IT Licentiate theses
2015-001

Techniques for Finite Element Methods on Modern Processors

KARL LJUNGKVIST

UPPSALA UNIVERSITY
Department of Information Technology

Techniques for Finite Element Methods on Modern Processors

Karl Ljungkvist
karl.ljungkvist@it.uu.se

January 2015

Division of Scientific Computing
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

Dissertation for the degree of Licentiate of Philosophy in Scientific Computing

© Karl Ljungkvist 2015
ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

In this thesis, methods for efficient utilization of modern computer hardware for numerical simulation are considered. In particular, we study techniques for speeding up the execution of finite-element methods.

One of the greatest challenges in finite-element computation is how to efficiently perform the the system matrix assembly efficiently in parallel, due to its complicated memory access pattern. The main difficulty lies in the fact that many entries of the matrix are being updated concurrently by several parallel threads. We consider transactional memory, an exotic hardware feature for concurrent update of shared variables, and conduct benchmarks on a prototype processor supporting it. Our experiments show that transactions can both simplify programming and provide good performance for concurrent updates of floating point data.

Furthermore, we study a matrix-free approach to finite-element computation which avoids the matrix assembly. Motivated by its computational properties, we implement the matrix-free method for execution on graphics processors, using either atomic updates or a mesh coloring approach to handle the concurrent updates. A performance study shows that on the GPU, the matrix-free method is faster than a matrix-based implementation for many element types, and allows for solution of considerably larger problems. This suggests that the matrix-free method can speed up execution of large realistic simulations.

Acknowledgments

First of all, I would like to thank my main advisor Jarmo Rantakokko for your help, advice and encouragement. Secondly, I thank my two co-advisors, Gunilla Kreiss and Sverker Holmgren, for your valuable experience and contacts, and for providing a broader picture. I also thank my co-authors Martin Tillenius, David Black-Schaffer, Martin Karlsson, and Elisabeth Larsson for collaboration. Furthermore, I am grateful for my colleagues and friends at TDB for providing a friendly atmosphere and making it a pleasure to go to work.

The work that resulted in Paper I was supported by the Swedish Research Council and carried out within the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center. The experiments in Paper II were conducted on computer systems provided by UPMARC. Finally, the travel grants received from Ångpanneföreningens Forskningsstiftelse and Anna-Maria Lundins stipendiefond are gratefully acknowledged.

List of Papers

This thesis is based on the following two papers referred to as Paper I and Paper II.

I K. Ljungkvist, M. Tillenius, D. Black-Schaffer, S. Holmgren, M. Karlsson, and E. Larsson. Using Hardware Transactional Memory for High-Performance Computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on* , pages 1660-1667, May 2011.
Contribution: Development and experiments were conducted in close collaboration with the second author. The manuscript was written by the first two authors in collaboration with the remaining authors.

II K. Ljungkvist. 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*, pages 450-461. Springer International Publishing, 2014.
Contribution: The author of this thesis is the sole author of this paper.

Contents

Chapter 1

Introduction

1.1 Setting

When solving partial differential equations using the finite element method, the standard procedure consists of two distinct steps, a) an Assembly Phase, where a system matrix and right hand side vector are created to form a linear system, and b) a Solve Phase, where the linear system is solved. In most problems, most of the time is spent solving the linear system, and therefore, parallelization of the Solve Phase is a well studied problem, and derives on existing methods for parallel solution of sparse linear systems. On the other hand, the Assembly Phase is a more difficult task involving unstructured data dependencies.

However, with efficient parallelizations of Solve Phase in place, it becomes increasingly important to speed up also the Assembly Phase. This is also important for problems where the Assembly Phase needs to be performed repeatedly throughout the simulation, such as non-linear problems problems with time-dependent data.

A very related problem shows up when using the finite-element method with a matrix-free approach, where the Assembly Phase is merged into the sparse matrix-vector product inside linear solver, thus getting rid of the explicit system matrix. Since the resulting Operator Application operation is performed with a high frequency throughout the simulation, efficient implementation of this operation is crucial.

This matrix-free approach is motivated by the limited memory available for the system matrix, but also by recent trends in processor design where memory bandwidth is becoming the main bottleneck. The Operator Application is essentially equivalent to the assembly, so techniques for parallelization and optimization of the assembly are largely applicable also to the Operator Application.

One of the fundamental difficulties when performing the matrix assembly or matrix-free operator application, is the concurrent update of shared variables. Both these operations are computed as sums of contributions from all elements in the finite-element mesh, where the contributions from a single element correspond to degrees-of-freedom residing within that element, and they are typically parallelized by computing the sum and the contributions in parallel. Since many degrees-of-freedom are shared between multiple elements, many destination variables will be affected by multiple concurrent updates from different elements. This poses the challenge to avoid race conditions and maintain correctness.

This thesis concerns the efficient implementation of the matrix assembly and matrix-free operator application on modern parallel processors.

1.2 Disposition

In Chapter 2, the finite-element method is introduced, including the matrix-free version. In Chapter 3, recent trends in processor hardware are discussed; in particular, graphics processors and hardware transactional memory. In this context, the main contributions of the two papers are also presented. Finally, in Chapter 4, a overview of further ongoing and future work is found.

Chapter 2

Finite Element Methods

2.1 Background

The finite-element method (FEM) is a popular numerical method for problems with complicated geometries, or where flexibility with respect to adaptive mesh refinement is of importance. It finds applications in, e.g., structural mechanics, fluid mechanics, and electromagnetics [1]. Rather than solving the strong form of a partial differential equation, the finite element method is finds approximations of solutions to the variational, or weak, formulation. Consider the example of a Poisson model problem,

$$\begin{cases} \nabla^2 u = f & \text{on } \Omega\,, \\ u = 0 & \text{on } \partial\Omega\,. \end{cases}$$

The corresponding weak formulation is obtained by multiplying by a test function v in a function space $\mathcal{V}$, and integrating by parts, obtaining

$$(\nabla u, \nabla v) = (f, v)\ \forall v \in \mathcal{V}\,.$$

From this, the Finite Element Method is obtained by replacing the space $\mathcal{V}$ with a finite-dimensional counterpart $\mathcal{V}_h$. This is typically done by discretizing the domain Ω into a partitioning $\mathcal{K}$ of elements. $\mathcal{V}_h$ is then usually chosen to be the space of all continuous continuous functions which are polynomial within each element. There are different types of finite elements but the most popular ones include *simplices* (triangles, tetrahedra, etc.), as in Paper I, and quadrilaterals/hexahedrons, as in Paper II. For the case of triangle elements and their higher-dimensional generalizations, one typically considers functions polynomial of order p, referred to as $\mathcal{P}^p$. For quadrilateral elements, one instead usually considers functions polynomial of order p in each coordinate direction, e.g., bilinear for $p = 1$ or biquadratic for $p = 2$ in 2D. These

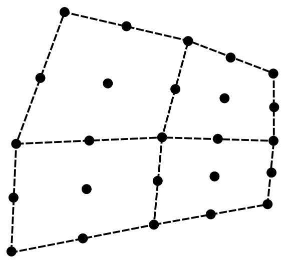


Figure 2.1: *Location of the node points for second-order quadrilateral elements in 2D ($\mathcal{Q}^2$).*

elements are referred to as $\mathcal{Q}^p$. In the remained of this introduction, we assume $\mathcal{Q}^p$ elements, but $\mathcal{P}^p$ are treated in a similar fashion.

To fix a unique such polynomial within an element, we need to determine its values at $(p+1)^D$ node points (see Figure 2.1). Introducing basis functions $\{\varphi_i\}$ in $\mathcal{V}_h$, which are zero at all node points except the i'th where it equals one, we can expand the solution u in this basis, and use the fact that we may replace v with φ_i since they constitute a basis in V_h. We then arrive at the following discrete system

$$\sum_{i=1}^{N} (\nabla\varphi_j, \nabla\varphi_i)\, u_i = (f, \phi_j) \;\; j = 1 \ldots N$$

or, in matrix notation,

$$A\mathbf{u} = \mathbf{f}, \tag{2.1}$$

where

$$A_{ij} = (\nabla\varphi_j, \nabla\varphi_i)\,, \; \mathbf{f}_j = (f, \phi_j)$$

which has to be solved for the unknowns u_j, also referred to as *degrees-of-freedom* (*DoFs*). For a more complicated problem, e.g., involving time-dependence or non-linearities, a similar system will have to be solved in a time-stepping iteration or Newton iteration.

In summary, the computational algorithm for finding the finite-element solution consists of the following two phases:

1. Assembly phase
2. Solve phase

In the first phase, the system matrix A and the right-hand side vector $\mathbf{f}$ are constructed. The second phase consists of solving the system in (2.1). In most applications, the solve phase is the most time consuming of the two. Therefore, speeding it up has been the target of much research. Since the matrix is, in general, very large and very sparse, iterative Krylov-subspace methods are typically used for the solution. Within such iterative methods, most of the work is spent performing matrix-vector multiplications [2]. Parallelizations of the solve phase thus relies on efficient parallelization of the matrix-vector product for sparse matrices, which is a well studied problem. On the other hand, the assembly phase, which is a conceptually more complicated problem involving data dependencies and updating of shared variables, has only recently been attacked.

2.2 The Matrix Assembly

For the example of a stiffness matrix, the assembly consist of the following computation,

$$A_{ij} = (\nabla\varphi_i, \nabla\varphi_j) = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}\mathbf{x}\,.$$

Now the integral is split into a sum over all the elements in the mesh $\mathcal{K}$,

$$A_{ij} = \sum_{k\in\mathcal{K}} \int_{\Omega_k} \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}\mathbf{x}\,. \tag{2.2}$$

Since each basis function φ_i is non-zero only at the i'th DoF and zero on all others, it will only be non-zero on elements to which the i'th DoF belong. This effectively eliminates all but a few combinations of basis functions from each integral in the sum, i.e., the matrices in the sum are very sparse. 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 $\varphi_l^k = \varphi_{I^k(l)}$ as the l'th basis function on element k, we can write (2.2) on matrix form as

$$A = \sum_{k\in\mathcal{K}} P_k a^k P_k^T\,, \tag{2.3}$$

where the local matrix a^k is defined as

$$a_{l,m}^k = \int_{\Omega_k} \nabla\varphi_l^k \cdot \nabla\varphi_m^k \mathrm{d}\mathbf{x}\,. \tag{2.4}$$

To summarize, the assembly consists of computing all of the local matrices a^k, and summing them up. The effect of the multiplication with the permutation matrices is merely that of distributing the element of the small and dense matrix a^k into the appropriate locations in the large and sparse matrix A.

2.2.1 Parallelization

The standard way of parallelizing the assembly is to have multiple parallel threads compute different terms in the sum of (2.3). Since each of the local matrices are independent of each other, they can readily be computed in parallel. However, when the contributions are distributed onto the result matrix A, great care must be taken to ensure correct results.

As can be seen in Figure 2.1, a given degree of freedom is in general shared between multiple elements. This means that multiple contributions in the sum in (2.3) will update the same memory location. It is thus of great importance to perform the updates in a way such that race conditions are avoided (see Section 3.2). In Paper I, we avoid the race conditions by using various primitives for safe update of variables, chiefly transactional memory but also locks and atomic intrinsics. In Paper II, in addition to atomic intrinsics, we also utilize a mesh coloring approach where we can process the elements in groups such that no race conditions arise (see Section 2.3.2)

Alternative approaches for the parallelization have also been suggested, such as a node- or row-based parallelization where each thread is responsible for computing a single entry or row in the matrix [3, 4]. This removes the concurrent updates of shared variables, but instead introduces a lot of extra computations, as the local-element matrix has to be recomputed for each matrix entry it contributes to.

2.3 A Matrix-Free Approach

There are several problems with the two-phase approach considered in Section 2.1. First of all, as the degree of the finite elements is increased, the system matrix gets increasingly large and less sparse, especially for problems in 3D. In these cases, the system matrix A may simply be too large to fit in the memory of the computer.

Secondly, the sparse matrix-vector multiplications (SpMV) constituting the bulk of work in the solve phase are poorly suited for execution on modern processors, since the number of arithmetic operations per memory access is low, making them bound by memory bandwidth rather than compute power [5, Chapter 7]. As can be seen in Figure 2.3, the *computational intensity*, defined as the number of arithmetic operations divided by the number of bytes accessed, is about 0.2, which is more than an order of magnitude lower than what is required for the most recent processors (see Section 3.4). Note that the SpMV operation has a very irregular access pattern for general matrices which reduces the utilization of the available bandwidth, requiring an even higher number of arithmetic operations per memory access.

Finally, when solving non-linear problems or problems with time-dependent coefficients, it might be necessary to reassemble the system matrix frequently throughout the simulation. This changes the relative work size of the assembly phase, and precomputing the large system matrix for only a few matrix-vector products may not be efficient. Similarly, when using adaptive mesh refinement, the matrix has to be reassembled each time the mesh is changed.

Motivated by this, several authors have suggested a matrix-free finite-element method. The idea builds on the observation that, to solve the system using an iterative method, the system matrix A is never needed explicitly, only its effect as an operator $\mathcal{A}[\,\cdot\,]$ on a vector v. This means that if we can find a recipe $\mathcal{A}[v]$ for how to form Av without having access to an explicit A, we can use $\mathcal{A}[v]$ in the linear solver, just as we would have used Av.

Matrix-free methods have been used for a long time in computational physics in the form of Jacobian-Free Newton-Krylov methods to solve non-linear PDEs. There, by approximating the Jacobian-vector product Jv directly, one avoids having to explicitly form the Jacobian J of a Newton iteration for solving non linear problems, reducing space and computation requirements within the iterative linear solver [6].

The Spectral Element Method is a popular choice for hyperbolic problems with smooth solutions, which essentially is a finite-element method with very high-order elements; up to order ten in many cases [7]. Because of this, one can get the flexibility of finite-element methods combined with the rapid convergence of spectral methods. The latter are generalizations of the Fourier method, and converge at an *exponential* rate, in contrast to finite-difference methods or finite-element methods which only converge at an arithmetic rate [8]. Because of the hyperbolic nature of the PDEs involved, they can be integrated explicitly in time, and no linear system has to be solved during the simulation. Still, one needs to apply a FEM-type differential operator to the solution once to march forward in time, making a matrix-free approach interesting. Also, due to the typically very high number of degrees-of-freedom per element in spectral-element methods, the resulting matrix has a very low sparsity, which further motivates that a matrix-free technique can be beneficial. Usually, it is then based on the tensor-product approach described in Section 2.3.3.

Melenk et al. investigate efficient techniques for assembly of the spectral-element stiffness matrix, based on a tensor-product approach [9], In [10], Cantwell et al. compare matrix-free techniques based on a local matrix and tensor-product evaluation, with the standard sparse-matrix approach, and find that the optimal approach depends on both the problem setup and the computer system.

Kronbichler and Kormann describe a general framework implementing tensor-based operator application parallelized using a combination of MPI for inter-node communication, multicore threading using Intel Threading Building Blocks, and SIMD vector intrinsics [11]. The framework has been included in the open-source finite-element framework `deal.II` [12, 13].

We form the matrix-free operator by using the definition of A from (2.3),

$$\mathcal{A}[v] = Av = \left(\sum_{k \in \mathcal{K}} P_k a^k P_k^T \right) v = \sum_{k \in \mathcal{K}} \left(P_k a^k P_k^T v \right) .$$

In other words, we have simply changed the order of the summation and multiplication. Once again, the P_k simply permutation matrices which simply picks out the correct DoFs and distributes them back, respectively. In summary, the algorithm consists of (a) reading the local degrees of freedom v_k from the global vector v, (b) performing the local multiplication, and (c) summing the resulting vector u_k into the correct positions in the global result vector u,

$$\begin{array}{ll} \text{(a)} & v_k = P_k^T v \\ \text{(b)} & u_k = a^k v_k \\ \text{(c)} & u = \sum_{k \in \mathcal{K}} P_k u_k \end{array} \tag{2.5}$$

This is essentially a large number of small and dense matrix-vector products which lends itself well to parallelization on throughput-oriented processors such as GPUs. Due to the similarity of this operation with the assembly, the same parallelization approach can be used, where we evaluate the local products in parallel. Also, for handling the concurrent updates of shared variables, in this case the DoFs, the same techniques can be used (see Section 3.5.2).

2.3.1 Evaluation of Local Operator

Although the matrix-free operation as defined in (2.5) is a collection of dense operations, and thus significant improvement over the initial sparse matrix-vector product, the actual computational properties will depend on the evaluation of the local matrix-vector product,i.e, operation (b). To evaluate the integral in the definition of the local matrix, see (2.4), a mapping $\mathbf{x} = f_k(\boldsymbol{\xi})$ from a reference unit element $\hat{\Box}$ to element k is used. Using this transformation, (2.4) can be rewritten as

$$a_{ij}^k = \int_{\hat{\Box}} \left(J_k^{-1} \hat{\nabla} \hat{\varphi}_i \right) \cdot \left(J_k^{-1} \hat{\nabla} \hat{\varphi}_j \right) |J_k| \mathrm{d}\boldsymbol{\xi} \, , \tag{2.6}$$

where J_k is the Jacobian of the transformation f_k, and $\hat{\nabla}\hat{\varphi}_i(\boldsymbol{\xi})$ is the reference element gradient of reference element basis function $\hat{\varphi}_i$. In practice, this integral is evaluated using numerical quadrature,

$$a_{ij}^k = \sum_{q=1}^{N_q} \left(J_k^{-1}(\boldsymbol{\xi}_q) \hat{\nabla} \hat{\varphi}_i(\boldsymbol{\xi}_q) \right) \cdot \left(J_k^{-1}(\boldsymbol{\xi}_q) \hat{\nabla} \hat{\varphi}_j(\boldsymbol{\xi}_q) \right) |J_k(\boldsymbol{\xi}_q)| w_q \,, \tag{2.7}$$

where $\boldsymbol{\xi}_q$ and w_q are the N_q quadrature points and weights, respectively. Typically, Gaussian quadrature is used since polynomials of arbitrary degree can be integrated exactly, provided enough points are used.

Now, if the mesh is uniform, i.e., if all elements have the same shape, then the Jacobian will be independent of k, and all the a^k will be equal. In this case, we can precompute a single local matrix a, which can be read by all the threads during the multiplication. In this case, a very favorable computational intensity can be achieved (see Figure 2.3).

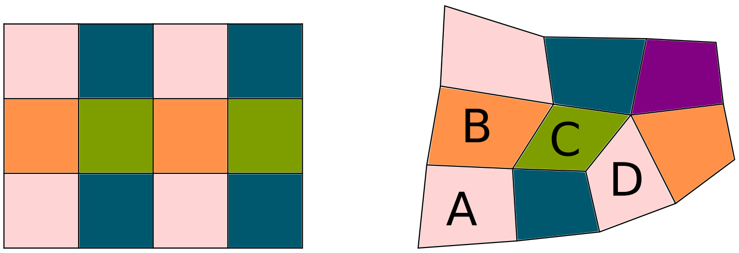
For a general mesh, this cannot be assumed, and precomputing and storing the individual local matrices yields much too much data to be efficient. In this case, a tensor-product approach can be used to decrease the data usage, and increase the computational intensity (see Section 2.3.3). This is also the case for non-linear problems, even for a uniform mesh, since the a^k depend on the solution on element k.

In Paper II, we study the performance of the matrix-free approach in the case of a constant local matrix. Although this is restricted to linear problems on uniform meshes, it is still interesting to consider since in many cases, a uniform mesh can be used for large parts of the computational domain. The experiments and results are discussed in detail in Section 3.5.2.

2.3.2 Mesh Coloring

In addition to the approaches for handling concurrent updates presented in Sections 3.2 and 3.3, we have also looked at a technique based on mesh coloring. This idea uses the fact that collisions can only appear between very specific combinations of elements, namely the ones sharing a degree-of-freedom. Thus, if we can process the elements in a way such that these specific combinations are never run concurrently, we can avoid the race conditions altogether. Looking at Figure 2.1, we note that only elements sharing a vertex will share DoFs with each other. Therefore, we would like to find a partitioning of the elements into groups, or *colors*, such that no two elements within a given color share a vertex. We could then process the elements a single color at a time, and prevent all conflicts from appearing.

Since we do not process all elements at once, the parallelism is reduced by a factor $\frac{1}{N_c}$ where N_c is the number of colors. In general, N_c will be equal to the maximum *degree* of the vertex graph, which is defined as the number of elements sharing a vertex. For logically Cartesian meshes, $N_c = 2^d$, since all interior vertices are shared by the same number of elements; 2^d, where d is the dimensionality of the problem. On the other hand, for more general meshes, N_c will be larger. Still, in both cases N_c is limited and independent of the size of the mesh, and the overhead will not grow with mesh refinement. Therefore, for large enough problems, the overhead will be small. The situation is illustrated in Figures 2.2 (a) and (b).



(a) apa (b) prutt

Figure 2.2: *Coloring of (a) a uniformly Cartesian mesh, and (b) a more general mesh. For (b), note that A and B share a vertex and are thus given different colors, as are A and C. Since A and D do not share any vertices, they can be given the same color.*

For a general mesh, finding the coloring can be done using various graph coloring algorithms [14, 15, 16]. In Paper II, where we use a uniform logically Cartesian mesh, handing out the colors is trivial. We assign colors by computing a binary number based on the index coordinates of an element. In 3D, the formula for the color $c \in \{0, 1, \ldots, 2^d - 1\}$ is

$$c = (i_x \bmod 2) + 2 \cdot (i_y \bmod 2) + 4 \cdot (i_z \bmod 2)\,,$$

where i_x, i_y, and i_z are the index coordinates of the element in the x, y, and z directions, respectively.

We note that, while the reduction of parallelism is low, this approach still introduces some superficial dependencies, since not all elements of different color will conflict with each other (e.g. elements from different corners of the domain). In particular, this introduces a global barrier between the processing of one color and the next one, preventing overlapping of elements although conflict free ones could be found. Resolving this situation completely is a difficult problem, but one way to solve it is to consider the dependency graph of the problem, which can be achieved using a task-based parallelization framework. One such task library is SuperGlue, which can handle complex dependencies, including atomicity [17, 18]. This type of dependency can be used to express the situation described in Section 3.2, namely that two operations can be performed in any order, but not concurrently. In [18], Tillenius applies SuperGlue and achieves very good scaling for an N-body problem similar to the one considered in Paper I (see Section 3.3.4)

2.3.3 Tensor-Product Local Operator

When the local matrix is not the same for all elements in the mesh, then precomputing and storing the local matrices becomes impractical due to the high memory requirement, which even exceeds that of the sparse matrix. In this case, we can take another step in the same direction as the original matrix-free approach, and exploit the specific structure of the local matrix. Once again, we note that the local matrix a^k is never needed explicitly, only its effect upon multiplication with a local vector v,

$$u_i = \sum_{j=1}^{n_p} a_{ij} v_j \,,$$

where n_p is the number of local DoFs. Equation (2.7) lets us rewrite this operation in the following manner,

$$u_i = \sum_{q=1}^{n_q} \left(J^{-1}(\boldsymbol{\xi}_q)^T \hat{\nabla}\hat{\varphi}_i(\boldsymbol{\xi}_q) \right) \cdot \left[J^{-1}(\boldsymbol{\xi}_q) \sum_{j=1}^{n_p} \hat{\nabla}\hat{\varphi}_j(\boldsymbol{\xi}_q) v_j \right] |J(\boldsymbol{\xi}_q)| w_q \,, \tag{2.8}$$

where the element index k has been dropped. The entity in brackets is simply the gradient of v evaluated on the q'th quadrature point. Defining,

$$\nabla v_q = J^{-1}(\boldsymbol{\xi}_q) \sum_{j=1}^{n_p} \hat{\nabla}\hat{\varphi}_j(\boldsymbol{\xi}_q) v_j \,, \tag{2.9}$$

(2.8) can be written as

$$u_i = \sum_{q=1}^{n_q} \left(J^{-1}(\boldsymbol{\xi}_q) \hat{\nabla}\hat{\varphi}_i(\boldsymbol{\xi}_q) \right)^T \cdot \nabla v_q |J(\boldsymbol{\xi}_q)| w_q \,.$$

or

$$u_i = \sum_{q=1}^{n_q} \hat{\nabla}\hat{\varphi}_i^{\,T}(\boldsymbol{\xi}_q) J^{-T}(\boldsymbol{\xi}_q) \nabla v_q |J(\boldsymbol{\xi}_q)| w_q \,. \tag{2.10}$$

Now, for the quadrilateral or hexahedral elements under consideration, the basis functions $\hat{\varphi}_i$ can be expressed as tensor products of d one dimensional basis functions ψ_μ. Assuming three dimensions, we get

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi_{\mu_d}(\xi_d) \,,$$

where we have dropped the symbol ˆ for reference element, and introduced the multi-index $i = (\mu, \nu, \sigma)$. For the basis function gradient, this implies

$$\nabla \varphi_i(\boldsymbol{\xi}) = \begin{pmatrix} \psi'_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3) \\ \psi_\mu(\xi_1)\psi'_\nu(\xi_2)\psi_\sigma(\xi_3) \\ \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi'_\sigma(\xi_3) \end{pmatrix} .$$

Likewise, the quadrature points $\boldsymbol{\xi}_q$ can be expressed as

$$\boldsymbol{\xi}_q = (\xi^\alpha, \xi^\beta, \xi^\gamma) ,$$

where ξ^α are the one dimensional quadrature points, and $q = (\alpha, \beta, \gamma)$ is a multi-index. Note that we consistently use subscripts μ, ν, σ to index in DoF space, and superscript α, β, γ as index for quadrature points. Defining

$$\psi_\mu^\alpha = \psi_\mu(\xi^\alpha) , \; \chi_\mu^\alpha = \psi'_\mu(\xi^\alpha) ,$$

and using the multi-index notation, we can can compute the solution at the quadrature points as

$$v^{\alpha\beta\gamma} = \sum_\mu \psi_\mu^\alpha \sum_\nu \psi_\nu^\beta \sum_\sigma \psi_\sigma^\gamma v_{\mu\nu\sigma} .$$

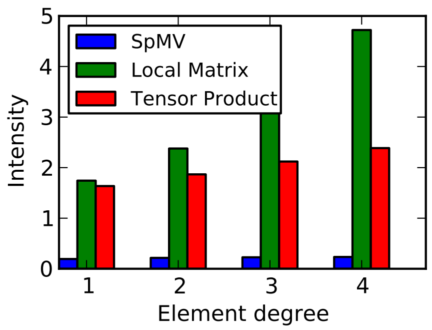
We can now rewrite (2.9) as

$$\nabla v^{\alpha\beta\gamma} = (J^{\alpha\beta\gamma})^{-1} \sum_\mu \begin{pmatrix} \chi_\mu^\alpha \\ \psi_\mu^\alpha \\ \psi_\mu^\alpha \end{pmatrix} \sum_\nu \begin{pmatrix} \psi_\nu^\beta \\ \chi_\nu^\beta \\ \psi_\nu^\beta \end{pmatrix} \sum_\sigma \begin{pmatrix} \psi_\sigma^\gamma \\ \psi_\sigma^\gamma \\ \chi_\sigma^\gamma \end{pmatrix} v_{\mu\nu\sigma} , \tag{2.11}$$

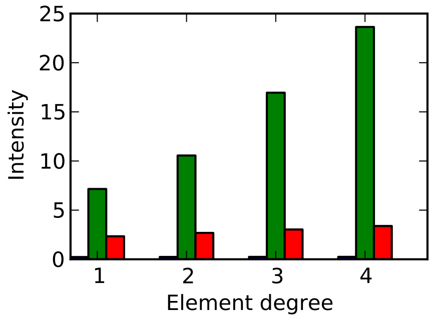
where $J^{\alpha\beta\gamma} = J(\boldsymbol{\xi}_q)$ and the vector products are to be understood as element-wise multiplication. Likewise, for the second step, the numerical integration in (2.10), we have

$$v_{\mu'\nu'\sigma'} = \sum_\alpha \begin{pmatrix} \chi_{\mu'}^\alpha \\ \psi_{\mu'}^\alpha \\ \psi_{\mu'}^\alpha \end{pmatrix} \sum_\beta \begin{pmatrix} \psi_{\nu'}^\beta \\ \chi_{\nu'}^\beta \\ \psi_{\nu'}^\beta \end{pmatrix} \sum_\gamma \begin{pmatrix} \psi_{\sigma'}^\gamma \\ \psi_{\sigma'}^\gamma \\ \chi_{\sigma'}^\gamma \end{pmatrix} \left(J^{\alpha\beta\gamma}\right)^{-T} \nabla v^{\alpha\beta\gamma} w^{\alpha\beta\gamma} |J^{\alpha\beta\gamma}| . \tag{2.12}$$

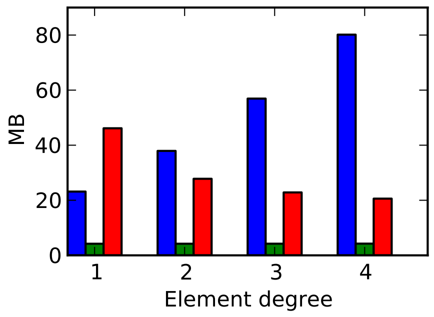
The two very similar operations (2.11) and (2.12) both consists of a series tensor contractions – essentially dense matrix-matrix products – and a couple of element-wise scalar and vector operations. It is thus an significant improvement over the initial sparse matrix-vector product when it comes to utilization computational intensity, making very attractive for execution on throughput-optimized processors such as GPUs. This can be seen in Figure 2.3, which shows the computational intensity and bandwidth usage of the three main operator application approaches considered in this thesis. It is clear that in the cases when the local matrix implementation cannot be used, the tensor-product technique is very attractive alternative, especially for elements of high complexity, i.e. high dimension and polynomial order.



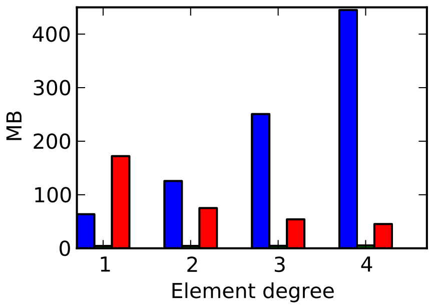
(a) Computational intensity (2D)



(b) Computational intensity (3D)



(c) Bandwidth usage (2D)



(d) Bandwidth usage (3D)

Figure 2.3: *Comparison of operator approaches with respect to bandwidth usage and computational intensity. The bandwidth is computed as the amount of data which has to be read for a single operator application.*

Chapter 3

Modern Processors

3.1 Background

As widely known, computer processors undergo an extremely fast development. According to the empirical observation of Moore's law, the number of transistors in a chip grows exponentially with a doubling every two years [19]. However, since the early 2000s this does no longer translate directly into a corresponding increase in serial performance. At that point, it was no longer possible to make significant increases in the clock frequency, since voltage could no longer be scaled down with the transistor size due to power issues. This is usually referred to as the power wall, or breakdown of Dennard scaling [20]. Instead, the focus has shifted towards increasing parallelism within the processor in the form of multicore designs.

This change has increased the burden on the programmer since utilizing a parallel processor is more difficult than a serial one [5]. When writing a parallel program, the work must be split into smaller tasks that can be performed concurrently, and issues such as communication and synchronization must be addressed.

3.2 Techniques for Updating Shared Variables

One of the main problems when programming multicore processors is the coordination of memory access. In contrast to cluster computers where the memory is distributed over the nodes, a multicore processor has a single memory which is shared between the cores. The multicore processors are typically programmed using threads, which all have access to the complete shared memory. While flexible, this approach allows for subtle bugs called *race conditions*.

A race condition occurs when two threads concurrently manipulate the same memory location, resulting in undefined results (see example in Table 3.1). It is up to the application programmer or library developer to

x	Thread 1	Thread 2
0	$x_1 \leftarrow x$	
0	$x_1 \leftarrow x_1 + 1$	$x_2 \leftarrow x$
1	$x \leftarrow x_1$	$x_2 \leftarrow x_2 + 1$
1		$x \leftarrow x_2$

Table 3.1: *Two threads increment the same variable x concurrently, leading to the result $x = 1$ instead of the expected $x = 2$.*

guarantee that no race conditions can appear, and this is one of the main difficulties when writing multithreaded programs.

The code segments that potentially might conflict with each other, or with other copies of itself, is referred to as *critical sections.* In order to avoid race conditions, the critical sections need to be executed *mutually exclusively*, i.e. only a single thread is allowed within a critical section simultaneously.

The most common technique for achieving exclusivity is to use *locks*, which are mutually exclusive data structures with two operations – **`lock`**, which obtains the lock, and **`unlock`**, which releases the lock. If the critical section is surrounded with a lock and an unlock operation, then only a single thread will be allowed to be in the critical section, since any other thread will not succeed with the lock operation until the first thread has completed the unlock operation following its critical section.

Another way of achieving mutual exclusivity is the concept of *atomicity.* If all the operations in the critical section are considered an indivisible entity, which can only appear to the memory as a whole, then it can be executed safely concurrently. For simple critical sections, e.g. an incrementation of a variable, the processor architecture may offer native atomic instructions.

Atomic instructions usually perform well because of the efficient implementation in hardware. However, they cannot be used for general critical section and are limited to the available atomic instructions available. Although simple atomic instructions such as *compare-and-swap* can be used to implement somewhat more complex atomic operations, this is still very limited since, typically, not more than a single memory location can be manipulated.

Locks on the other hand are completely general, but can lead to several performance related issues. A deadlock occurs when two or more threads are waiting for each other's locks. Lock convoying is when many threads are waiting for a lock while the thread holding the lock is context switched and prevented from progressing. Priority inversion happens when a low-priority thread holds the lock, preventing execution of threads of higher priority [21].

3.3 Hardware Transactional Memory

Another more recent technique which also uses atomicity to achieve mutual exclusion is *transactional memory.* Rather than surrounding the critical section by `lock` and `unlock` operations, all the instructions of the critical section are declared to constitute an atomic transaction. Then, when the transaction is executed, the transactional memory system monitors whether any conflicting operations have been performed during the transaction. If conflicts are detected, the transaction is aborted, i.e., all of its changes to the memory system are rolled back to the pre-transactional state. On the other, if no conflicts were detected, the transaction commits, i.e., its changes are made permanent in the memory system.

Since this approach assumes a successful execution, and only deals with conflicts if they appear, it can be regarded as an *optimistic* approach. This is in contrast to the locks-based technique, where we always perform the locking even if no actual conflicts occurred, thus being a more *pessimistic* approach. This can potentially lead to lower overheads for the cases when contention is low, i.e., when conflicts are rare.

Another great benefit with transactional memory is in the ease of use. To get good performance with locks, it is necessary to dissect the algorithm and identify fine-grained dependencies in order to get the most parallelism out of it, and the problems mentioned in the previous section. On the other hand, with transactions, one simply has to declare the whole critical section as a transaction, and then the system will find conflicts automatically with a high granularity (typically cache-line size). With a sufficiently efficient implementation of the transactional memory system, this has the potential to simplify the programming of high-performance parallel programs.

Transactional Memory was first introduced by Herlihy and Moss in 1993, where they suggest an implementation based on a modified cache-coherence protocol [21]. In the following years, several studies investigating possible transactional-memory implementations in hardware or software were published [22, 23, 24, 25]. Around 2008, Sun Microsystems announced the Rock processor; the first major processor intended for commercial use to feature transactional memory in hardware [26]. Although eventually canceled after the acquisition of Sun by Oracle, several prototype Rock chips were available to researchers. More recently, IBM has included transactional memory in the BlueGene/Q processor [27], whereas Intel has introduced transactional memory in the form of the Transactional Synchronization Extensions in their Haswell processor series [28]. For both of these systems, relatively good performance when applied to scientific computing has been demonstrated [29, 30].

3.3.1 The Rock Processor

All the experiments in Paper I were performed a prototype version of the Rock processor. The Rock is a 16-core processor featuring the SPARC V9 instruction set. The cores are organized in four clusters of four cores, where each cluster shares a 512 kB L2 cache, a 32 kB instruction cache, two 32 kB L1 data caches, and two floating point units. Except for hardware transactional memory, other exotic features of the Rock processor includes Execute Ahead and Simultaneous Speculative Threading. Execute Ahead lets a single core continue performing future independent instructions in the case of a long-latency event, e.g., a cache miss or TLB miss, and return to the previous point once the lengthy operation is ready, performing a so called replay phase. In Simultaneous Speculative Threading, this is expanded even further where two cores can cooperate with performing the future independent instructions and the replay phase, thus executing a single serial instruction stream at two points simultaneously. In both of these cases, an additional benefit is that thread executing future instructions can encounter further cache misses, effectively acting as a memory prefetcher. Both EA and SST utilize the same checkpointing mechanism used to implement transactional memory [26].

The Rock supports transactional memory by adding two extra instructions:

- `chkpt <fail_pc>`
- `commit`

The `chkpt` instruction starts a transaction, and the `commit` instructions ends it. If a transaction is aborted, execution jumps to the address referred to by the `fail_pc` argument. A new register `%cps` can then be read to get information on the cause of the abort.

3.3.2 Failed Transactions

The Rock processor implements a *best-effort* transactional memory system, meaning that any transaction may fail, whereas other, *bounded*, implementations provide guarantees that transactions satisfying certain size criteria always commit. A best-effort implementation offers flexibility and has the advantage of potentially committing transactions of much larger size than on a corresponding bounded implementation. However, since transactions always may fail, even if there are no conflicting writes, a clever fail handler is necessary to ensure forward progress and performance. In Table 3.2, the various reasons for a fail are listed, along with the associated bits of the `%cps` register. In the following, we discuss the most important reasons for failure and how we handle them, based on our hands-on experience and the

Bit	**Meaning**
0x001	Invalid
0x002	Conflict
0x004	Trap Instruction
0x008	Unsupported Instruction
0x010	Precise Exception
0x020	Asynchronous Interrupt
0x040	Size
0x080	Load
0x100	Store
0x200	Mispredicted Branch
0x400	Floating Point
0x800	Unresolved Control Transfer

Table 3.2: *Reasons for a failed transaction as indicated by the bits of the* `%cps` *register. This information is based on a table in [31].*

information in [32]. The absolute majority of the failed transactions belonged to one of the following types:

Conflict Another thread tried to modify the same memory address.

Load A value could not be read from memory.

Store A value could not be stored to memory.

Size The transaction was too large.

Due to limited hardware, there are many constraints on transactions, such as the total number of instructions, and the number of memory addresses touched. In our experiments, we were able to successfully update up to 8 double precision variables (64 bytes) in a single transaction. More updates frequently resulted in the transaction failing with the **Size** bit set. However, once again we note that the transactional memory still provides no guarantees, and also smaller transactions sometimes failed with the **Size** bit set. A transaction will be aborted with the **Conflict** bit set if another core made a conflicting access to the same cache line. The **Load** or Store **Store** bits are set when the transaction tries to access memory which is not immediately available, which happens in the case of a L1 cache miss or a TLB miss. In addition, the **Store** bit is set if the memory location is not exclusive in the cache coherency protocol.

For the transactions failing due to **Conflict**, we employ a backoff strategy in our fail handler, where we introduce a random but exponentially increasing delay before retrying the transaction. This is reasonable, since if two threads accessed the same data recently, they are likely to do so again. If the transaction failed due to a **Store** error, we noticed that simply retrying it indefinitely often never led to a success. The problem is that, although the memory system notices that the data is not available or in the proper L1 cache state, and aborts the transaction, it does not fetch it and make it exclusive. To trigger this, we must write to the memory location outside of a transaction, while at the same time making sure not to change or corrupt the data. We achieve this by utilizing a trick from [32], in which we perform a dummy compare-and-swap – write zero if the memory location contains zero. After this, we retry the transaction. In the event of a **Load** error, we simply read the corresponding data outside of a transaction to have the memory system fetch it into the caches, and then retry the transaction. The fail handling scheme is summarized in Listing 3.1.

```
while transaction fails:
  if cps == conflict:
    back-off
  else if cps == store:
    compare-and-swap data
  else if cps == load:
    load data
  retry transaction
```

Listing 3.1: Strategy for handling failed transactions

In Table 3.3, we see the amount of failed transactions for the different experiments conducted on the Rock processor.

Experiment	Failed Trans.	Conflict	Load	Store
Overhead (1 update)	0.00026 %			
Overhead (8 updates)	0.0004 %			
Contention (100 %)	13.2 %			
Contention (0.098 %)	0.099 %			
FEM (few computations)	18.8 %	0.4 %	95.6 %	4.0 %
FEM (many computations)	19.2 %	0.2 %	96.7 %	3.1 %
N-body	0.4 %	51.9 %	34.0 %	3.1 %

Table 3.3: *Amount of failed transactions as percentage of the total number of updates. Included is also statistics of the fail cause, except for the microbenchmarks, where the fail reason was not recorded to minimize overhead.*

3.3.3 Microbenchmarks

In order to investigate the performance of the transactional memory system for performing floating point updates, two microbenchmarks were devised, studying the overhead of transactions, and the sensitivity to contention, respectively. In all the experiments, we tried using two types of locks: Pthread mutex and Pthread spin lock. However, since the spin lock always performed the same or worse, we exclude it from the discussion.

Overhead of Synchronization

The point of the **overhead** microbenchmark is to see if transactions have lower overhead than the other synchronization techniques. In this program, we let a single thread perform one million increments of a small floating point array using protected updates. In addition to performing the update inside an atomic transaction, surrounding the update with a lock, and performing the update using the compare-and-swap instruction, we also include a reference version which does no protection at all, constituting base line. Since only a single thread is updating the array, overhead related to contention should not affect the result. To see if more updates can compensate for the overhead, we vary the size of the array from one to eight doubles. The compare-and-swap version is limited to a single double precision value and cannot be used for larger arrays if the whole array should be updated in a single atomic operation. Note that for cases where it is acceptable to ensure atomicity elementwise, the compare-and-swap technique would likely perform very poorly since the overhead is proportional to the number of updates. Finally, the Rock processor can only perform compare-and-swap on integer values, incurring some overhead in the form of conversion between floating point and integer numbers, which involve stores and loads.

In the version based on transactions, we do not utilize any fail strategy other than simply retrying the transaction if it failed. However, we do make sure that all the data is present in the cache before the experiment to avoid the problem with **Store** errors described in Section 3.3.2. To get information on the amount of failed transactions, we count both the number of started transactions, and the number of successful ones.

Since the operation we are trying to benchmark – the update of a few floating point variables – is very small, we have to be very careful during the implementation. For our first version, which was implemented in C, we observed that the compiler produced very different results for the different methods, yielding large performance differences. Therefore, we instead reimplemented all of the benchmarks in assembly code, which produced much more similar programs, and also improved the overall performance. Moreover, to further minimize the difference between the different methods, we count the number of failures also for the methods where all updates will succeed.

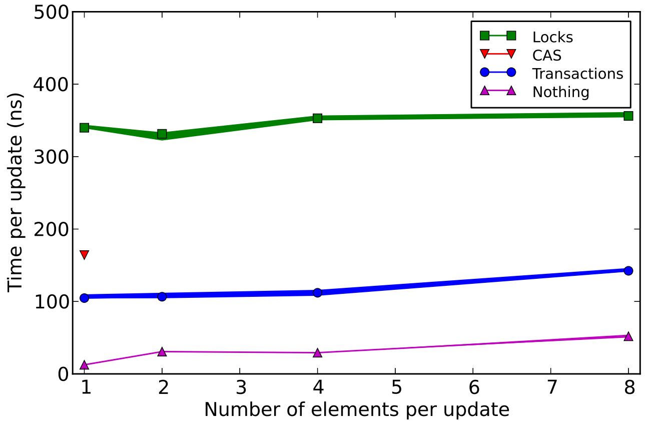


Figure 3.1: *Result of the* ***overhead*** *benchmark. The width of the lines show the variation from 100 runs, measured as lower to upper quartile.*

Looking at Figure 3.1, we see that the results confirmed our prediction that transactions have the lowest overhead. Although we observed a few spurious failed transactions – see Table 3.3, these do not affect the results and are neglected. We also see that, while limited to a single entry, the compare-and-swap version performed decently in that case.

Effect of Contention

In the **contention** microbenchmark, we want to see how the performance is affected if we add contention in the form of other threads trying to update the same memory location. We let one thread per core update its own entry in an array of length 16, and introduce contention by having threads 2 – 16 sometimes also write to the memory location of thread 1. It is not entirely straightforward how to go about to attain a desired level of conflicts, since although two threads write to the same location, a conflict might not actually happen. By controlling with what probability threads 2 – 16 write to the shared location, we can quantify percentage of potential conflicts. Although this is no perfect measure of the number of conflicts, the two will definitely be correlated, and at probability 100 %, in which case all threads always update the same location, we will have the highest amount of contention possible. Furthermore, we avoid false sharing by making sure that array entries reside their own cache lines. The experiment consists of each thread performing one million updates each.

Here, we include three versions; using transactions, locks and compare-

and-swap, respectively. Since we expect quite a few conflicts, the version based on transactions includes the fail handling strategy in Listing 3.1. The compare-and-swap version employs a simplified strategy involving only the exponential backoff. To see what number of actual conflicts we obtain for a given contention probability, we maintain a counter of the number of transactions failing on the initial try. We also measure the number of conflicts for the locks case, which is achieved by first attempting to grab the lock using the **`pthread_mutex_trylock`** function, which gives us information on whether the lock was available on the first try or not. If there was a conflict, we simply retry grabbing the lock, but this time using the regular blocking **`pthread_mutex_lock`**.

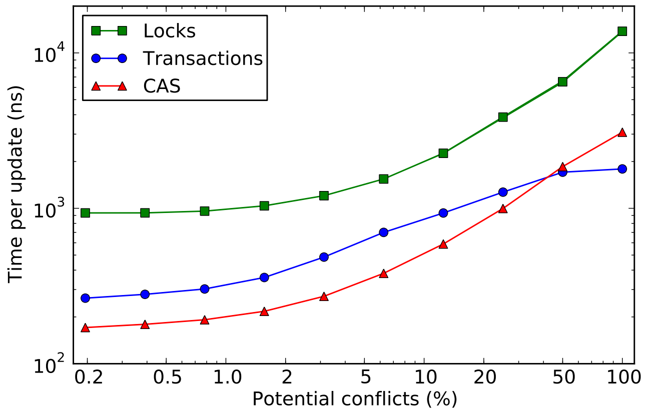


Figure 3.2: *Results of the **contention** benchmark, presented as the median of 100 runs. The width of the lines show the variation from lower to upper quartile.*

In Figure 3.2, we see that the version using locks once again performed the worst. What was more surprising was the results for transactions and compare-and-swap. We had expected transactions to have a low overhead when contention was low, and a high overhead when contention was high as many transactions would fail and have to be retried. Also, from the overhead experiment, we expected compare and swap to be slower than transactions due to the higher overhead, but possibly be less sensitive to contention. In contrast, the results show that compare-and-swap was faster than transactions for low contention, whereas transactions leveled off for high contention making it the fastest method in that case. One explanation for this can be that the store buffer serializes the writes, stopping more transactions than necessary from failing.

From Table 3.3, we conclude that there indeed was a clear correlation between potential and actual conflicts. However, we also see that even if

16 threads update a single location as frequently as possibility (the “100 %” case), only a fraction of the transactions actually fail. While we do use the exponential backoff, the initial attempts are in no way designed to avoid conflicts.

3.3.4 FEM Assembly Experiment

In Paper I, the performance of the transactional memory system of the Rock processor for performing floating point updates is studied. We compare the performance of three different versions; one based on locks, one based on atomic operations using the compare-and-swap instruction, and one based on transactional memory.

Since transactional memory has the potential to simplify implementation of parallel algorithms, we consider a straight-forward assembly based on (2.3). For instance, this means that we store the matrix in a full format, since a sparse format would require knowledge of the sparsity pattern beforehand. Although this limits how large problems we can study, it should still let us get a rough estimate of the expected performance. While it does use much more bandwidth and introduce a lot of additional cache misses compared to a sparse matrix format, the memory pattern is still similar to what can be expected when assembling a much larger sparse matrix of the same size as our full matrix. Also, by storing the matrix elements more sparsely, we largely avoid false sharing.

To study how the performance is effected by the number of operations within the evaluation of the local matrix, we have considered two versions; one with only a few computations, and one with many computations. The actual computations are artificial in both cases. The second version represents more advanced methods, such as multi-scale finite-element methods, where a smaller finite-element problem is solved within each element [33].

Figure 3.3 shows the results of the finite-element assembly application. We can see that the compare-and-swap implementation was fastest for the memory-intensive version with few operations, whereas for the compute-intensive version with many operations, transactions were the fastest. This confirms that transactions perform the best when contention is low. The locks version was the poorest overall. Also, we note the large statistical variation due to the rather short run times of this experiment. Finally, we observed a very low speedup for the memory-intensive version – 3.3x at best for the compare-and-swap implementation – which can be explained by the fact that the application is memory bound.

Considering the amount of aborted transactions (Table 3.3), we see that there was a relatively high amount of aborts – the highest of all experiments. However, looking at the distribution of causes, we see that there was in fact

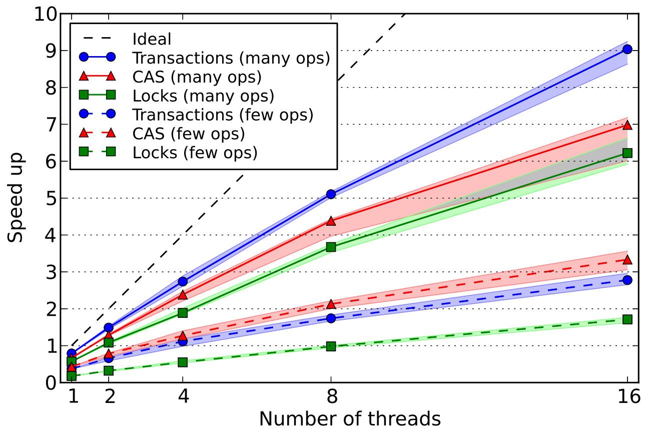


Figure 3.3: *Median speedup over serial for the different finite-element assembly approaches. The shaded areas show the variation from lower to upper quartile.*

almost no conflicts. Rather, the vast majority of all the fails are caused by cache misses or other memory-related problems. This is not surprising as most of the variables are stored in their own cache line, and are only changed very rarely.

N-Body Simulation

In Paper I, we also study a similar problem; an N-body simulation. This is a very common model in scientific applications, such as cosmology [34], molecular dynamics [35], and even machine learning [36]. In the N-body simulation, the forces acting between a number of particles have to be calculated. Since the force on a given particle consists of a sum of contributions from all other particles, it also involves concurrent updates of shared variables. Once again, we have only studied the simple all-to-all interaction. More advanced algorithms for N-body simulation include, e.g., Barnes-Hut [37], or the Fast Multipole Method [38].

Just as for the FEM application, we evaluate a locks-based version, one using compare-and-swap, and one based on transactions. To decrease the overhead, we consider chunks of particles and update a whole chunk using a single transaction or lock. This approach is not available for the compare-and-swap version, due to the limitation to a single variable. Finally, we include a more advanced algorithm using privatization of memory to avoid the conflicts altogether, at the price of more memory and more effort from the programmer.

In Figure 3.4, we see the results of the N-body experiment, from which we can conclude that transactions performed slightly better than locks. Compare-and-swap performed considerably worse, probably because of the lack of blocking. However, we also see that the more advanced implementation based on privatization is still the superior version.

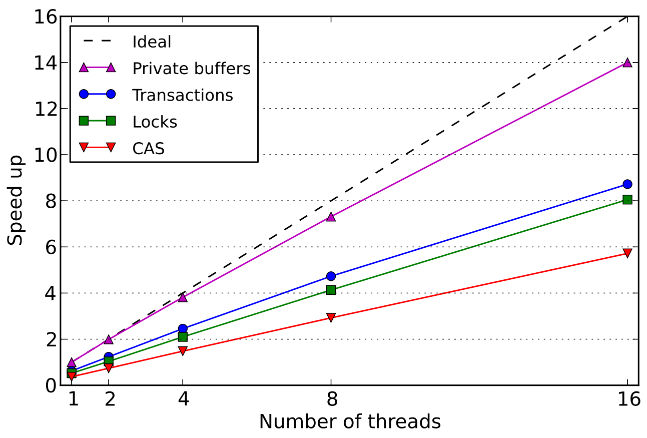


Figure 3.4: *Results of the N-body experiment as speedup over the fastest serial version. The (negligible) width of the lines show the statistical variation from first to third quartile.*

In this case, only 0.4 % of the transactions failed (see Table 3.3). The reason for this surprisingly low number is that the algorithm has a low memory footprint – only 900 cache lines compared to more than 21 000 cache lines for FEM application. In addition, this data is accessed in a fairly dense matter, due to the the chunking technique. This means that most of the memory accesses will hit in the cache. Out of the transactions that did fail, the majority were caused by conflicting writes. Note that for this application as much as 11 % of the failed transactions had an unknown cause.

Conclusion

Summarizing the results from both applications, we conclude that transactions were faster than both locks and compare-and-swap for updating shared floating-point variables, but that avoiding concurrent updates is the best choice when it is available. However, we also saw that for some applications, such as the compute-intensive matrix assembly, transactions can provide a moderate speed up also when a very naive algorithm is used. With the complicated fail handling strategy provided by a library, and with better compiler support for transaction programming, this certainly confirms that transactional memory can simplify parallel programming.

3.4 Computational Intensity Trends

In addition to the arrival of multicore processors, another problem which has become more severe recently is the fact that memory bandwidth has not scaled at the same rate as the processors.This situation was not changed by the transition to multicore processors; many slow cores will need as much memory bandwidth as a single fast one. Between 2007 and 2012, the processing power (Gflop/s) of Intel processors increased by about 30 % annually, whereas corresponding increase in the bandwidth (GB/s per socket) was only about 14 %. This balance between computations and bandwidth can be characterized by the *computational intensity*, which is defined as the ratio of the peak processing power in Gflop/s to the maximum bandwidth in GB/s. Looking at Figure 3.5, we see that recent multicore processors and GPUs have a double-precision computational intensity of about 5-10. This means that for each double-precision floating-point number (8 bytes) fetched from the memory, about 40-80 double-precision arithmetic operations are required.

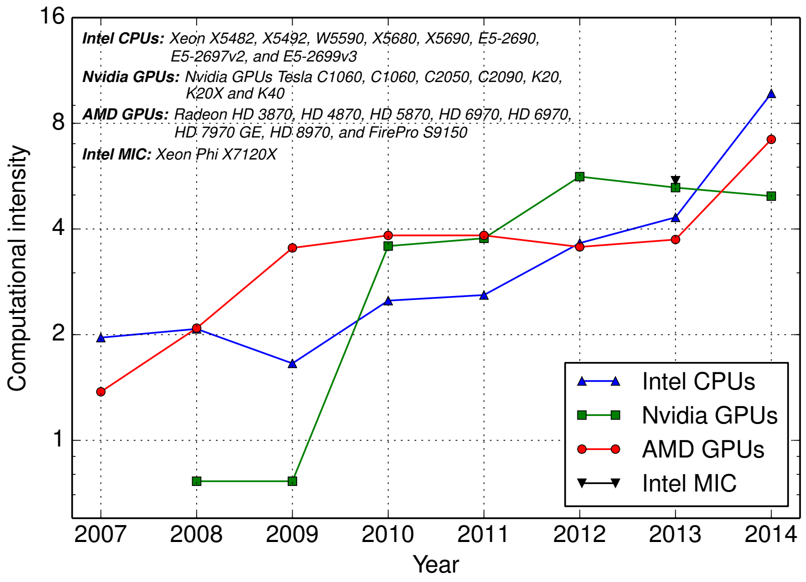


Figure 3.5: *Change in computational intensity for processors during the last six years for double precision operations. Graph based on a blog post by Karl Rupp [39].*

We therefore have a situation where most of the 'performance improvement' of new processors will come in the form of additional computations per memory fetch. Only for applications with a high enough computational intensity will this translate to any actual performance improvement. For applications which are dominated by operations limited by bandwidth, e.g., sparse matrix-vector products, stencil computations, or FFT, we can not expect any speed up.

This means that, in order to continue speeding up our applications by leveraging new capacity, we have to find algorithms that better match upcoming processor architectures. In practice, this means algorithms which need less memory bandwidth, possibly at the price of requiring additional computations. Indeed, as computations become abundant, being wasteful and recomputing previous results might actually prove beneficial.

Next, we look at one of the main new trends in computer processors – graphics processors, and how the high-performance computing community has been taking advantage of them.

3.5 Graphics Processors

In recent years, programming of graphics processing Units (GPUs) for general computations have become very popular. Driven by the insatiable demand from the gaming market for ever more detailed graphics, the GPUs have evolved into highly parallel streaming processors capable of performing hundreds of billions of floating point operations per second.

The design of GPUs is streamlined to the nature of their workload. Computer graphics essentially consists of processing a very large number of independent polygon vertices and screen pixels. Because of the very large number of tasks, there is no problem with executing each individual task slow as long as the overall throughput is high. Therefore, most of the transistors of a GPU can be used for performing computations. This is in contrast to CPUs, which are expected to perform large indivisible tasks in serial, or a few moderately sized tasks in parallel, possibly with complicated inter-dependencies. To optimize for this workload, i.e. making a single task finish as quickly as possible, a considerable amount of the hardware of a CPU – in fact most of it – is dedicated to non-computation tasks such as cache, branch prediction and coherency. Also, to get the necessary data for all the individual work items, the memory system of graphics cards tend to be focused on high bandwidth, whereas the caching system of a CPU aims at achieving low latency. Finally, as computer graphics in many cases can tolerate a fairly low numerical precision, the GPU architecture has been optimized for single-precision operations. This means that while CPUs can typically perform operations in single precision twice as fast as in double precision, this factor is of the order of 3-8 for GPUs. As a consequence of the higher computing power per transistor, GPUs achieve a higher efficiency, both economically (i.e. Gflops/$) and power-wise (i.e. Gflops/W), although the most recent multicore CPUs are improving in this respect. See Table 3.4 for a comparison of different modern CPUs and GPUs.

Processor	Cores	Gflops	GB/s	Gflops/W	Gflops/$
Intel Core i7 5960X	8	384 (768)	68	2.7 (5.5)	0.38 (0.77)
Intel Xeon E7-2890v2	15	336 (672)	85	2.2 (4.3)	0.052 (0.1)
Intel Xeon E5-2699v3	18	662 (1 325)	68	4.6 (9.1)	0.16 (0.32)
Nvidia Tesla M2090	512	666 (1 332)	177	3.0 (5.9)	0.17 (0.33)
Nvidia Tesla K20X	2 688	1 310 (3 935)	250	5.6 (16.7)	0.34 (1.0)
Nvidia Tesla K40	2 880	1 429 (4 291)	288	6.1 (18.3)	0.26 (0.78)
Nvidia Tesla K80	4 992	1 868 (5 611)	480	6.2 (18.7)	0.37 (1.1)
AMD FirePro S9150	2 816	2 534 (5 069)	320	10.8 (21.6)	0.77 (1.5)
Intel Xeon Phi 7120P	61	1 208 (2 417)	352	4.0 (8.1)	0.29 (0.59)

Table 3.4: *Comparison of various processing units. All numbers are peak values and taken from the manufacturers' specifications. The numbers in parentheses are for single precision, and the others are for double precision.*

Scientific applications, such as e.g. stencil operations or matrix-matrix multiplications, are usually comprised of many small and similar tasks with a high computational intensity. Because of this, the fit for the throughput-optimized high-bandwidth GPU hardware has in many cases been great. However, several limitations of the graphics-tailored GPU architecture limit how well applications can take advantage of the available performance potential of GPUs. For instance, few applications possess the amount of parallelism needed to saturate the massively parallel GPUs. In addition, for most scientific applications, double precision is necessary to obtain meaningful results, which, as mentioned, has a performance penalty over single precision. Furthermore, while dependencies and synchronization are unavoidable parts of most algorithms, these are often very difficult or even impossible to resolve on GPUs. Thus, in order to fully utilize GPUs, it is often necessary to make substantial changes to existing algorithms, or even invent new ones, which take these limitations into account. Another issue is that data has to be moved to the graphics memory before it can be accessed by the GPU, which is presently done by transferring the data over the relatively slow PCI bus. To avoid this bottleneck, data is preferably kept at the GPU for the entire computation. Another approach is to hide the latency by overlapping computation and data transfer.

The history of general-purpose graphics-processing unit (GPGPU) computations started around 2000 when dedicated graphics cards were becoming mainstream. In the beginning, the general-purpose computations had to be shoehorned into the graphics programming pipeline by storing the data as textures and putting the computations in the programmable vertex and pixel shaders. Examples of early successful general-purpose computations on graphics hardware are matrix-matrix multiplication [40], a solution of the compressible Navier-Stokes equations [41], and a multigrid solver [3]. A summary of early work in GPGPU can be found in the survey paper by Owens et al. [42].

However, the many restrictions and the fact that a programming model for graphics had to be exploited made it a daunting task to do general-purpose computations with the graphics pipeline. In response to this, at the end of 2006, Nvidia released CUDA, *Compute Unified Device Architecture*, which simplified the programming and led to a dramatic increase in interest for GPGPU.

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 multi core processors capable of executing a large number of threads concurrently. The threads of the application are then grouped into blocks of threads and each block is executed on a single SM, independently of the other blocks. Within a thread block or an SM, there is a piece of shared memory. Furthermore, it is possible to have synchronization between the threads of a single block, but it is not possible to synchronize threads across blocks, except for a global barrier. There is also a small cache shared between the threads of a block.

An important feature of CUDA, and arguably the most crucial aspect to attain good utilization of the hardware, is the memory model. Because transfers from the main device memory are only made in chunks of a certain size, and due to poor caching capabilities, it is important to use all the data of the chunks which are fetched. When the threads within a block access a contiguous piece of memory simultaneously, such a *coalesced* memory access is achieved. For further details on the CUDA platform, see the CUDA C Programming Guide [43]. Examples of fields where CUDA has been successfully utilized include molecular dynamics simulations [44], fluid dynamics [45], wave propagation [46], sequence alignment [47] and, Monte Carlo simulations of ferromagnetic lattices [48].

In response to CUDA and the popularity of GPU programming, OpenCL was launched by the consortium Khronos Group in 2008 [49]. In many respects, such as the hardware model and the programming language, OpenCL and CUDA are very similar. However, in contrast to CUDA, which is proprietary and restricted to Nvidia GPUs, OpenCL is an open standard, and OpenCL code can be run on all hardware with an OpenCL implementation; today including Nvidia and AMD GPUs, and even Intel and AMD CPUs. While the same OpenCL code is portable across a wide range of platforms, it is usually necessary to hand tune the code to achieve the best performance. In addition, CUDA, being made by Nvidia specifically for their GPUs, is still able to outperform OpenCL in comparisons and when optimal performance is desirable, CUDA is still the natural choice [50, 51].

Critique has been raised as to the long-term viability of techniques and codes developed for GPUs in general and CUDA in particular, since these are very specific concepts which might have a fairly limited life time. However, an important point is that GPUs are part of a larger movement – that of heterogeneity and increasing use of specialized hardware and accelerators. Recently, all the major processor vendors have started offering dedicated accelerators for computations, which, in addition to the Tesla GPUs of Nvidia, include Intel's Xeon Phi co-processor and the very recently announced FirePro cards by AMD (see Table 3.4). Since most of these accelerators share a similar throughput-oriented architecture, once an algorithm has been designed for one of them it is not very difficult to convert to the others. Therefore, developing algorithms and techniques for dedicated accelerators, such as GPUs, is relevant also for the technology of the future.

3.5.1 Finite-Element Methods on GPUs

Due to the higher complexity of the assembly phase, early attempts at leveraging GPUs for finite-element computations focused on speeding up the solve phase [3, 52, 53, 54]. Since in matrix-based finite-element software, the solve phase is based on a general sparse matrix-vector product, these can readily take advantage of sparse linear algebra libraries for GPUs. Examples of such libraries include CUSPARSE, an Nvidia library for sparse computations included in CUDA since 2010 [55]; and PARALUTION, a library for sparse linear algebra targeting modern processors and accelerators including GPUs [56], available since 2012.

On the other hand, as explained in Section 2.2, the assembly is a much more complicated operation requiring explicit implementation to be performed on the GPU. In [4], Cecka et al. explore different techniques for performing the assembly on GPUs, using both element-wise and node-wise parallelization. Dziekonski et al. propose a GPU-based implementation of the assembly with computational electrodynamics in mind [57]. Markall et al. study what implementations of FEM assembly are appropriate for many-core processors such as GPUs compared to multi-core CPUs [58].

A number of studies have considered finite-element solution of hyperbolic problems on GPU, which often take on a matrix-free approach. However, we are not aware of existing matrix-free work for the GPU targeting elliptic and parabolic PDEs. In [16], Komatitsch et al. study a high-order spectral-element method applied to a seismic simulation on a single GPU. In

[59], the authors expand this to a cluster of GPUs using MPI. While using the spectral-element method, basically a high-order finite-element method, in the earthquake application considered, the partial differential equations are hyperbolic and an explicit time stepping can be used. Together with the Gauss-Lobatto-Legendre integration scheme, which yields a diagonal mass matrix, this means that no linear system is solved during the simulation, although the matrix-free operator application still constitute the most important operation.

Klöckner et al. investigate a GPU-parallelization of a discontinuous Galerkin (DG) method for hyperbolic conservation laws [60]. Due to properties of the DG method, the mass matrix is block diagonal, which means that the system can easily be solved element-wise, removing the need for an iterative solve phase.

3.5.2 Matrix-Free FEM Experiments

In Paper II, we study the performance of a GPU-based implementation of the matrix-free finite-element operator application described in Section 2.3, for future inclusion in a FEM solver of parabolic and elliptic PDEs. This is motivated by the improved computational intensity of this approach, which has the potential of letting us take advantage of immense the performance offered by the GPUs.

To do this, we apply the matrix-free method to a Poisson problem similar to the one in Section 2.1. We consider unit square or cube discretized by a uniform Cartesian mesh, and elements of polynomial order one to four, in 2D and 3D. While being a quite specific and simplified case, it still constitutes an interesting problem since more advanced non-linear or time-dependent equations are treated in a very similar way, and since a uniform logically Cartesian mesh is often used in major parts of the domain. Since the most important operation in the solve phase is the application of the matrix-free operator, corresponding to a sparse matrix-vector product, we only consider this operation, rather than solving an actual problem.

In our experiments, we include a number of different versions of the stiffness operator. We have implemented these in a small C++/CUDA framework for matrix-free FEM computations. The implementations for the CPU use OpenMP for the parallelization, whereas the ones for the GPU use CUDA. We use double precision in all our computations, since high numerical accuracy is necessary for scientific applications.

Matrix-Free Versions

Six different matrix-free implementations of the stiffness operator are included; three for the CPU and three for the GPU, with the main difference being mainly how the conflicting updates are handled. Firstly, there is a serial version, **`Mfree`**, which simply performs all the elements on a single thread, in which case no protection from conflicting updates is needed. Secondly, the version **`Color`** utilizes the mesh coloring strategy described in Section 2.3.2, to process the elements in conflict-free chunks. Finally, there is a version **`PrivateBuffers`** which utilizes privatization in the same way as was used in the N-body experiments (see Section 3.3.4). By creating a separate result vector for each thread, they can safely perform their updates without risk for race conditions. Eventually, the multiple vectors are summed up in parallel in a reduction phase.

For the GPU, there is also one version, referred to as **`GPU_Color`**, which uses mesh coloring. In the **`GPU_Atomic`** version, we use atomic operations much like the compare-and-swap technique in the transactional-memory experiments (see Section 3.3). In CUDA, these are supported as intrinsic functions. At last, we include a third version **`GPU_Max`** which processes the elements all at once much like the version using atomics, but performing the updates without any protection at all. This of course does not yield correct results, but is included as an reference of what performance to expect with perfect treatment of conflicts (i.e. no overhead). Due to the very large number of threads used by the GPU, the privatization approach used on the CPU is infeasible on the GPU.

Matrix-Based Versions

For further comparison, we include matrix-based implementations of stiffness operator for both the CPU and the GPU, which use a *compressed sparse row* (CSR) matrix format, since this is efficient for sparse matrix-vector products and other row-wise operations. In this format, the matrix is stored in three arrays: **`val`**, which stores the non-zero entries of the matrix in row order; **`col_ind`**, which stores the corresponding column indices; and **`row_ptr`**, which stores the index in the **`val`** array of the first entry of each row.

For the GPU, we also tried using the hybrid (HYB) format [61]. However, we found that this did not perform better than the CSR format for our case. The reason for this is probably that the HYB format is intended for matrices with very different number of nonzeros per row, whereas our matrix has pretty decently balanced rows.

For dynamic construction, the CSR format performs extremely poorly, since, for each element added, all subsequent entries in the arrays must be moved. Instead, we perform the assembly a sparse matrix of the list-of-lists format (LIL), where each matrix row is stored in a separate STL `vector`. This improves performance of dynamic construction, while still allowing for fast conversion to the CSR format.

The CPU version is implemented manually using OpenMP, and parallelized by letting each thread compute a number of consecutive rows of the result. The GPU version on the other hand uses the efficient SpMV routine from the CUSPARSE, a sparse linear algebra library for CUDA [55]. In both cases, the same CPU code is used for the assembly, and in the GPU case, we simply copy the matrix into the GPU memory.

Experimental Setup

The experiment consists of:

(a) Setting up the data structures

(b) Transferring data for the vector of unknowns to the GPU (if applicable)

(c) Performing a number of successive operator applications

(d) Transferring the result vector back to the main memory (if applicable)

In our time measurements, we include step (b) and (d) to account for the transfer of data over the PCI bus. We perform 20 applications in step (c), which is a low number; in practice one might need orders of magnitude more, depending on the problem and the use of preconditioners. The numbers reported are normalized to the number of operator applications, i.e. divided by 20. To get solid results unaffected by statistical dispersion, we report the minimum time from 20 identical runs of the entire experiment. Finally, we also measure the time for step (a) separately, to quantify the benefit of not having to assemble a matrix.

All experiments are performed on a dedicated system with an eight-core Intel Xeon E5-2680, 64 GB DRAM, and an Nvidia Tesla K20c. The system runs Linux 2.6.32, with GCC 4.4 and CUDA 5.5. All manually implemented CUDA versions use 256 threads per block, while for the version using CUSPARSE, this was not possible to specify. All the OpenMP-parallel CPU versions use four threads since this was found to be the fastest.

Results

Looking at the setup times in Table 3.5, we see that the matrix-free versions need many orders of magnitude less time than the matrix-based counterparts. The difference between the matrix-based version for the CPU and the GPU is due to the transfer of the whole sparse matrix to the GPU memory. We note that, while the setup time for the matrix-based versions grew with element complexity, for matrix-free version setup time actually decreases instead. This is simply because we consider problems with a fixed number DoFs, meaning that the number of elements decreases with the element order. For a given number of elements, we expect this number to be constant.

	2D			3D		
Element order	1	2	4	1	2	4
`SpM`	6.2	8	14	8.4	18	66
`Color`	0.33	0.25	0.18	0.48	0.24	0.075
`GPU_SpM`	7.9	10	18	12	23	-
`GPU_Color`	0.4	0.17	0.1	0.43	0.15	0.1

Table 3.5: *Setup time in seconds of different operator implementations (for 4.1 and 6.6 million DoFs in 2D and 3D respectively). A fail due to insufficient memory is indicated by "-".*

In Figures 3.6 and 3.7, we see the performance of all versions for the largest problems, in 2D and 3D, respectively. Here, the performance numbers are calculated as the number of useful operations divided by the run time of a single operator application. In other words, this does not include the extra computations performed by the matrix-free approach.

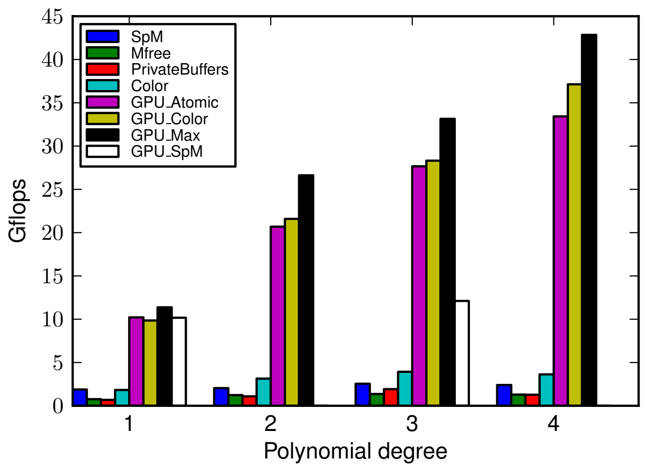


Figure 3.6: *Performance for the largest problems solved in 2D. The problem size was 26 million DoFs for elements of order one, two, and four; and 15 million DoFs for third-order elements.*

First of all, we see the tremendous computing power of the GPU. For both the matrix-free and matrix-based algorithms, the GPU is 4-10 times faster than the CPU. Furthermore, this difference increases as the element order is

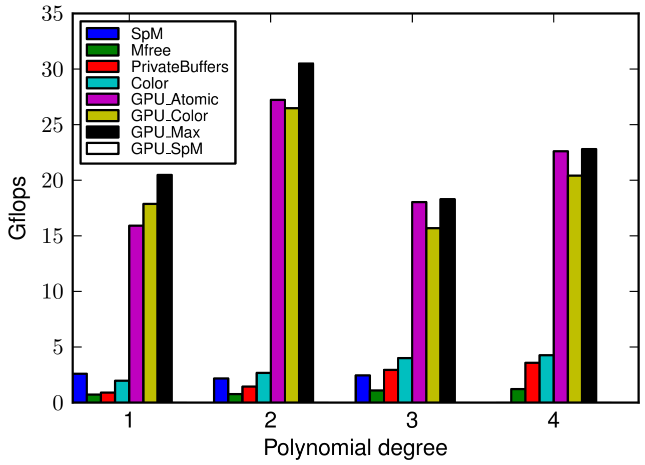


Figure 3.7: *Performance for the largest 3D problems. The problem size was 33 million DoFs for elements of order one, two and four; and 14 million DoFs for third-order elements.*

increased; the number of computations per second increases strongly for the matrix-free GPU-based, whereas this tendency is much weaker for both the matrix-based GPU version, and all the CPU versions as well. However, this pattern is broken for third and fourth order elements in 3D, where we see a large drop in the performance. This is caused by the local matrix becoming too large for the L1 cache, which for the Tesla K20 is equal to 32 kB, or 4096 doubles. The local matrix consists of $(p+1)^{2d}$ doubles, and is listed in Table 3.6 for all the configurations included in the experiments. There, we see that exactly for $p = 3$, the local matrix plus various other smaller data structures such as element indices mappings will no longer fit in the cache. This means that the threads in a block can no longer share the local matrix in the cache and neither will it remain there when it is needed again, thus no longer taking advantage of the spatial and temporal locality, causing a decrease in performance utilization. Yet, there is still a large speed up relative to the CPU-based versions.

	p			
	1	2	3	4
2D	16	81	256	625
3D	64	729	4096	15625

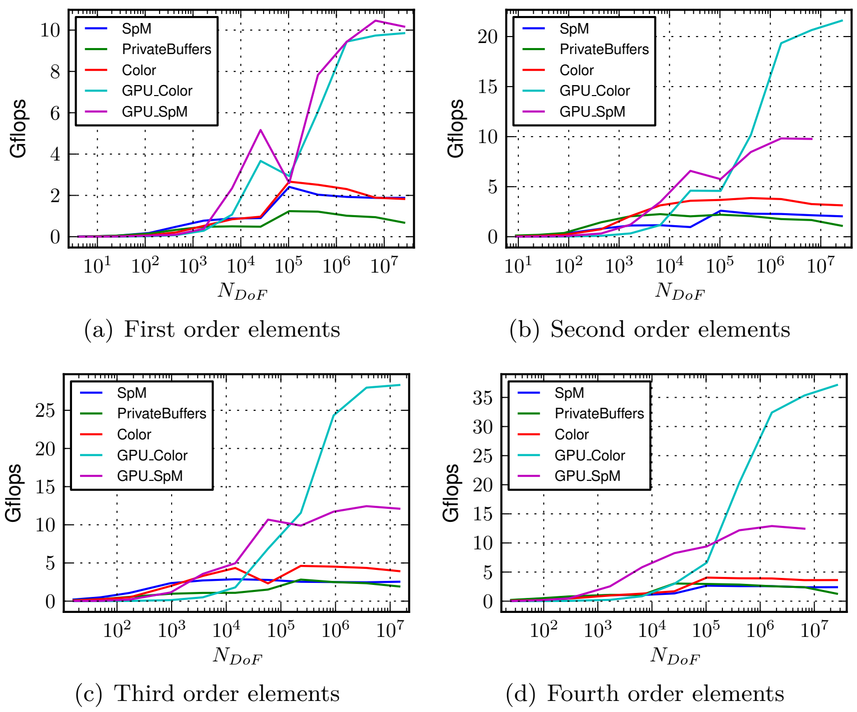
Table 3.6: *Size of local matrix for various parameter values (in doubles).*

Looking at the performance of the two different approaches to handling conflicts, atomic instructions and mesh coloring, we see that both of these had a slight overhead compared to the reference version (`GPU_Max`). In 2D, the overhead is always smaller for the coloring approach except for elements of order one, where the two approaches perform roughly the same. In 3D, the coloring approach was fastest for elements of order one, whereas for second-order elements, the overheads were once again about the same. For the two configurations which are affected by the caching problem described above, i.e. third- and fourth-order elements in 3D, this pattern is not repeated. Instead we see that atomic operations have a negligible overhead whereas there is still a slight overhead for the coloring approach. The explanation for this is that the slower execution hides the overhead of the atomic operations by making the updates less frequent. The coloring approach on the other still has to process the colors separately, and thus still have roughly the same overhead as for the other configurations.

If we look at the results on the CPU, we see that, for all element order higher than one, the matrix free algorithm was faster than the one based on a sparse matrix. Also on the GPU, the matrix-free versions outperformed the one using a sparse matrix for all but first-order elements. However, note that many of the entries for the matrix-based version are missing from Figures 3.6 and 3.7. This is due the fact that the sparse matrix can amount to a substantial size, which in this case could not be fit in the available memory. We see this problem also for the CPU, but it is most severe for the GPU, which is limited to matrices smaller than 5 GB. This can be seen in greater detail in Figures 3.8 and 3.9, which show how performance scales as we increase the number of DoFs for different element orders, in 2D and 3D, respectively. Here, we see that for all but a few cases in 2D, the curves are truncated because of insufficient memory.

Since the matrix-free algorithm is parallelized over the elements, enough elements are needed for the system to be fully utilized. This is seen by the curves for the matrix-free versions which first stays relatively constant at a low level, followed by a rapid increase when the number of elements becomes large enough. Furthermore, we see that this inflection point is shifted slightly to the right as element complexity increases. For the matrix-based versions, since these are parallelized over the rows, i.e. over the DoFs, we do not see this tendency. Rather, we see that enough parallelism is available for smaller problems, in particular when elements are complex. This can be seen especially from the 3D results.

Finally, the implementation based on mesh coloring was the fastest version for the CPU, except for first-order elements where the matrix-based version performed the best, in both two and three dimensions. Also, for the CPU implementation using privatization (**PrivateBuffers**), we observe that, while performing decently for medium sized problems, there is a decrease in performance for the largest problems, probably due to the overhead in terms



(a) First order elements

(b) Second order elements

(c) Third order elements

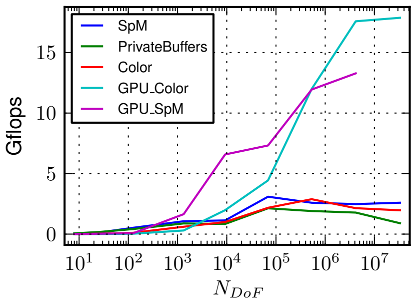
(d) Fourth order elements

Figure 3.8: *Performance versus problem size for the 2D experiments.*

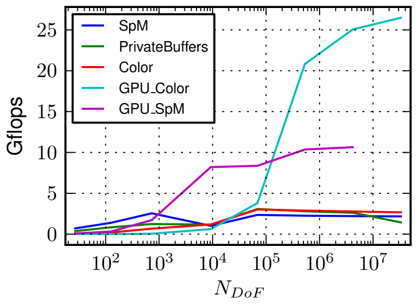
of memory and the reduction phase.

Conclusion

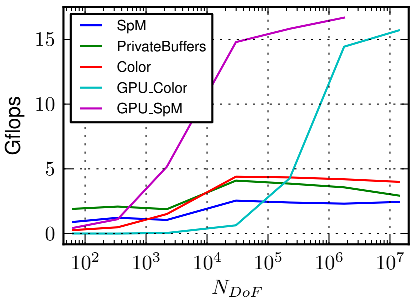
In summary, using the matrix-free method, we were able to take advantage of the computational power of the GPU, while still being able to solve much larger problems than the version using a sparse matrix. This, together with the fact that the assembly can be avoided, shows that the matrix-free method should be considered for solving realistic problems involving non-linearities and large meshes in 3D. Finally, we saw that for element types which result too large local matrices, our local-matrix approach performed much worse due to data no longer fitting in cache. This indicates that a tensor-based approach can pay off also for problems with a constant local matrix, i.e. linear problems on a uniform mesh.



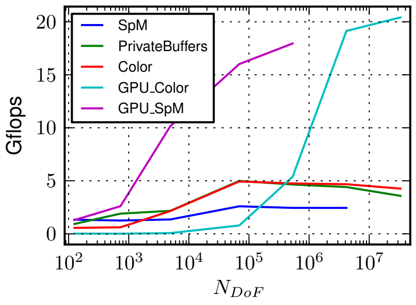
(a) First order elements



(b) Second order elements



(c) Third order elements



(d) Fourth order elements

Figure 3.9: *Performance versus problem size for the 3D experiments.*

Chapter 4

Outlook

4.1 Ongoing and Future Work

Currently, we are working on implementing the tensor-based approach described in Section 2.3.3, to be able to solve problems on general meshes or problems involving non-linear PDEs. In addition to this, we saw in Section 3.5.2 that the local matrix implementation used in Paper II did not perform well when the local matrix did not fit in the L1 cache. The tensor-product technique on the other hand does use more distinct memory in total (see Figure 2.3), but since each thread reads less data, it may perform better for the problematic configurations, i.e. for high element order in 3D.

We plan on including the tensor-based operator and the present functionality in the finite-element library `deal.II`, to facilitate the spreading of the code for usage by researchers within application fields [12, 13]. In addition, this also opens up for the possibility of leveraging existing functionality such as treatment of general geometries, usage of adaptive mesh refinement, and solution of more general possibly coupled equations. Ultimately, this will let us use our GPU implementation to speed up solution of realistic application problems such as multiphase flow, where solving physically realistic problems in 3D is not feasible today.

Also, we are looking at alternative approaches to the parallelization including the node-wise approach suggested by Cecka et al. [4]. Since our implementation is still memory bound, recomputing the element integrals may pay off if a better memory access pattern can be achieved. In addition, the conflicting writes are avoided.

Another related approach which we are investigating employs tiling by letting a thread block process a connected set of elements. If the DoFs within such an element tile are ordered in a consecutive manner, these can be read collectively by the thread block, for a good access pattern. The DoFs inside the tile are not shared with other tiles and can thus be ordered in an optimal manner, whereas the DoFs on the tile boundary have to be treated specially since they are shared with neighboring tiles. It is therefore desirable with a high ratio of internal elements to elements on the boundary, implying that tiles should be as close to cubic as possible. In practice, the partitioning is a trade off between the shape of tiles and a balance in the size of the tiles. Apart from the benefit of efficient reading of the DoFs, this has the additional advantage of allowing for many of the shared updates, namely the tile-internal ones, to be done using atomic operations in the CUDA shared memory. Alternatively, a manual merging strategy can be used, since threads within a block can be synchronized conveniently.

Finally, motivated by the large difference in compute power between single-precision and double-precision operations of many graphics processors, we also plan on investigating if some of the computations can be performed in single precision without sacrificing overall numerical accuracy. This is especially interesting due to the iterative nature of the linear solver in which one can allow for extra iterations if a single iteration can be performed faster, and has been shown to perform well [52, 62].

Bibliography

[1] M. G. Larson and F. Bengzon, *The Finite Element Method: Theory, Implementation, and Applications*, vol. 10 of *Texts in Computational Science and Engineering.* Berlin: Springer, 2013.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems.* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2nd ed., 2003.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Transactions on Graphics*, vol. 22, pp. 917–924, JUL 2003.

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

[5] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, 4th ed., 2009.

[6] D. A. Knoll and D. E. Keyes, "Jacobian-free Newton-Krylov methods: a survey of approaches and applications," *Journal of Computational Physics*, vol. 193, pp. 357–397, JAN 20 2004.

[7] D. Komatitsch and J. Tromp, "Introduction to the spectral element method for three-dimensional seismic wave propagation," *Geophysical Journal International*, vol. 139, pp. 806–822, DEC 1999.

[8] S. A. Orszag, "Spectral methods for problems in complex geometries ," *Journal of Computational Physics* , vol. 37, no. 1, pp. 70–92, 1980.

[9] J. M. Melenk, K. Gerdes, and C. Schwab, "Fully discrete hp-finite elements: fast quadrature ," *Computer Methods in Applied Mechanics and Engineering* , vol. 190, no. 32–33, pp. 4339–4364, 2001.

[10] C. D. Cantwell, S. J. Sherwin, R. M. Kirby, and P. H. J. Kelly, "From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements," *Computers & Fluids*, vol. 43, no. 1, SI, pp. 23–28, 2011.

[11] M. Kronbichler and K. Kormann, "A generic interface for parallel cell-based finite element operator application," *Computers & Fluids*, vol. 63, no. 0, pp. 135–147, 2012.

[12] W. Bangerth, R. Hartmann, and G. Kanschat, "deal. II – A General Purpose Object-Oriented Finite Element Library," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, pp. 24/1–24/27, 2007.

[13] W. Bangerth, T. Heister, G. Kanschat, *et al.*, `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[14] P. Berger, P. Brouaye, and J. C. Syre, "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.

[15] C. Farhat and L. Crivelli, "A General-Approach to Nonlinear Fe Computations on Shared-Memory Multiprocessors," *Computer Methods in Applied Mechanics and Engineering*, vol. 72, no. 2, pp. 153–171, 1989.

[16] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451–460, 2009.

[17] "SuperGlue – A C++ Library for Data-Dependent Task Parallelism." `http://tillenius.github.io/superglue/`. Accessed: 2014-12-19.

[18] M. Tillenius, "SuperGlue: A shared memory framework using data-versioning for dependency-aware task-based parallelization," Tech. Rep. 2014-010, Department of Information Technology, Uppsala University, Apr. 2014.

[19] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, April 1965.

[20] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, pp. 256–268, Oct 1974.

[21] M. Herlihy and J. E. B. Moss, “Transactional memory: architectural support for lock-free data structures,” *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, 1993.

[22] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded Transactional Memory,” in *HPCA ’05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, (Washington, DC, USA), pp. 316–327, IEEE Computer Society, 2005.

[23] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, “Architectural Support for Software Transactional Memory,” in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 185–196, IEEE Computer Society, 2006.

[24] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, “Performance pathologies in Hardware Transactional Memory,” in *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, pp. 81–91, 2007.

[25] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski, “Anatomy of a scalable software transactional memory,” in *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2009.

[26] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, “Rock: A High-Performance SPARC CMT Processor,” *IEEE Micro*, vol. 29, pp. 6–16, MAR-APR 2009.

[27] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, G. L. T. Chiu, P. A. Boyle, N. H. Chist, and C. Kim, “The IBM Blue Gene/Q Compute Chip,” *Micro, IEEE*, vol. 32, pp. 48–60, March 2012.

[28] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture*, September 2014.

[29] M. Kunaseth, R. K. Kalia, A. Nakano, P. Vashishta, D. F. Richards, and J. N. Glosli, “Performance Characteristics of Hardware Transactional Memory for Molecular Dynamics Application on BlueGene/Q: Toward Efficient Multithreading Strategies for Large-Scale Scientific Applications,” in *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pp. 1326–35, 2013.

[30] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel reg Transactional Synchronization Extensions for high-performance computing," in *2013 SC - International Conference for High-Performance Computing, Networking, Storage and Analysis*, p. 11, 2013.

[31] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 157–168, ACM, 2009.

[32] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early experience with a commercial hardware transactional memory implementation," Tech. Rep. TR-2009-180, Sun Laboratories, 2009.

[33] T. Y. Hou and X.-H. Wu, "A Multiscale Finite Element Method for Elliptic Problems in Composite Materials and Porous Media ," *Journal of Computational Physics* , vol. 134, no. 1, pp. 169–189, 1997.

[34] G. Efstathiou, M. Davis, C. Frenk, and S. White, "Numerical Techniques for Large Cosmological N-Body Simulations," *Astrophysical Journal Supplement Series*, vol. 57, no. 2, pp. 241–260, 1985.

[35] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular-Dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, MAR 1 1995.

[36] A. G. Gray and A. W. Moore, "'N-body' Problems in Statistical Learning," in *Advances in Neural Information Processing Systems 13*, pp. 521–527, 2001.

[37] J. Barnes and P. Hut, "A Hierarchical $O(N \log N)$ Force-Calculation Algorithm," *Nature*, vol. 324, pp. 446–449, Dec 1986.

[38] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations," *Journal of Computational Physics*, vol. 73, pp. 325–348, DEC 1987.

[39] K. Rupp, "CPU, GPU and MIC Hardware Characteristics over Time." `http://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`, 2013.

[40] E. S. Larsen and D. McAllister, "Fast Matrix Multiplies Using Graphics Hardware," in *Supercomputing, ACM/IEEE 2001 Conference*, pp. 43–43, 2001.

[41] J. Kruger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics*, vol. 22, pp. 908–916, JUL 2003.

[42] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," in *Eurographics 2005, State of the Art Reports*, pp. 21–51, 2005.

[43] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, July 2013. Version 5.5.

[44] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[45] E. Elsen, P. LeGresley, and E. Darve, "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, vol. 227, no. 24, pp. 10148–10161, 2008.

[46] D. Michéa and D. Komatitsch, "Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards," *Geophysical Journal International*, vol. 182, no. 1, pp. 389–402, 2010.

[47] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment," *BMC Bioinformatics*, vol. 9, no. 2, 2008. Annual Meeting of the Italian-Society-of-Bioinformatics, Naples, ITALY, APR 26-28, 2007.

[48] T. Preis, P. Virnau, W. Paul, and J. J. Schneider, "GPU accelerated Monte Carlo simulation of the 2D and 3D Ising model," *Journal of Computational Physics*, vol. 228, pp. 4468–4477, JUL 1 2009.

[49] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, pp. 66–72, MAY-JUN 2010.

[50] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *ArXiv e-prints*, may 2010.

[51] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Parallel Processing (ICPP), 2011 International Conference on*, pp. 216–225, Sept 2011.

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

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

[54] A. Dziekonski, A. Lamecki, and M. Mrozowski, "A Memory Efficient and Fast Sparse Matrix Vector Product on a GPU," *Progress in Electromagnetics Research-Pier*, vol. 116, pp. 49–63, 2011.

[55] NVIDIA Corporation, *CUDA CUSPARSE Library*, July 2013.

[56] PARALUTION, *User Manual.* Version 0.8.0, November 2014.

[57] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Finite Element Matrix Generation on a GPU," *Progress in Electromagnetics Research-Pier*, vol. 128, pp. 249–265, 2012.

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

[59] D. Komatitsch, G. Erlebacher, D. Göddeke, and D. Michéa, "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster," *Journal of Computatinal Physics*, vol. 229, no. 20, pp. 7692–7714, 2010.

[60] A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven, "Nodal discontinuous Galerkin methods on graphics processors," *Journal of Computational Physics*, vol. 228, no. 21, pp. 7863–7882, 2009.

[61] N. Bell and M. Garland, "Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 18–1, ACM, 2009.

[62] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, "Using GPUs to improve multigrid solver performance on a cluster," *International Journal of Computational Science and Engineering*, vol. 4, pp. 36–55, 2008 2008.

Paper I

2011 IEEE International Parallel & Distributed Processing Symposium

Using Hardware Transactional Memory for High-Performance Computing

Karl Ljungkvist*, Martin Tillenius*, David Black-Schaffer*, Sverker Holmgren*, Martin Karlsson*, Elisabeth Larsson*
*Department of Information Technology
Uppsala University
⟨firstname⟩.⟨lastname⟩@it.uu.se

***Abstract*—This work investigates the benefits of transactional memory for high-performance and scientific computing by examining n-body and unstructured mesh applications on a prototype computer system with transactional memory support in hardware. Transactional memory systems have the potential to both improve performance, through reduced lock overhead, and ease development, by providing simplified synchronization semantics. To date, only a few early experiments have been presented on actual transactional memory hardware, with none of them investigating the benefits of transactional memory for scientific computing.**

We investigate straight-forward implementations to see if replacing locks by transactions can yield better performance and compete with more complex algorithms. Our experiments show that using transactions is the fastest way to concurrently update shared floating-point variables, which is of interest in many scientific computing applications. However, if it is possible, avoiding concurrent updates altogether yields the best performance.

***Keywords*-transactional memory; performance; high performance computing;**

I. Introduction

When using shared memory for parallel programming communication is trivial, since each thread can access the memory used by all other threads. However, because of the concurrent execution, this also means that race conditions may occur with unpredictable results when several threads access the same shared resource. To resolve this, such critical sections of the program must be executed mutually exclusively.

The standard way to achieve the required exclusivity is to use locks, which allow the programmer to enforce exclusive access to the critical sections of the code by only one thread at a time. To use locks it is necessary to explicitly define the critical sections in the code. Even simple uses of locks can cause difficult to diagnose problems, such as deadlock, lock convoying, and priority inversion [1]. Lock convoying occurs when a thread holding a lock gets preempted, preventing other threads that are waiting for the lock from running until the preempted thread is rescheduled. Priority inversion occurs when a high-priority thread is prevented from running by a low-priority thread holding a lock.

In addition to the difficulties of correctly using locks, acquiring and releasing them incurs an overhead from accessing the lock variable, whether or not there is any contention for the lock. If the amount of work to be done inside the lock is small, this overhead can be significant. Because the lock variable is checked before executing the code, a lock can be considered pessimistic, in that it does not speculatively assume it can execute.

An alternative approach to achieve exclusivity is to group operations that need to be executed mutually exclusively together into an *atomic transaction*. Such transactions can either commit entirely (succeed) or abort entirely (fail). If a transaction commits, all changes made to memory by the code within the transaction are made permanent and visible to the other threads. If a transaction aborts, all changes are rolled back and the pre-transactional state of the system is recalled. As all changes show up to other threads either all at once or not at all, the transaction is atomic.

With a transactional model, instead of acquiring locks that guard the critical sections, the critical sections are executed inside transactions, and conflicts between threads are handled when they occur by aborting at least one of the conflicting transactions. The result of this is that only one thread can successfully execute a critical section (transaction) at a given time, thereby achieving the required exclusivity. Transactions can therefore be considered to be an optimistic approach because they execute first and evaluate whether they may commit afterwards. This is made possible by the ability to roll back the changes made during the failed transaction.

Unlike locks, transactions can avoid the overhead of accessing lock variables for each transaction. This has the potential to result in better performance for very fine grained locking approaches where the amount of computation done for each lock is small. Equally important for the performance is the fact transactions are allowed to execute simultaneously inside the critical section as long they do not touch the same data.

In addition to the technical differences, transactions have the potential to provide a simpler programming model for parallel systems. With transactions, the programmer need only reason about what portions of the application must execute exclusively, and not about how to build an efficient locking structure that ensures that behavior. If transactional programming is sufficiently simple, and still provides acceptable performance, it could have a very significant impact on the field of high-performance scientific computing. This paper seeks to investigate these two issues of performance and simplicity on a prototype hardware transactional memory system. To answer these questions we first investigate the overhead and potential of transactions compared to locks through a pair of micro-benchmarks. From there we apply our experience to two representative scientific applications and compare their performance to lock-based versions. Throughout these experiments we further address some of the realities of programming on a (prototype) hardware transactional memory system, and describe the difficulties we encountered.

II. Transactional Memory and the Test System

A transactional memory (TM) system provides atomicity through transactions, which are code sections monitored for conflicting memory accesses. For instance, if transaction A writes to a memory location from which transaction B has read, the TM system will abort at least one of A and B. For the programmer, this means that the piece of code declared to be transactional will be executed atomically, or not at all.

Transactional memory can be implemented in software, but with support in hardware it has potential to be efficient enough for high-performance computing. In 1993 Herlihy and Moss showed that Hardware Transactional Memory can be implemented as a modified cache coherence protocol with very successful simulation results [1].

The study presented in this paper is performed on a rather unique prototype system with support for hardware transactional memory [2]. The application interface for transactions consists of two instructions: one that starts a transaction and one that commits it. Access to these is provided through intrinsic functions in the prototype compiler or directly in assembly language. Transaction status codes are returned through a special register for transaction failures. The system has a single 16-core processor where four cores share a 512KB L2 cache and a total of 128 GB of system RAM. More details can be found in [2].

Early experiences with this system were reported in [3], where they emphasized that transactions may fail for many other reasons than just conflicting accesses. These include transactions that exceed the available hardware storage, system interrupts during a transaction, too many outstanding instructions during a transaction, and misses in the TLB and caches, which are not serviced during the transaction. We encountered similar failures during our experiments, with a few particular failure causes occurring most frequently. We therefore grouped our transaction failures into the following four classes; **coherence:** for transactions failing due to conflicting memory accesses, **load:** for failures due to problems reading memory, **store:** for failures when writing to memory, and **other:** for all other problems. A much more thorough investigation of the error codes and their meanings is given in [4].

III. Experimental Methodology

All parallel implementations were done using the pthreads library. With locks, we refer to pthread mutexes. We have also performed tests using the pthread spinlock which gave similar but slightly worse results in all tests and is excluded here for clarity. To avoid false sharing, each lock was assigned its own cache line.

On SPARC systems, the common way to achieve atomic updates is to use the compare-and-swap instruction. Using this method, it is only possible to update a single value atomically, in contrast to using locks or transactions. For our applications, this is not a restriction. A limitation is that the compare-and-swap instruction only works on integer registers, while we want to update floating point values. This means that values must be moved back and forth between floating point registers and integer registers. To do this, the values have to be stored and reloaded, which is quite expensive.

IV. Transaction Overhead

The overhead of acquiring and releasing a lock can be significant for code with the fine-grained locking necessary to maximize the parallelism in many applications. Transactions have the potential to avoid this overhead by eliminating the need to access the lock variable when acquiring and releasing the lock. To evaluate these overheads, we wrote a micro-benchmark that increments a small array of floating point numbers. We compared the performance with the array accesses inside a transaction, surrounded by a lock, using compare-and-swap, and with no synchronization at all. The size of the array was varied to determine if more work would amortize the overhead of the synchronization primitives and result in higher throughput. All tests were running on a single thread to avoid any true concurrency issues, thereby allowing us to assess the pure overhead of these approaches. To further isolate the overhead, the data is prefetched into the cache before the test begins. The maximum size of the array was limited by the hardware to 8 elements or less for the transactions.

A. Results

The results of the micro-benchmark are shown in Figure 1 and in Table I. In Figure 1, the lines are drawn thicker to cover the upper and lower quartiles to show the statistic dispersion. As can be seen by the only slightly thicker lines, the variation is small.

On the prototype system, a transaction is roughly three times as fast as using a lock. This shows that the *non-contention overhead* of a transaction is significantly lower than that of a lock. However, real lock-based applications are likely to see an additional penalty from the increased bandwidth of accessing multiple lock variables, that will not be present in a similar transaction-based program.

The compare-and-swap version can only protect a single element, so updates with more than one element must be done by updating each element separately. This means that this operation is not atomic in the same sense as the other methods. In our case, we only need element-wise atomicity so this poses no problem, but makes the comparison unfair. Because of this, we only show the measurement value for one update in Figure 1 and Table I.

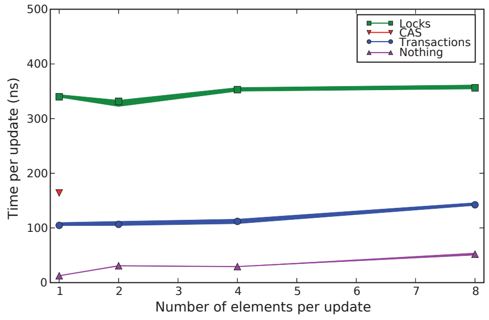


Figure 1. Overhead: Time per update (ns). The width of the lines represent the variation of the results from the lower to the upper quartile.

Table I
OVERHEAD: MEDIAN TIME PER UPDATE (ns)

	Number of elements per update			
	1	2	4	8
Nothing	12.5	30.8	29.3	52.0
Transactions	104.7	106.5	112.0	142.3
CAS	164.1	-	-	-
Locks	339.9	331.5	352.9	356.2

It is worth noting that even when running on a single thread, we observed an average of about two failed transactions per million, with as many as eight and as few as zero. For the purpose of this micro-benchmark these failed transactions were simply ignored as they did not affect the measurements.

V. FAILED TRANSACTIONS

As mentioned earlier, a transaction may fail to commit for many different reasons. In a real application this must be handled correctly or the execution will produce incorrect results. The most basic approach is to simply retry the transaction until it succeeds. However, since the prototype system used in our experiments provides a best-effort form of TM, this is not a viable approach. Our initial results further confirmed that some of our transactions never succeeded.

We observed that when this happens, the error is always of the **store** type, meaning that writing to the memory failed for some reason. This problem is also described in [3], and the solution they presented is to perform a write access to the address from outside of a transaction first. Doing so causes the page to be loaded into the TLB and marked as writable and the appropriate line loaded into the cache. To accomplish this, we perform a compare-and-swap operation to the memory page with dummy data (write zero, if content was zero), as described in [3].

However, even with this strategy, we found that some transactions still failed to commit. The reason turned out to be that when using the compiler-intrinsic TM-functions, the compiler reordered other write instructions into the transactions, and it was one of these writes that caused the transaction to fail. To solve this, we switched to using inlined assembly code for the transactions. However, we should note that this is not an intrinsic problem with TM, but rather an effect of using a prototype system that lacks adequate SW support.

We finally settled on the following approach for handling failed transactions. If a transaction fails with a **store** error, we execute a compare-and-swap to the address to initiate an appropriate TLB and cache load, and then retry the transaction. If a transaction fails with a **load** error, we read the data from outside the transaction and loop until the data is ready before retrying[1]. If the transaction fails due to a **coherence** error, we use a back-off scheme, which uses an exponentially increasing maximum delay to choose a random wait time before retrying.

```
while transaction fails
    if error == store
        compare-and-swap data
    else if error == load
        read data
    else if error == coherence
        back-off
    retry transaction
```

Listing 1. Strategy for handling a failed transaction.

Although we invested a lot of time and effort in handling failed transactions, this does not mean that using transactional memory must be difficult, as many of our solutions are general and can be included in a software library and reused.

VI. SCALING AND CONTENTION

To test how transactions and locks scale with different amounts of contention, we wrote a micro-benchmark that allows us to control the degree of conflict between the threads. The benchmark for n threads consists of n items[2] where each thread updates its own item constantly, and threads 2-n also update item 1 with some probability. By changing this probability we can control the likelihood of conflicts.

[1]The loop is performed using the architecture-specific *branch on register ready* instruction to determine when this data is ready.

[2]The data is arranged to avoid false sharing through cache lines between the threads.

Three versions of the benchmark was implemented, where the atomicity was achieved using locks, transactions, and compare-and-swap, respectively.

The benchmark also measures the actual number of concurrency conflicts by counting the number of initial concurrency failures in the transaction case and the number of initially busy locks in the lock case. In the benchmark, each thread performs 1,000,000 updates and each presented result is the median of 100 runs.

A. Results

In Figure 2, the time needed per update for the different implementations is plotted against the amount of contention. Here, 25% "potential conflicts" indicates that each thread would write to the shared value on 1 out of 4 updates. We see that the benchmark is dramatically slower for locks than for transactions and compare-and-swap.

The compare-and-swap version is fastest overall, except for the highest levels of contention where the transactions version levels out.

Although all benchmarks write to the shared memory address with the same probability, the actual number of conflicts varies between the versions. To analyze this, we looked at the actual percentage of conflicts (y-axis) compared to what we expected (x-axis) for 16 threads (Figure 3).

This data shows us that the lock-based approach experienced an almost perfect alignment between potential and actual conflicts, whereas when using the transaction-based approach, only about a twentieth of the potentially conflicting writes resulted in an actual conflict. The reason for this is not fully understood, but possible explanations are that the exponential back-off has this effect, or that the transactions get serialized by the memory system.

The difference in actual conflicts translates directly into the difference in performance for the benchmark.

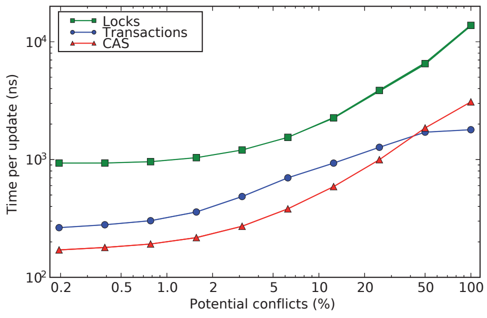


Figure 2. Scaling and contention: Time per update vs. potential conflict rate. The width of the lines represent the variation of the results from the lower to the upper quartile.

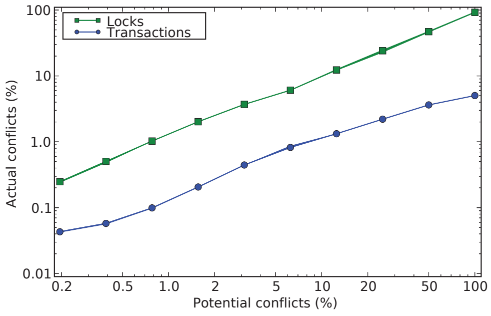


Figure 3. Scaling and contention: Actual conflict rate vs. potential conflict rate. The width of the lines represent the variation of the results from the lower to the upper quartile.

VII. Applications

Two common applications in the field of scientific computing were chosen for our study: assembling of a stiffness matrix in finite-element methods (FEM), and a simple particle dynamics simulation. These were chosen for evaluating transaction memory in scientific computing because they are both used in a multitude of computational science applications.

It has been proposed that transactions have the potential to simplify parallel programming. If transactions performed significantly better than the corresponding locks implementation, this might reduce the need for algorithm redesign in order to avoid concurrent memory accesses. Therefore we have considered straight-forward algorithms rather than the most advanced ones.

VIII. Stiffness Matrix Assembly

Solving a differential equation by using the finite element method consists of two steps: Assembling the stiffness matrix for the problem and solving a linear system of equations. There are numerous efficient and highly parallel algorithms available for the solution of the linear system of equations in a FEM solver, including parallel matrix factorizations [5] and parallel iterative methods as, e.g., the multigrid method [6]. However, efficient parallelization of the matrix assembly is more difficult and this is often still performed using a serial algorithm.

The reason that the matrix assembly is a difficult problem to parallelize is that it consists of a large number of concurrent updates of matrix elements with a highly irregular access pattern. Listing 2 shows the basic algorithm for constructing the stiffness matrix.

Given a FEM mesh of n_p points and n_t triangles, the assembly process consists of looping over all triangles and filling in the corresponding elements in a zero-initialized $n_p \times n_p$ matrix M. For each triangle, a local 3×3 matrix `m` is computed, whose elements are then distributed to the global matrix `M` at locations determined by the vertices of the current triangle `t`.

```
for t in triangle_list
    m = compute_local_matrix(t)
    for i in 1..3
        for j in 1..3
            M(t(i), t(j)) += m(i,j)
```

Listing 2. Stiffness-matrix assembly

A. Parallel Implementation

We have used the straight-forward way of parallelizing the algorithm in Listing 2 by introducing parallelism at the level of the outermost loop. Since each non-zero element of the matrix will be updated multiple times, the update constitutes a critical section and has to be protected. A general solution is shown in Listing 3, where the **atomic** keyword suggests that the corresponding operation must be executed atomically. This can be implemented using locks, by performing the operation in a transaction, or by using the `compare-and-swap` instruction.

```
for t in my_triangle_list
    m = compute_local_matrix(t)
    for i in 1..3
        for j in 1..3
            atomic {
                M(t(i), t(j)) += m(i,j)
            }
```

Listing 3. General parallelization of the matrix assembly.

If Listing 3 is implemented using locks, the straight forward implementation would use one lock per matrix element. This means that a full matrix of $n_p \times n_p$ locks is required, making the space requirement for the locks larger than that of the matrix to be assembled, as a lock is larger in size than a matrix element.

If Listing 3 is instead implemented using transactions or compare-and-swap, this space issue can be avoided. An additional benefit is avoiding the lock accesses, which will not cache well due to the irregular and sparse access pattern. For the transaction-based and CAS-based versions we therefore expect a smaller memory footprint as well as a higher ratio of computation-to-data, resulting in better scaling than for the lock-based code.

To make our experiments resemble a modern FEM assembly scheme, we introduce a significant number of floating point operations to emulate the computation of the local element matrices. Algorithms such as the element preconditioning described in [7], where a small optimization has to be performed for each triangle and multi-scale FEM [8], where a smaller FEM problem has to be solved within each triangle, frequently have non-trivial amounts of computation in the local element calculation. By varying the number of floating point operations for the local computations, we can vary the computation-data access ratio for the algorithm. Since both the transactions-based program and the CAS-based program have fewer memory accesses to start with, we expect them to perform better at a smaller number of local floating point computations compared to the locks-based version.

B. Experiments

The three different parallel implementations of Listing 3, with locks, compare-and-swap and transactions, respectively, as well as a serial reference version corresponding to Listing 2, were tested with 1, 100 and 1000 operations inside the local computations, and the run-times were measured. To obtain stable measurements, each implementation was executed 100 times, and the median run-time is reported. Similarly to in the micro benchmarks, the upper and lower quartiles are included in the plots. To concentrate on the assembly itself, we do not include memory allocation, reading of the mesh data or initialization of threads in the measurement.

The mesh used in the experiments was generated in MATLAB's PDE Toolbox using the `initmesh` function with an `hmax` value of 0.05. It has 4146 triangles and 2154 points, which requires 82 kB of memory. The memory required for the generated matrix is 35 MB, and the memory required for locks in the lock-based version is 106 MB, both of which easily exceed the 2 MB of combined L2 cache.

C. Results

The results from the experiments are shown in Table II and Figure 4. For clarity, the results from a workload of 100 operations have been suppressed in Figure 4, but can be found in Table II. We see that the speed-up improves with the number of operations performed for each triangle, which is expected as this increases the number of computations per memory access. The transaction-based programs performed better than the lock-based ones for all workloads. Compare-and-swap performed the best for small workloads, but when the workload was increased, it behaved similarly to the lock-based version. It also had the largest statistical variation, especially at large workloads. One reason that the variations are generally so large for this application is that the run-times are very small; less than 5 ms on 16 threads.

Note that in the case of the lock-based program, the allocation and initialization of the locks will account for a significant portion of the execution time, which is not included here.

IX. N-Body Simulation

N-body simulations are used in a vast range of computational science applications for determining the motion of n_p mutually interacting "particles", where the particles could represent anything ranging from atoms in molecular dynamics to stars and galaxies in astrophysics.

In standard force-field models (e.g. involving electromagnetic forces or gravitation) the particles interact pairwise

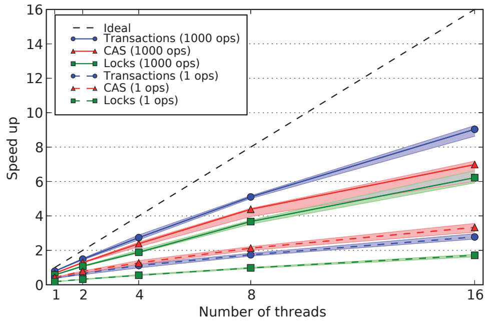


Figure 4. FEM assembly: Median speed-up relative to serial at different workloads. The width of the lines represent the variation of the results from the lower to the upper quartile. Note that the intervals for CAS (1000 ops) and Locks (1000 ops) overlap at 16 threads in the figure.

Table II
FEM ASSEMBLY: MEDIAN RUN-TIMES IN MS (SPEED-UP VS 1 THREAD)

N_{threads}	1	2	4	8	16
Serial (1 ops)	4.09	-	-	-	-
HTM	10.87	6.18	3.65	2.35	1.47
	(1.0)	(1.8)	(3.0)	(4.6)	(7.4)
CAS	9.52	5.20	3.22	1.92	1.23
	(1.0)	(1.8)	(3.0)	(5.0)	(7.7)
Locks	23.20	12.68	7.42	4.17	2.39
	(1.0)	(1.8)	(3.1)	(5.6)	(9.7)
Serial (100 ops)	6.50	-	-	-	-
HTM	13.47	7.79	4.54	2.64	1.63
	(1.0)	(1.7)	(3.0)	(5.1)	(8.3)
CAS	12.30	6.58	3.90	2.30	1.42
	(1.0)	(1.9)	(3.2)	(5.3)	(8.7)
Locks	26.39	14.33	8.43	4.70	2.70
	(1.0)	(1.8)	(3.1)	(5.6)	(9.8)
Serial (1000 ops)	26.33	-	-	-	-
HTM	33.38	17.68	9.62	5.16	2.91
	(1.0)	(1.9)	(3.5)	(6.5)	(11.5)
CAS	39.41	20.46	11.05	6.00	3.77
	(1.0)	(1.9)	(3.6)	(6.6)	(10.5)
Locks	46.26	24.31	13.95	7.17	4.23
	(1.0)	(1.9)	(3.3)	(6.5)	(10.9)

with each other. When the force between a pair of particles has been calculated, both particles need their force variable updated. When the task to calculate the forces is distributed over several threads, two threads must not update the force variable of a given particle simultaneously. Such collisions are expected to be fairly rare, making the application suitable for transactions by exploiting their optimistic nature.

We evaluate the interaction between each pair of particles ($O(n_p^2)$ interactions) in each time step. In more advanced algorithms, attempts are made to avoid calculating all interactions, such as by ignoring long-range interactions, exploiting periodicity, or grouping distant particle together. However, for long-range force fields and for particles within a short distance of each other, all interactions need to be evaluated explicitly, as examined here.

A. Implementation

A straight-forward serial implementation of the force evaluations is shown in Listing 4:

```
for $i = 0$ to $n_p - 1$
    for $j = i + 1$ to $n_p - 1$
        $\Delta f$ = evalForce($p_i$,$p_j$)
        $f_i$ += $\Delta f$
        $f_j$ -= $\Delta f$
```

Listing 4. N-body: Simple serial implementation.

Here f_i is the total force acting upon particle p_i, and Δf is the contribution to f_i from particle p_j.

B. Parallel Implementation

When parallelizing the algorithm in Listing 4, care must be taken to prevent multiple threads from concurrently updating the force acting on a given particle. Furthermore, dividing the outer loop up in even-sized chunks and distributing them over the cores gives poor load balancing, as the amount of work in the inner loop varies. For better load balancing, we instead distribute the outer-most loop in a cyclic manner over the threads. This assignment to threads can either be done statically, as in Listing 5, or dynamically.

```
1 for $i =$ id to $n_p - 1$ step by $n_t$
2     for $j = i + 1$ to $n_p - 1$
3         $\Delta f$ = evalForce($p_i$,$p_j$)
4         atomic { $f_i$ += $\Delta f$ }
5         atomic { $f_j$ -= $\Delta f$ }
```

Listing 5. General parallelization of the n-body force evaluation.

In Listing 5, id is the thread number and n_t is the number of threads. Note that Listing 5 only suggests that row 4 and 5 need to be executed atomically and says nothing about whether they make one critical section each or share a single critical section.

C. Implementation Using Locks

The first implementation, called **basic locks** follows the code in Listing 5, and uses one lock per particle. This leads to many locks and too much time spent on acquiring and releasing them. To improve this, we can group the particles such that each lock protects a group of particles. However, if the group size is chosen too large then groups are more likely to conflict and too much time will be spent waiting for the group locks instead. In our case it turned out that grouping particles into groups of 4 gave the best performance.

When dynamic scheduling is used, the contention on the lock protecting the global index can be high. This issue can be addressed by assigning whole chunks of indices to threads, rather than treating them individually. Also here, we found that handling chunks of 16 indices gave the best performance. This method is referred to as **blocked locks**.

D. Implementation Using Private Buffers

A different approach is to trade concurrency for memory and letting each thread have its own private buffers for calculated force contributions to which the thread have exclusive access. These private buffers are then added to the global variables in parallel when all force contributions have been calculated. This requires $O(n_p n_t)$ memory instead of $O(n_p)$, but completely avoids concurrent updates of the forces. It also requires a more substantial modification of the original algorithm than in the other approaches. This implementation is called **private buffers**.

E. Implementation Using Transactions

For the implementations using transactions, the forces are updated within transactions to get atomic updates. We use the same grouping strategies as when using locks, to get the same memory access patterns. The implementation that updates a single particle in each transaction, is called **basic transactions**, and the implementation where particles and indices are blocked into groups is called **blocked transactions**.

The failed transaction handling strategy was adjusted somewhat as we found that the **store** error occur so frequently that always including the compare-and-swap operation yielded better results.

F. Implementation Using CAS

An implementation of Listing 5 using compare-and-swap was also included, which is called **basic CAS**. We also implemented a version grouping particles and indices as in the locks implementation, called **blocked CAS**.

G. Experiments

Several different implementations were used and evaluated for the serial, parallel, and transactional memory versions. Just as in the matrix assembly application, we report the median run-time from 100 executions, and upper and lower quartiles. In each execution, 1024 particles were simulated for 40 time steps. The execution time (wall-clock time) was only measured for the last 20 time steps to concentrate on the behavior of long runs and filter out possible initialization issues such as filling the caches. Storing 1024 particles only required 56 kB of memory in our implementations, meaning that all data easily fit in the L2 cache. Because of the high computation-to-data ratio of this algorithm (the number of force evaluations grows quadratically with the number of particles), increasing the problem size further quickly resulted in intractably long run-times.

H. Results

The results of the n-body experiments are shown in Table III and Figure 5. We first note that the statistical variation is small compared to that of the FEM application, and barely visible in the figure. This is probably due to the longer execution times.

When comparing the most basic versions, the locks and transactions versions performed similarly, while compare-and-swap was slightly faster. The basic versions all performed poorly, with a speed-up factor below 3 when running on 16 cores.

Introducing blocking improved the performance in all cases, resulting in speed-ups of up to almost 9 in the case of transactions. However, we see that the benefit was smaller for compare-and-swap compared to the other two. This is expected from the results in the overhead benchmark.

The fastest run-times were obtained when using the private-buffers implementation, which avoid concurrent memory accesses completely. At 16 threads, it had a speed-up factor of almost 14.

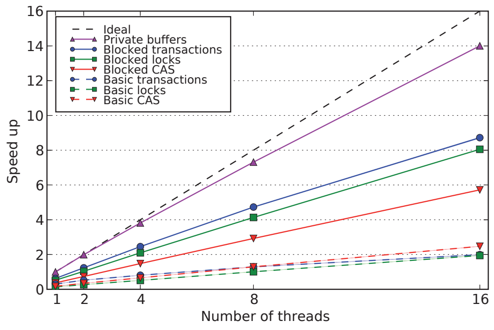


Figure 5. N-body: Median speed-up relative to serial. The width of the lines represent the variation of the results from the lower to the upper quartile.

Table III
N-BODY: MEDIAN RUN-TIMES IN SECONDS (SPEED-UP VS 1 THREAD)

N_{threads}	1	2	4	8	16
Basic locks	8.99 (1.0)	4.62 (1.9)	2.42 (3.7)	1.23 (7.3)	0.64 (14.0)
Basic transactions	4.14 (1.0)	2.36 (1.8)	1.53 (2.7)	0.96 (4.3)	0.62 (6.7)
Basic CAS	7.09 (1.0)	3.59 (2.0)	1.88 (3.8)	0.96 (7.4)	0.50 (14.2)
Blocked locks	2.37 (1.0)	1.20 (2.0)	0.59 (4.0)	0.30 (7.9)	0.15 (15.8)
Blocked transactions	1.98 (1.0)	1.00 (2.0)	0.51 (3.9)	0.26 (7.6)	0.14 (14.1)
Blocked CAS	3.34 (1.0)	1.68 (2.0)	0.84 (4.0)	0.42 (8.0)	0.22 (15.2)
Private buffers	1.24 (1.0)	0.62 (2.0)	0.33 (3.8)	0.17 (7.3)	0.09 (13.8)

X. Conclusions and Future Work

This work investigated the performance of hardware transactional memory for scientific computing. Our micro-benchmarks show that the potential of transactions is significant, particularly for replacing fine-grained locking as the overhead is one-third that of locks. This, combined with eliminating the bandwidth overhead of accessing lock variables, resulted in significant speed-ups for the FEM stiffness matrix generation we examined. An additional benefit of using transactions is that the initialization and storage of lock variables is avoided.

The experiments on the n-body application showed that when using similar algorithms, using transactions was slightly faster than using locks. However, the private buffers approach we used showed (unsurprisingly) that changing the underlying algorithm significantly to avoid most of the synchronization can be far more effective. This shows that replacing locks by transactions is not an option to reformulating the algorithm to avoid concurrency when aiming for high performance.

Using the compare-and-swap instruction to achieve atomicity turned out to perform well in some experiments. However, this is only a viable approach when performing single element updates, since compare-and-swap is limited to these. The good performance is interesting considering that we had to move data between the floating-point and integer registers. We expect that hardware support for floating-point compare-and-swap would improve the performance further.

Furthermore, our experience on this prototype system is that programming using best-effort hardware transactional memory is non-trivial. We experienced significant difficulty in developing a reliable and efficient mechanism for handling failed transactions. We think that there is room for improvement of our fail handling, which could improve the performance further.

Other topics of interest for future research are investigation of more applications, such as sparse-matrix operations, and experimenting with HTM in existing, lock-based codes. It would also be interesting to compare the results from the FEM application to other, more advanced, implementations.

In summary, transactions showed to be the fastest way to concurrently update shared floating-point variables, and also avoids the additional storage, initialization, and memory accesses that come with locks. Compare-and-swap performs well but are limited to single elements updates. The best performance overall is achieved if concurrent updates can be avoided altogether.

Acknowledgment

This work was supported by the Swedish Research Council and carried out within the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, no. 2, pp. 289–300, 1993.

[2] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance Sparc CMT processor," *IEEE Micro*, vol. 29, pp. 6–16, 2009.

[3] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, "Early experience with a commercial hardware transactional memory implementation," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2009, pp. 157–168.

[4] D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski, "Early experience with a commercial hardware transactional memory implementation," Sun Laboratories, Tech. Rep. TR-2009-180, 2009.

[5] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[6] A. M. Bruaset and A. Tveito, Eds., *Numerical solution of partial differential equations on parallel computers*, ser. Lecture Notes in Computational Science and Engineering. Berlin: Springer-Verlag, 2006, vol. 51. [Online]. Available: http://dx.doi.org/10.1007/3-540-31619-1

[7] G. Haase, U. Langer, S. Reitzinger, and J. Schoberl, "Algebraic multigrid methods based on element preconditioning," *International Journal of Computer Mathematics*, vol. 78, no. 4, pp. 575–598, 2001.

[8] T. Hou and X. Wu, "A multiscale finite element method for elliptic problems in composite materials and porous media," *Journal of Computational Physics*, vol. 134, no. 1, pp. 169–189, JUN 1997.

Paper II

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 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

$$A^k_{l,m} = \int_{\Omega_k} \nabla \psi_l^k \cdot \nabla \psi_m^k \mathrm{d}V \,. \tag{9}$$

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.

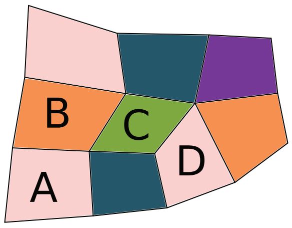


Fig. 1. Elements A and B are neighbors, as are elements A and C, and are thus given different colors. Elements A and D are not neighbors and can be given the same color.

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.

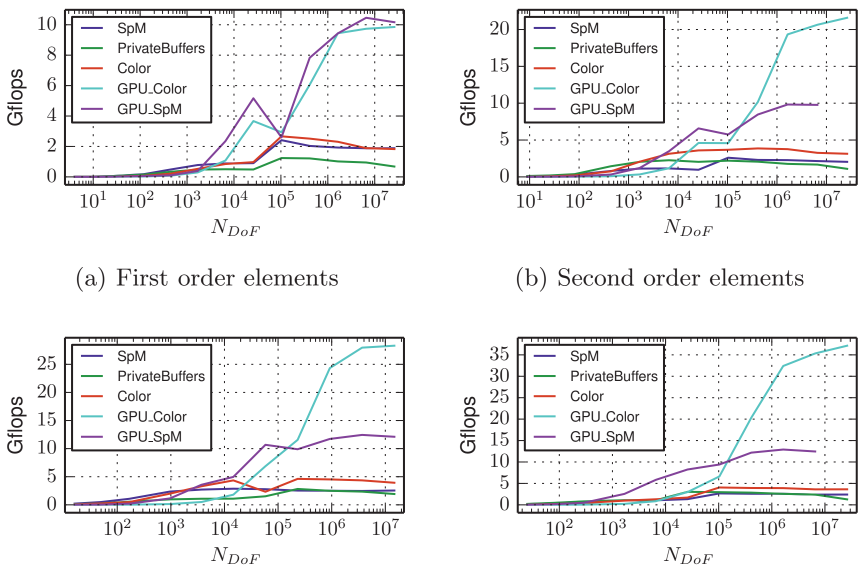
To evaluate the execution time for the operator application, the time for steps 2–4 is measured, and the time for a single application is calculated by dividing by the number of iterations, i.e. 20. Furthermore, to get more stable results, 20 repetitions of steps 2–4 are performed, and the minimum time 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

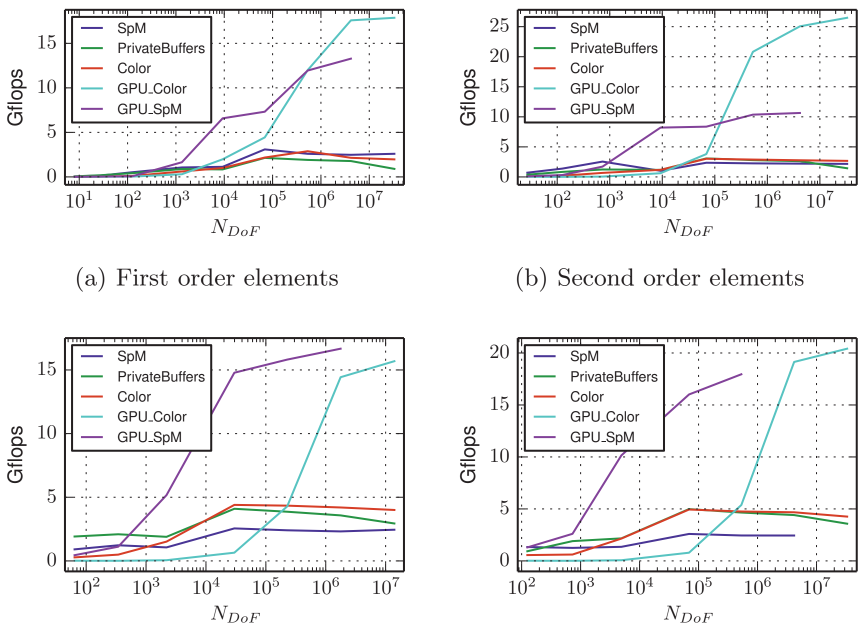
(c) Third order elements

(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 ($4.5 - 10\times$). 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

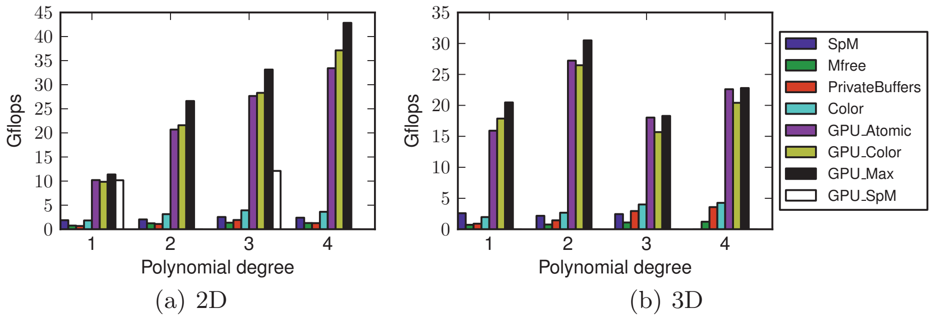


Fig. 4. Performance for the largest problems solved (with 26.2M, 26.2M, 14.8M, and 26.2M DoFs (2D) ; and 33.1M, 33.1M, 14.0M and 33.1M DoFs (3D), respectively). The missing bars for `SpM` and `GPU_SpM` indicate a fail, i.e. the matrix did not fit in main memory.

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)

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden