, \tag{4}$$

where

$$A_{i,j} = \int_\Omega \nabla \psi_i \cdot \nabla \psi_j \mathrm{d}V \tag{5}$$

$$b_j = \int_\Omega f \psi_j \mathrm{d}V \,. \tag{6}$$

This is a linear system in the DoFs u_i, which needs to be solved in order to obtain the approximate solution u to the original problem (1).

Noting that (5) can be writen as a sum over the elements in the mesh $\mathcal{K}$,

$$A_{i,j} = \sum_{k \in \mathcal{K}} \int_{\Omega_k} \nabla \psi_i \cdot \nabla \psi_j \mathrm{d}V \,, \tag{7}$$

we observe that each sub-integral will only be non-zero for very few combinations of basis functions, namely the ones that have a non-zero overlap on element k. If we introduce a local numbering of the DoFs within an element, there will be an element-dependent mapping I^k translating local index j to global index $I^k(j)$, and an associated permutation matrix ${P^k}_{i,j} = \delta_{i,I^k(j)}$. Using this, and introducing ψ_l^k as the l'th basis function on element k, we can write (7) on matrix form as

$$A = \sum_{k \in \mathcal{K}} P^k A^k {P^k}^T \,, \tag{8}$$

where the local stiffness matrix A^k is defined as

$$A^k_{l,m} = \int_{\Omega_k} \nabla \psi^k_l \cdot \nabla \psi^k_m \mathrm{d}V \,. \tag{9}$$

2.1 Computation of the Local Matrix

The integral in (9) is usually computed by transforming Ω_k to a reference element, and using numerical quadrature. Typically, Gaussian quadrature is used since polynomials can be integrated exactly.

$$A^k_{i,j} = \sum_q \left[J_k^{-1}(\hat{x}_q) \hat{\nabla} \hat{\psi}_i(\hat{x}_q) \right] \cdot \left[J_k^{-1}(\hat{x}_q) \hat{\nabla} \hat{\psi}_j(\hat{x}_q) \right] |\det J_k(\hat{x}_q)| w_q \,,$$

where J_k is the Jacobian matrix of the transformation from reference element to the k'th real element, $\hat{x}_q$ are the quadrature points of the reference element, and w_q are the quadrature weights.

Now, if the mesh is uniform, i.e. all elements have the same shape and size, J_k will be the same for all k. In this case, also A^k will be independent of k, and a single $\hat{A}$ can be precomputed and stored in memory. For a non-uniform mesh, however, all the A^k will be distinct and a precomputation is unfeasible due to the extensive storage requirement. In such a case, a tensor based approach can be used, as described by Kronbichler and Kormann [4].

2.2 Matrix Free Operator Application

In the case of standard finite-element methods where an explicit matrix is used, (8) is computed once and the resulting matrix is stored, to be used in the subsequent multiplications. To obtain the matrix-free case, we multiply (8) by the vector u and simply rewrite it the following way,

$$Au = \left(\sum_{k \in \mathcal{K}} P^k A^k {P^k}^T \right) u \Leftrightarrow Au = \sum_{k \in \mathcal{K}} \left(P^k A^k {P^k}^T u \right) . \tag{10}$$

Since the permutation matrices merely selects and reorders rows, we have essentially disassembled the operator application from a sparse matrix-vector multiplication into a sum of many, small and dense matrix-vector multiplications, where each such multiplication involves a computation of the local matrix A^k.

2.3 Parallelization

Being made up of many small, independent matrix-vector products and the associated local-matrix computations, the matrix-free operator application in (10) is almost trivially parallelized – the list of elements is simply split into chunks of appropriate size and then all the chunks are processed in parallel. However, a problem arises when assembling the results into the single output vector.

For a given row i of the result, most of the terms in the sum in the right-hand side of (10) will be zero, however, the terms corresponding to all elements to which the i'th DoF belongs will be non-zero. All of these contributions will need to be added to the single memory location at row i of the result. Since these are computed in parallel, care must be taken to avoid race conditions while updating the shared memory location.

Mesh Coloring. As previously stated, only the elements to which a given node i belongs will give a contribution to the i'th row of the result. Conversely, this means that any two elements which do not share a DoF will be free of any conflicting updates, and may thus be processed concurrently.

One way of achieving this, is to use graph coloring. Denote two elements in a mesh as *neighbors* if they do not share any node points, which will hold if they do not share any vertices (see Fig. 1). Then, if all elements in the mesh are colored such that within each color, no two elements are neighbors, then all the elements within a single color can safely be executed in parallel.

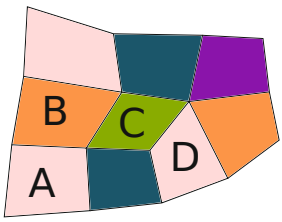


Fig. 1. Elements A and B are neighbors, as are elements A and C, and are thus given different colors. Elements A and D are not neighbors and can be given the same color.

Since not all elements are processed in parallel, there is a reduction of parallelism of $\frac{1}{N_c}$, where N_c is the number of colors needed. For a logically Cartesian mesh, $N_c = 2^d$, where d is the dimensionality of the problem, whereas for an

unstructured FEM mesh, $N_c > 2^d$ in general (see Fig. 1). In both cases, however, N_c will be independent of the number of elements of the mesh. Thus, for sufficiently large problems, the overhead will be small enough. For the uniform meshes considered in this paper, the coloring is trivial. For the case of a general mesh, a more advanced graph coloring algorithm must be used, such as the ones of Berger et al. [8], Farhat and Crivelli [9], or Komatitsch et al. [7].

3 Graphics Processors

Recently, graphics processing units (GPUs) have seen an increasing use as general-purpose processors for high-performance computations within science and technology. Computer graphics consists of processing a large number of independent polygon vertices. Tailored for this very parallel and compute-intensive task, the architecture of GPUs is optimized for high throughput rather than low latency, which is the case for CPUs. Because of this, a much larger area of the GPU chip is dedicated to computations compared to a CPU. Also, memory bandwidth is typically considerably higher than on a CPU, whereas the caching system of a CPU aims at achieving low latency. As a consequence of the higher computing power per transistor, GPUs achieve a much higher efficiency, both economically (i.e. Gflops/$) and power-wise (i.e. Gflops/W).

Being comprised of many small similar tasks with a high computational intensity, scientific applications, such as e.g. stencil operations or linear algebra, have in many cases been well suited for the throughput-optimized GPU hardware. However, few applications fit the graphics-tailored GPU architecture perfectly and in practice, issues like the limited support for double precision or the necessity for very high parallelism may limit the utilization of a GPU system.

The first attempts at utilizing commodity graphics hardware for general computations were based on exploiting the programmable vertex and pixel shaders of the graphics pipeline. For a summary of the early endeavors in GPGPU, see the excellent survey by Owens et al. [10]. However, programming the graphics pipeline was difficult, and the real revolution came at the end of 2006, when Nvidia released CUDA, *Compute Unified Device Architecture*. The CUDA platform provides a unified model of the underlying hardware together with a C-based programming environment. The CUDA GPU, or *device*, comprises a number of Streaming Multiprocessors (SMs) which in turn are highly parallel multicore processors. The threads of the application are then grouped into thread blocks which are executed independently on a single SM. Within a thread block or an SM, there is a piece of shared memory, and a small cache. Finally, synchronization is limited and only possible between threads within a block, except for global barriers. For further details on the CUDA platform, see the CUDA C Programming Guide [11]. Examples of studies based on CUDA include molecular dynamics simulations [12], fluid dynamics [13] and wave propagation [14].

Although CUDA is vendor specific and GPUs have a very specialized architecture, they are both part of a larger movement – that of heterogeneity and increasing use of specialized hardware and accelerators. Thus, developing

algorithms and techniques for dedicated accelerators, such as GPUs, is relevant also for the technology of the future.

4 Experiment Code

As part of this research, a small framework for high-order finite-element application in efficient, heavily templated C++/CUDA has been developed. Because of the high accuracy which is needed when solving scientific problems, double precision is used throughout the code. The mesh is stored in an array of points and an array of elements. For the elements, an element struct is used comprising a list of DoF indices. This array-of-structure format was found to perform better than a structure-of-array approach, both for the CPU and the GPU.

We have implemented several different versions of the stiffness-matrix operator. Apart from the matrix-free GPU implementations, we include serial and parallel matrix-free implementations for the CPU, as well as matrix-based implementations for both CPU and GPU, for comparison.

4.1 Matrix-Based Implementations

The matrix-based reference implementation for the CPU, `SpM`, uses a Compressed Sparse Row (CSR) matrix format, since this performs well during matrix-vector multiplication. For the assembly, a list-of-lists (LIL) format is used, since this has superior performance during incremental construction. After the construction, the LIL matrix is converted to the CSR format, without much overhead. Still, the matrix construction amounts to a significant part of the total execution time (see results under Sect. 5.1). The sparse matrix-vector product is parallelized in OpenMP, by dividing the rows in chunks evenly over the processors. We used four threads, since this gave the best performance.

The corresponding implementation for the GPU, `GPU_SpM`, uses the efficient SpMV kernel of CUSPARSE, a sparse matrix library released by Nvidia as part of CUDA. The matrix assembly is performed on the CPU identically to the `SpM` implementation, and then copied to the GPU.

4.2 Matrix-Free Implementations

Our matrix-free implementations follows the idea described in Sect. 2.2. Since a uniform mesh is assumed, the local matrix is the same for all elements and a single copy is precomputed and stored. The serial version is called `Mfree`.

There are two versions parallelized using OpenMP, both based on computing the contribution from multiple elements in parallel. The main difference between the versions is the technique used to solve the conflict issue described in Sect. 2.3. In the `PrivateBuffers` implementation, each OpenMP thread writes its result to its own version of the output vector. After all threads have finished computing, a parallel reduction phase sums up the buffers into a single vector, trading

off the conflicts for the extra storage and computations. Finally, there is an implementation `Color` which uses the mesh coloring method described in Sect. 2.3 to avoid the conflicts. Once again, four threads are used since this gave the best speedup relative to the serial version.

Much like the matrix-free implementations for the CPU, the ones for the GPU mainly differ in the treatment of conflicts. In all implementations, each thread handles a single element. A block size of 256 threads was chosen since this performed best in the experiments. There is one version, `GPU_Atomic`, which uses the built-in atomic operations of CUDA to protect the conflicting writes. There is also an implementation `GPU_Color` using the more advanced coloring-based treatment of conflicts described in Sect. 2.3. Finally, a version without any protection, `GPU_Max`, is also included to get an upper bound on the performance for an element-wise parallelization of the matrix-free operator application.

5 Numerical Experiments

The performance of the different implementations described above are evaluated through a series of benchmark experiments. These are based on the Poisson problem studied in Sect. 2. The unit square domain is discretized by a Cartesian mesh of quadrilateral elements of order p. A similar problem in 3D is considered, i.e. a unit cube discretized by a Cartesian mesh of p'th-order hexahedral elements. In detail, the experiment consists of the following parts:

1. Setup of data structures for the mesh, the vectors, and the operator.
2. Transfer of data to the appropriate memory location (i.e. device memory for GPU-based implementations).
3. 20 successive applications of the operator.
4. Transfer of data back to main memory.

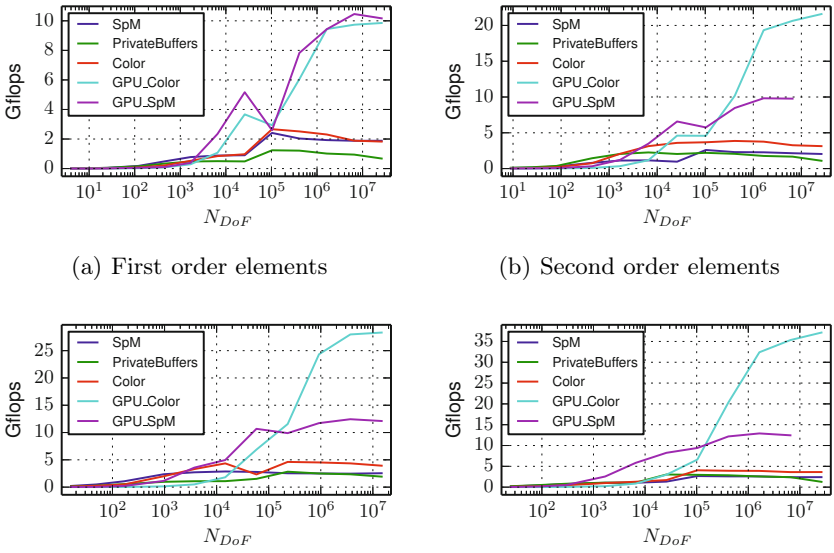
To evaluate the execution time for the operator application, the time for steps 2–4 is measured, and the time for a single application is calculated by dividing by the number of iterations, i.e. 20. Furthermore, to get more stable results, 20 repetitions of steps 2–4 are performed, and the minimum time is recorded. The experiment is run for all the operator implementations described in Sect. 4, with polynomial degrees of one to four.

All experiments are performed on a server with an Intel Xeon E5-2680 eight-core processor @ 2.70GHz, 64 GB DRAM and an Nvidia Tesla K20c GPU with 2496 cores and 5 GB of ECC-enabled memory. The test system runs Linux 2.6.32, with a GCC compiler of version 4.4, and a CUDA platform of version 5.5.

5.1 Results

Figures 2 and 3 depict the performance of the most important implementations as a function of the number of degrees of freedom, in 2D and 3D respectively.

Firstly, we see that performance increases with the problem size as the parallelism of the hardware is saturated, in particular for the versions for the GPU,



(a) First order elements

(b) Second order elements

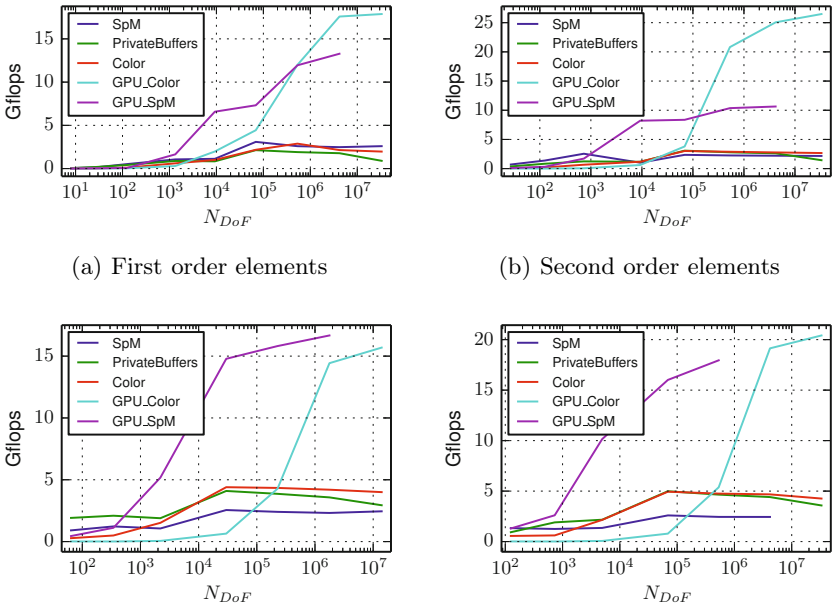
(c) Third order elements

(d) Fourth order elements

Fig. 2. Scaling of the performance with the problem size (N_{DoF}), for the 2D experiments

due to its much higher parallelism. Also, it is evident that the GPU versions performed significantly faster than the ones for the CPU. Furthermore, we see that, as the complexity of the elements increases, i.e. as polynomial degree and dimensionality grow, so does the benefit of using a matrix-free approach. Although the matrix-based implementations for CPU and GPU performed on par with the matrix-free ones for element order one, they are outperformed already for second order elements. Moreover, in many cases, as expected, it was simply impossible to use the matrix-based version, since the storage requirement for the matrix exceeded the system memory (indicated by the truncated curves for `SpM` and `GPU_SpM`). Finally, as predicted, the setup times were reduced considerably. For the example of fourth-order polynomials in 2D, `SpM` required 14 seconds for the setup, whereas `Color` required only 0.2 seconds, a difference that was even larger in 3D. Similar times were recorded for the matrix-based and matrix-free GPU implementations. The performance for the largest problems is presented in more condensed form in Fig. 4 (a) and (b), which display the performance of all implementations at the largest problem size as p varies, for 2D and 3D, respectively.

For the results in 2D (Fig. 4(a)), we begin by noting that the matrix-free GPU versions gave very good speedups over the reference versions (between 5.4 and 10 times versus the fastest CPU version). In fact, the amount of work performed per time by the matrix-free GPU versions grew steadily with the polynomial order, whereas for both the matrix-based GPU implementation and all the CPU imple-



(a) First order elements

(b) Second order elements

(c) Third order elements

(d) Fourth order elements

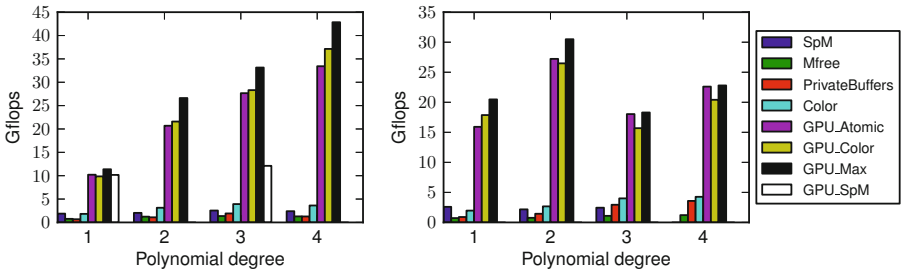
Fig. 3. Scaling of the performance with the problem size (N_{DoF}), for the 3D experiments

mentations, this stayed roughly constant. Comparing the results of `GPU_Color` and `GPU_Atomic` with the result of version without any protection, `GPU_Max`, we see that there is an overhead of dealing with conflicting updates, but that using a coloring approach was more efficient than using atomic intrinsics.

From the results of the CPU-based matrix-free versions, it is clear that the straightforward implementation using private buffers gave a very poor speedup, due to the overhead of performing the buffer reduction. On the other hand, just as in the case of the GPU implementations, the parallelization based on coloring achieved a good speedup of about 3.5.

Looking at the results for the 3D experiment (see Fig. 4(b)), we see that, once again, using a matrix-free method on the GPU can give large speedups (4.5 – 10×). However, although we still see a speedup over the CPU, there is a significant drop in performance when going to order 3 and 4. An explanation for this can be found by looking at the size of the local matrix, $(p+1)^{(2d)} \cdot 8B$, which for $d = 3$ and $p = 3$ exactly matches the size of the L1 cache available per SM, namely 32kB. Thus, the threads within a block can no longer fetch the local matrix collectively by sharing reads.

Finally, we note that the Gflops numbers in Fig. 2 - 4 are fairly low, and quite far from the theoretical 1.17 double precision Tflops of the K20. However, this is no surprise since the SpMV operation is bandwidth-bound, which is also the case



(a) 2D (b) 3D

Fig. 4. Performance for the largest problems solved (with 26.2M, 26.2M, 14.8M, and 26.2M DoFs (2D) ; and 33.1M, 33.1M, 14.0M and 33.1M DoFs (3D), respectively). The missing bars for `SpM` and `GPU_SpM` indicate a fail, i.e. the matrix did not fit in main memory.

for a matrix-free version using a precomputed local matrix. This is confirmed by the numbers for global memory bandwidth utilization reported by `nvprof`, which lie around 110 GB/s, compared to the official peak 208GB/s (reported for ECC off), indicating a fairly well utilized bandwidth.

6 Conclusions

Our GPU implementations of the matrix-free stiffness operator achieved speedups of 4.5 and 10 times relative to the fastest CPU-based implementation. The results indicate that as element complexity grows, i.e. if the dimensionality and element degree increases, so does the performance benefit of using the GPU, which is promising for future use in a high-order finite-element method solver of elliptic and parabolic PDEs. Finally, as indicated by our results for the setup times, applications where frequent reassembly is necessary, such as time-dependent or non-linear problems, can benefit substantially from using a matrix-free approach. In addition, with the matrix-free method, we were able to solve problems an order of magnitude larger than with the matrix-based methods.

We saw that for a too large local matrix, performance drops significantly. However, as was pointed out in Sect. 2.1, the strategy based on a local matrix is limited to uniform meshes, meaning that for more realistic problems, other approaches, such as the tensor based technique of Kronbichler and Kormann [4], are necessary anyway. Considering this, the present result suggests that such methods can be favorable also for uniform meshes due to the lower memory footprint, for which the already good speedups can be expected to improve further.

Topics of ongoing research include development of a tensor-based operator implementation, as well as techniques for reduction of the high bandwidth usage, and solution of realistic problems within the field of two-phase flow simulation.

Acknowledgments. The computations were performed on systems provided by the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

1. Saad, Y.: Iterative Methods for Sparse Linear Systems, vol. 2. Society for Industrial and Applied Mathematics, Philadelphia (2003)
2. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–12 (2007)
3. Cantwell, C.D., Sherwin, S.J., Kirby, R.M., Kelly, P.H.J.: From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements. Computers & Fluids 43(1, SI), 23–28 (2011)
4. Kronbichler, M., Kormann, K.: A generic interface for parallel cell-based finite element operator application. Computers & Fluids 63, 135–147 (2012)
5. Cecka, C., Lew, A.J., Darve, E.: Assembly of finite element methods on graphics processors. International Journal for Numerical Methods in Engineering 85, 640–669 (2011)
6. Kckner, A., Warburton, T., Bridge, J., Hesthaven, J.S.: Nodal discontinuous Galerkin methods on graphics processors. Journal of Computational Physic 228(21), 7863–7882 (2009)
7. Komatitsch, D., Micha, D., Erlebacher, G.: Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. Journal of Parallel and Distributed Computing 69(5), 451–460 (2009)
8. Berger, P., Brouaye, P., Syre, J.C.: A mesh coloring method for efficient MIMD processing in finite element problems. In: Proceedings of the International Conference on Parallel Processing, pp. 41–46 (1982)
9. Farhat, C., Crivelli, L.: A General-Approach to Nonlinear Fe Computations on Shared-Memory Multiprocessors. Computer Methods in Applied Mechanics and Engineering 72(2), 153–171 (1989)
10. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. In: Eurographics 2005, State of the Art Reports, pp. 21–51 (2005)
11. NVIDIA Corporation: NVIDIA CUDA C Programming Guide, Version 5.5 (July 2013)
12. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. Journal of Computational Physics 227(10), 5342–5359 (2008)
13. Elsen, E., LeGresley, P., Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU. Journal of Computational Physics 227(24), 10148–10161 (2008)
14. Micha, D., Komatitsch, D.: Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. Geophysical Journal International 182(1), 389–402 (2010)