UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1512

Finite Element Computations on Multicore and Graphics Processors

KARL LJUNGKVIST

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017

ISSN 1651-6214
ISBN 978-91-554-9907-5
urn:nbn:se:uu:diva-320147

Dissertation presented at Uppsala University to be publicly examined in ITC 2446, Lägerhyddsvägen 2, Uppsala, Friday, 9 June 2017 at 10:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Xing Cai (Simula Research Laboratory).

Abstract
Ljungkvist, K. 2017. Finite Element Computations on Multicore and Graphics Processors. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1512. 64 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-554-9907-5.

In this thesis, techniques for efficient utilization of modern computer hardwarefor numerical simulation are considered. In particular, we study techniques for improving the performance of computations using the finite element method.

One of the main difficulties in finite-element computations is how to perform the assembly of the system matrix efficiently in parallel, due to its complicated memory access pattern. The challenge 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 multicore processor supporting it. Our experiments show that transactions can both simplify programming and provide good performance for concurrent updates of floating point data.

Secondly, we study a matrix-free approach to finite-element computation which avoids the matrix assembly. In addition to removing the need to store the system matrix, matrix-free methods are attractive due to their low memory footprint and therefore better match the architecture of modern processors where memory bandwidth is scarce and compute power is abundant. Motivated by this, we consider matrix-free implementations of high-order finite-element methods for execution on graphics processors, which have seen a revolutionary increase in usage for numerical computations during recent years due to their more efficient architecture. In the implementation, we exploit sum-factorization techniques for efficient evaluation of matrix-vector products, mesh coloring and atomic updates for concurrent updates, and a geometric multigrid algorithm for efficient preconditioning of iterative solvers. Our performance studies show that on the GPU, a matrix-free approach is the method of choice for elements of order two and higher, yielding both a significantly faster execution, and allowing for solution of considerably larger problems. Compared to corresponding CPU implementations executed on comparable multicore processors, the GPU implementation is about twice as fast, suggesting that graphics processors are about twice as power efficient as multicores for computations of this kind.

Keywords: Finite Element Methods, GPU, Matrix-Free, Multigrid, Transactional Memory

Karl Ljungkvist, Department of Information Technology, Division of Scientific Computing, Box 337, Uppsala University, SE-751 05 Uppsala, Sweden. Department of Information Technology, Computational Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-554-9907-5
urn:nbn:se:uu:diva-320147 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-320147)

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

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.

III K. Ljungkvist. Matrix-Free Finite-Element Computations on Graphics Processors with Adaptively Refined Unstructured Meshes. In *HPC '17: Proceedings of the 25th High Performance Computing Symposium*. Society for Computer Simulation International, 2017. *(In press)*
Contribution: The author of this thesis is the sole author of this paper.

IV K. Ljungkvist, and M. Kronbichler. Multigrid for Matrix-Free Finite Element Computations on Graphics Processors. Technical Report 2017-006, Department of Information Technology, Uppsala University, April 2017.
Contribution: The method and implementation were made by the author of this thesis. The design of experiments, the running of benchmarks, and the writing of the manuscript, were all done in collaboration with the second author.

Contents

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 the 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 with time-dependent data.

A closely related problem shows up when using high-order finite-element methods with a matrix-free approach. When simulating phenomena where solutions are smooth, such 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, the higher the order of the elements, the denser the system matrix becomes, putting a severe restriction on how large systems can be simulated. Moreover, as the element order increases, the sparse matrix-vector product comprising the absolute majority of the workload in the linear solver becomes increasingly memory bandwidth hungry, leading to poor utilization of modern multicore and manycore processors where bandwidth is expensive and computations are cheap. By merging the assembly phase into the matrix-vector product, a matrix-free operator application algorithm without an explicit system matrix is obtained, which is much more efficient in terms of memory, in particular for higher-order elements and in 3D. The matrix-free operator application is algorithmically similar to the assembly, meaning that techniques for parallelization and optimization of the assembly can be used also for 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 of avoiding race conditions to maintain correctness.

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

1.2 Disposition

The remainder of this thesis is structured as follows. 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 papers are also presented. Finally, in Chapter 4, an overview is given of the results and their implications in a wider setting.

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 [48]. Rather than solving the strong form of a partial differential equation, the finite element method finds approximations of solutions to the variational, or weak, formulation. Consider the example of a Poisson model problem,

$$\begin{cases} -\nabla^2 u = f & \text{in } \Omega, \\ u = 0 & \text{on } \partial\Omega. \end{cases} \tag{2.1}$$

This equation is a simple model problem, but it serves the purpose of illustrating the method, and can easily be extended to include, e.g., a vector valued solution, a variable coefficient, or a non-linearity. Furthermore, this equation appears as an important part of more complex applications, such as in projection-correction methods for solution of the Navier-Stokes equations, and when applying the Laplace operator in time integration of the wave equation.

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

$$(\nabla u, \nabla v) = (f, v) \; \forall v \in \mathcal{V}. \tag{2.2}$$

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 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 Papers II–IV. For the case of triangle elements and their higher-dimensional generalizations, one typically considers polynomials of order p, referred to as $\mathcal{P}_p$. For quadrilateral elements, one instead usually considers functions that are products of one polynomial of order p in each coordinate direction, e.g., bilinear for $p = 1$ or biquadratic for $p = 2$ in 2D. These elements are referred to as $\mathcal{Q}_p$. In the remainder 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

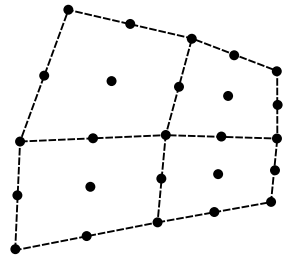


Figure 2.1. Location of the node points for second-order quadrilateral elements in 2D ($\mathcal{Q}_2$).

$\{\varphi_i\}$ 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

$$\sum_{i=1}^{N} (\nabla\varphi_j, \nabla\varphi_i)\, u_i = (f, \varphi_j)\ j = 1 \ldots N \tag{2.3}$$

or, in matrix notation,

$$A\mathbf{u} = \mathbf{f}, \tag{2.4}$$

where

$$A_{ij} = (\nabla\varphi_j, \nabla\varphi_i)\,, \mathbf{f}_j = (f, \varphi_j) \tag{2.5}$$

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:

(a) Assembly phase

(b) 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 (2.4). 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 [65]. Parallelizations of the solve phase thus rely 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,

$$A_{ij} = (\nabla\varphi_i, \nabla\varphi_j) = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}\mathbf{x}. \tag{2.6}$$

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

$$A_{ij} = \sum_{k\in\mathcal{K}} \int_{\Omega_k} \nabla\varphi_i \cdot \nabla\varphi_j \mathrm{d}\mathbf{x}. \tag{2.7}$$

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 $(P_k)_{i,j} = \delta_{i,I^k(j)}$, where $\delta_{i,j}$ is the Kronecker delta. Using this, and introducing $\varphi_l^k = \varphi_{I^k(l)}$ as the l'th basis function on element k, we can write (2.7) on matrix form as

$$A = \sum_{k\in\mathcal{K}} P_k a^k P_k^T, \tag{2.8}$$

where the local matrix a^k is defined as

$$a^k_{l,m} = \int_{\Omega_k} \nabla\varphi_l^k \cdot \nabla\varphi_m^k \mathrm{d}\mathbf{x}. \tag{2.9}$$

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 (2.8). 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 (2.8) 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.

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 [13, 17]. 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 restricted by the available memory bandwidth rather than compute power [33, Chapter 7][29]. For the SpMV operation, 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, a matrix-free finite-element method has been suggested by several authors, first introduced by Carey et al. [16]. 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 in a multiplication with a vector v. This means that if we can find a recipe for how to form Av without having access to an explicit A, we can use the matrix-free variant in the linear solver, just as we would have used an explicit matrix.

Matrix-free techniques 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 [40].

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 [43]. 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 [59]. 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 2.3.2.

Melenk et al. investigate efficient techniques for assembly of the spectral-element stiffness matrix, based on a tensor-product approach [52], In [15], 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 [44]. The framework has been included in the open-source finite-element framework `deal.II` [5, 7, 8].

We form the matrix-free operator by using the definition of A from (2.8),

$$Av = \left(\sum_{k \in \mathcal{K}} P_k a^k P_k^T \right) v = \sum_{k \in \mathcal{K}} \left(P_k a^k P_k^T v \right) . \tag{2.10}$$

In other words, we have simply changed the order of the summation and multiplication. Once again, the P_k are permutation matrices which simply picks out the correct DoFs and distribute 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,

$$\begin{aligned} &\text{(a)} \quad v_k = P_k^T v \\ &\text{(b)} \quad u_k = a^k v_k \\ &\text{(c)} \quad u = \textstyle\sum_{k \in \mathcal{K}} P_k u_k \end{aligned} \tag{2.11}$$

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 (2.11) 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 (2.9), a mapping $\mathbf{x} = f_k(\boldsymbol{\xi})$ from a reference unit element $\hat{T}$ to element k is used. Using this transformation, (2.9) can be rewritten as

$$a_{ij}^k = \int_{\hat{T}} \left(J_k^{-1} \nabla_{\boldsymbol{\xi}} \varphi_i(\boldsymbol{\xi})\right) \cdot \left(J_k^{-1} \nabla_{\boldsymbol{\xi}} \varphi_j(\boldsymbol{\xi})\right) |J_k| \mathrm{d}\boldsymbol{\xi}\,, \tag{2.12}$$

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

$$a_{ij}^k = \sum_{q=1}^{N_q} \left(J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} \varphi_i(\boldsymbol{\xi}_q)\right) \cdot \left(J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} \varphi_j(\boldsymbol{\xi}_q)\right) |J_k(\boldsymbol{\xi}_q)| w_q\,, \tag{2.13}$$

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, leading to a very favorable memory bandwidth footprint. 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.

On the other hand, for a general mesh, each quadrature point of each element has a distinct Jacobian, leading to distinct local matrices at each element. The same is true also for uniform meshes if the PDE contains a variable coefficient or a non-linearity. In this case, precomputing and storing the individual local matrices yields much too much data to be efficient – in fact even more than the original sparse matrix representation.

To find a method for applying the local matrices when these are distinct, we will now 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,

$$u_i = \sum_{j=1}^{n_p} a_{ij} v_j \,, \tag{2.14}$$

where n_p is the number of local DoFs. Equation (2.13) lets us rewrite this operation in the following manner,

$$u_i = \sum_{q=1}^{n_q} \left(J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} \varphi_i(\boldsymbol{\xi}_q) \right) \cdot \left[J_k^{-1}(\boldsymbol{\xi}_q) \sum_{j=1}^{n_p} \nabla_{\boldsymbol{\xi}} \varphi_j(\boldsymbol{\xi}_q) v_j \right] |J_k(\boldsymbol{\xi}_q)| w_q \,. \tag{2.15}$$

The entity in brackets is simply the gradient of v evaluated on the q'th quadrature point. If we define

$$\nabla v_q = J_k^{-1}(\boldsymbol{\xi}_q) \sum_{j=1}^{n_p} \nabla_{\boldsymbol{\xi}} \varphi_j(\boldsymbol{\xi}_q) v_j \,, \tag{2.16}$$

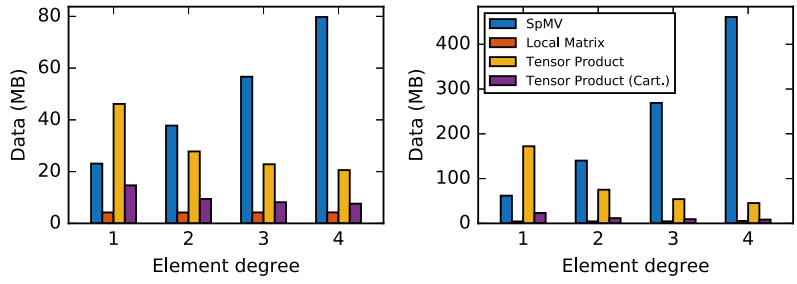
then, (2.15) can be written as

$$u_i = \sum_{q=1}^{n_q} \left(J_k^{-1}(\boldsymbol{\xi}_q) \nabla_{\boldsymbol{\xi}} \varphi_i(\boldsymbol{\xi}_q) \right) \cdot \nabla v_q |J_k(\boldsymbol{\xi}_q)| w_q \,, \tag{2.17}$$

or

$$u_i = \sum_{q=1}^{n_q} \nabla_{\boldsymbol{\xi}} \varphi_i(\boldsymbol{\xi}_q) \cdot J_k^{-T}(\boldsymbol{\xi}_q) \nabla v_q |J_k(\boldsymbol{\xi}_q)| w_q \,. \tag{2.18}$$

In (2.16) and (2.18), the only things that are specific to each element and need to be stored in memory are the inverse Jacobians $J_k^{-1}(\boldsymbol{\xi}_q)$ and the Jacobian determinant $|J_k(\boldsymbol{\xi}_q)|$. Comparing the memory consumption of this to that of a sparse matrix, we see that the matrix free method consumes less memory for second-order elements and higher (see Figure 2.2). For more details see, e.g., [15]. Finally, we note that for Cartesian meshes, the inverse Jacobians are constant within each element and also proportional to the identity matrix. This means that in that case only a single value needs to be stored for each element, yielding a further drastic reduction in memory consumption, which can be seen in Figure 2.2.



(a) 2D (b) 3D

Figure 2.2. Comparison of operator approaches with respect to bandwidth usage. The bandwidth is computed as the amount of data which has to be read for a single Laplace-operator evaluation on a Cartesian mesh with 263k DoFs in 2D and 275k DoFs in 3D.

2.3.2 Sum-Factorization for Tensor-Product Elements

While a lower memory usage will translate to better performance for bandwidth bound algorithms, it is still of interest to keep the operation count as low as possible. In fact, (2.16) and (2.18) comprise a significant number of computations in the form of several matrix-vector products of the same magnitude as the local matrix multiplication, in addition to the applications of the inverse Jacobian. However, for finite elements with tensor-product basis functions, such as the quadrilateral and hexahedral $\mathcal{Q}_p$ elements under consideration, the operations in (2.16) and (2.18) can be reformulated to involve considerably less operations.

For such tensor-product elements, the basis functions φ_i can be expressed as tensor products of d one dimensional basis functions ψ_μ. Assuming three dimensions, we get

$$\varphi_i(\boldsymbol{\xi}) = \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3)\,, \tag{2.19}$$

where we have introduced the multi-index $i = (\mu, \nu, \sigma)$. For the basis function gradient, this implies

$$\nabla\varphi_i(\boldsymbol{\xi}) = \begin{pmatrix} \psi'_\mu(\xi_1)\psi_\nu(\xi_2)\psi_\sigma(\xi_3) \\ \psi_\mu(\xi_1)\psi'_\nu(\xi_2)\psi_\sigma(\xi_3) \\ \psi_\mu(\xi_1)\psi_\nu(\xi_2)\psi'_\sigma(\xi_3) \end{pmatrix}. \tag{2.20}$$

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

$$\boldsymbol{\xi}_q = (\xi^\alpha, \xi^\beta, \xi^\gamma)\,, \tag{2.21}$$

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

$$\psi^\alpha_\mu = \psi_\mu(\xi^\alpha)\,,\ \chi^\alpha_\mu = \psi'_\mu(\xi^\alpha)\,, \tag{2.22}$$

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

$$v^{\alpha\beta\gamma} = \sum_{\mu} \psi_{\mu}^{\alpha} \sum_{\nu} \psi_{\nu}^{\beta} \sum_{\sigma} \psi_{\sigma}^{\gamma} v_{\mu\nu\sigma} \,. \tag{2.23}$$

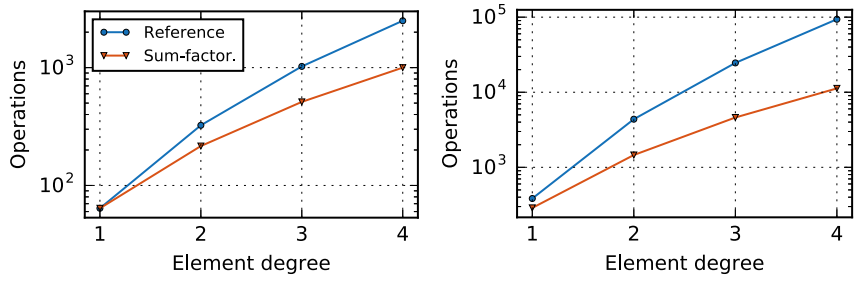
Using these new identities, we can now factorize the sums in (2.16) and rewrite it as

$$\nabla v^{\alpha\beta\gamma} = (J^{\alpha\beta\gamma})^{-1} \sum_{\mu} \begin{pmatrix} \chi_{\mu}^{\alpha} \\ \psi_{\mu}^{\alpha} \\ \psi_{\mu}^{\alpha} \end{pmatrix} \sum_{\nu} \begin{pmatrix} \psi_{\nu}^{\beta} \\ \chi_{\nu}^{\beta} \\ \psi_{\nu}^{\beta} \end{pmatrix} \sum_{\sigma} \begin{pmatrix} \psi_{\sigma}^{\gamma} \\ \psi_{\sigma}^{\gamma} \\ \chi_{\sigma}^{\gamma} \end{pmatrix} v_{\mu\nu\sigma} \,, \tag{2.24}$$

where $J^{\alpha\beta\gamma} = J_k(\boldsymbol{\xi}_q)$ and the vector products should be interpreted as element-wise multiplication. Likewise, for the second step, the numerical integration in (2.18), we have

$$u_{\mu'\nu'\sigma'} = \sum_{\alpha} \begin{pmatrix} \chi_{\mu'}^{\alpha} \\ \psi_{\mu'}^{\alpha} \\ \psi_{\mu'}^{\alpha} \end{pmatrix} \sum_{\beta} \begin{pmatrix} \psi_{\nu'}^{\beta} \\ \chi_{\nu'}^{\beta} \\ \psi_{\nu'}^{\beta} \end{pmatrix} \sum_{\gamma} \begin{pmatrix} \psi_{\sigma'}^{\gamma} \\ \psi_{\sigma'}^{\gamma} \\ \chi_{\sigma'}^{\gamma} \end{pmatrix} \left(J^{\alpha\beta\gamma}\right)^{-T} \nabla v^{\alpha\beta\gamma} w^{\alpha\beta\gamma} |J^{\alpha\beta\gamma}| \,. \tag{2.25}$$

The two very similar operations (2.24) and (2.25) both consist of a series tensor contractions – essentially dense matrix-matrix products – and a couple of element-wise scalar and vector operations. In Figure 2.3, we compare the operation count of these computations with that of (2.16) and (2.18). Clearly, the sum factorization of the tensor products yields a significant improvement, especially for elements of high complexity, i.e., high dimension and polynomial order. In Papers III and IV, we use the tensor-product sum-factorization approach to efficiently evaluate the Laplacian in a matrix-free manner (see Section 3.5.2 for performance results).



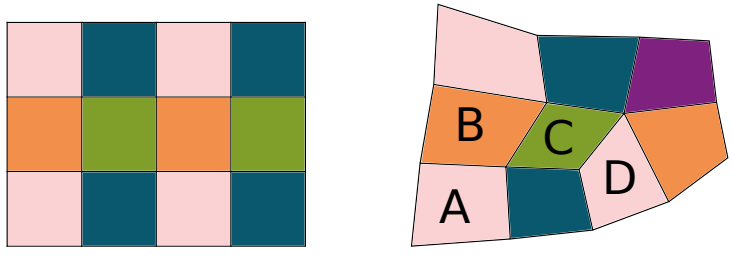
(a) 2D (b) 3D

Figure 2.3. Comparison of the number of arithmetic operations of the local product for the sum-factorization approach and the basic method with large matrix-vector products. Note the logarithmic scale on the y-axis.

2.3.3 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 2.4 (a) and (b).



(a) (b)

Figure 2.4. 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.

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.

For a general mesh, finding the coloring can be done using various graph coloring algorithms [11, 27, 42]. In Paper III, we employ a coloring algorithm by Turcksin et al [71]. For meshes with hanging nodes (see next Section 2.4), we note that the concept of neighborhood changes slightly. In this case, elements with hanging nodes are considered neighbors with elements that have vertices to which its hanging nodes are constrained.

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 [1, 69]. 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 [69], Tillenius applies SuperGlue and achieves very good scaling for an N-body problem similar to the one considered in Paper I.

2.4 Adaptive Refinement with Hanging Nodes

The accuracy of a finite-element discretization is determined by the size of the elements in the mesh. Thus, by refining the mesh to get smaller elements, the approximation can be improved. On the other hand, as the element size decreases the number of elements increases correspondingly and with it the amount of computational work and memory, so there is an upper limit to how small elements can be used if uniform mesh refinement is used. Fortunately, the error is typically not evenly spread out over the computational domain, but rather localized to areas where the solution, the data, or the geometry has abrupt changes. By using a finer mesh at these areas and a coarser mesh in areas where requirements are lower, we can reduce the error as efficiently as possible with a given number of elements. This localized refinement is usually referred to as *adaptive* mesh refinement.

For meshes with triangular or tetrahedral elements, adaptive refinement can be done straightforwardly by subdividing the elements flagged for refinement, and an additional small set of neighboring elements in order to keep the mesh conforming. However, when using the quadrilateral and hexahedral elements that are necessary for tensor-product matrix-free methods, this approach is not applicable, since simple subdivision can never yield a conforming mesh. Instead, for such meshes, a non-conforming refinement strategy is often used,

where *hanging nodes* are allowed. These are nodes on an edge which only belong to one of the elements sharing the edge (see Figure 2.5).

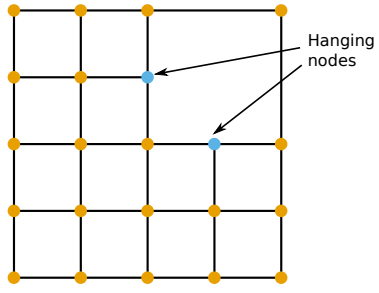


Figure 2.5. Hanging nodes in an adaptively refined mesh are node points on fine elements which lie on an edge shared with a coarser element. The two blue points are hanging nodes on the fine elements surrounding the coarse one, and are constrained to the two nodes of the corresponding coarse face.

In order to enforce the standard smoothness properties of the solution, the unknowns located on these nodes are not actually independent, but must fulfill some linear constraint coupling them to other unknowns. In this thesis, we consider the common case where cells with a common vertex differ in refinement level by at most one, which prevents hanging nodes to be coupled to other hanging nodes. See, e.g., Section 3.3 in [6] for more details on hanging-node constraints. Figure 2.6 shows an example from Paper III of a highly non-uniform mesh resulting from refinement along thin spherical shells.

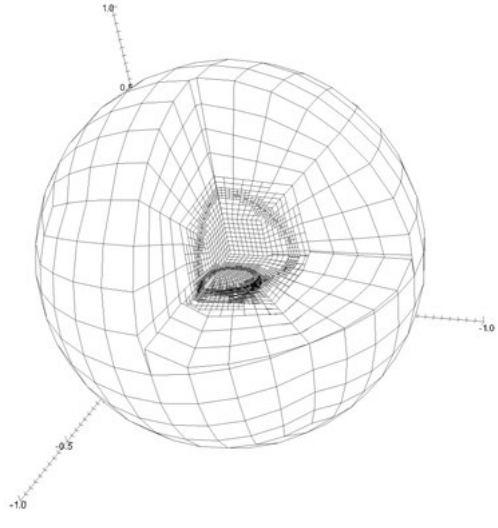


Figure 2.6. A mesh where an initial uniform spherical mesh has been repeatedly refined along a number of non-aligned spherical shells yielding a mesh where about 30% of the DoFs are constrained.

For matrix-based methods, the standard way of treating hanging node constraints is to *condense* the constrained values away from the system and add their contribution to the rows and columns of the DoFs to which they are constrained, yielding a new slightly smaller linear system only involving non-constrained DoFs. While it is conceptually simple to first perform the assembly as usual, and then eliminate rows and columns of constrained DoFs from the system, it is more efficient to eliminate these on the fly during the assembly itself (see Section 3.3.1 in [6]).

With the matrix-free approach, one cannot eliminate the constrained unknowns from the mesh since the sum-factorization evaluation requires each element to have a full set of DoFs. Instead, these constraints must be applied every time the affected unknowns are accessed (see e.g. [44]). In effect, this modifies the matrix-free algorithm in (2.11) by adding a matrix C_k taking care of resolving the constraints yielding

$$\begin{aligned} &\text{(a)} \quad v_k = C_k P_k^T v\,, \\ &\text{(b)} \quad u_k = a^k v_k\,, \\ &\text{(c)} \quad u = \textstyle\sum_{k\in\mathcal{K}} P_k C_k^T u_k\,. \end{aligned} \tag{2.26}$$

In Paper III, we are performing matrix-free computations on meshes with hanging nodes, and in Paper IV we extend these to a full multigrid algorithm.

2.5 Multigrid

One of the most popular techniques for solving the linear system arising from a finite-element discretization is the multigrid method. This algorithm exploits the fact that several frequency components of the error can be attacked simultaneously by using a hierarchy of grids, yielding a method where the computational work only grows linearly with the size of the problem [70].

The multigrid procedure works as follows. First, an approximate solution is obtained at level l using some iterative scheme, usually referred to as a *smoother*, which is efficient at removing high-frequency errors at level l. Then a corresponding residual is computed, which is then transferred to the next coarser level $l-1$ using a coarsening operator usually referred to as a *restriction*. Next, the same scheme is applied on level $l-1$, where the coarser grid makes the smoother remove other error components of lower frequency. This procedure is applied recursively for all levels until the coarsest level $l=0$ is reached, where the problem is small enough for an exact solution to be found. Finally, the contributions from each level are then transferred back up the hierarchy to finer levels using a *prolongation* operator, often combined with an additional application of the smoother on each level. When reaching the finest level, what is usually called a multigrid *V-cycle* is concluded, and a final global solution approximation has been obtained.

To obtain the hierarchy of level meshes, a successively refined mesh is usually assumed, where the levels are constructed by iteratively refining a coarse base level mesh to generate each new level in the hierarchy. Since each level constitutes a physical grid, the representation of the system matrix on each level is simply the corresponding differential operator discretization for that grid. In addition to such a mesh-based *Geometric Multigrid* approach, a more general matrix-based *Algebraic Multigrid* method has also been proposed where the prolongation and restriction operators are used to define system matrices for any coarser solution representation [67, 70]. However, since these methods require explicit knowledge of matrix entries for generation of the coarser-level operators, these cannot be used in general within a matrix-free framework.

When using adaptively refined meshes with hanging nodes, there are two main ways to apply smoothing to the individual levels. In the first case, which is referred to as *global smoothing*, each level grid is considered to cover the whole domain, meaning that it not only contains cells at that level, but also cells from coarser levels where fine-level cells are absent [10]. In the second case, each level grid is instead restricted to only the cells at that refinement level, yielding a *local smoothing* approach. In Paper IV, the local smoothing approach is used since no hanging node constraints need to be resolved when applying the smoother and the somewhat fewer operations involved [14].

When applying multigrid together with the matrix-free method for applying the finite-element operator, only smoothers which do not require explicit knowledge of matrix entries can be used, ruling out incomplete LU-factorizations or Gauss-Seidel methods that are otherwise popular. Instead, matrix-free versions of the multigrid method typically rely on Jacobi-type smoothers. Such methods only involve matrix-vector products with the system matrix, and possibly multiplication by the inverse diagonal of the system matrix, which can be readily be computed in a matrix-free manner. In Paper IV, we employ a popular variant of such smoothers called the Chebyshev smoother, where initially, a conjugate gradient iteration is applied to approximate the eigenvalues of the matrix, which are then used to construct an optimal polynomial combination of matrix-vector products for the smoother [2].

Finally, in practice, it is common to apply multigrid as a preconditioner in an iterative solver for additional robustness of the solver. This is the approach taken in Paper IV, where multigrid is used in combination with a conjugate gradient solver.

2.5.1 Numerical Experiments

In Paper IV, we apply a matrix-free multigrid implementation to three different variations of Poisson's equation. We first consider a 2D L-shaped domain with adaptive refinement which is interesting because of the highly localized error

resulting from the singularity at the boundary. We employ refinement policy where the cells with the 30% largest error are marked for refinement, and the cells with the 3% smallest error are marked for coarsening, leading to roughly a doubling of the number of unknowns per refinement cycle. An example of the the resulting mesh can be seen in Figure 2.7.

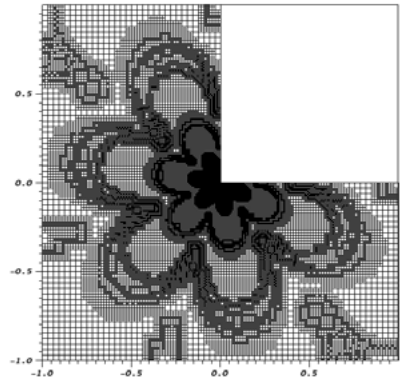


Figure 2.7. Mesh for a 2D L-shaped domain produced by four adaptive refinement cycles.

In Table 2.1, we see that the number of conjugate-gradient iterations are independent of the problem size meaning that a constant number of fine-level matrix-vector products will be performed. Since the computational complexity of the matrix-free operator is proportional to the problem size, this yields the expected linear scaling of work with problem size. Furthermore, we see that there is a reduction in the error by roughly a factor of 2.6 for each doubling of the problem size.

Table 2.1. *Convergence with iterative adaptive refinement when solving Poisson's equation on an L-shaped domain in 2D using second-order elements.*

Cells	DoFs	CG iterations	L_2 error
3.0k	13k	5	1.13e-04
5.8k	24k	7	4.41e-05
11k	47k	7	1.73e-05
21k	94k	7	6.83e-06
41k	0.18M	7	2.70e-06
78k	0.35M	7	1.07e-06
0.15M	0.68M	7	4.23e-07
0.29M	1.3M	7	1.68e-07
0.55M	2.5M	7	6.65e-08
1.1M	4.8M	7	2.65e-08
2.0M	9.4M	7	1.06e-08
3.9M	18M	7	4.25e-09

Secondly, we solve a Poisson problem on a spherical shell domain, with a strongly varying coefficient defined by

$$A(\mathbf{x}) = 1 + 10^6 \prod_{e=1}^{d} \cos^2(2\pi x_e + 0.1e),$$

and a solution depicted in Figure 2.8.

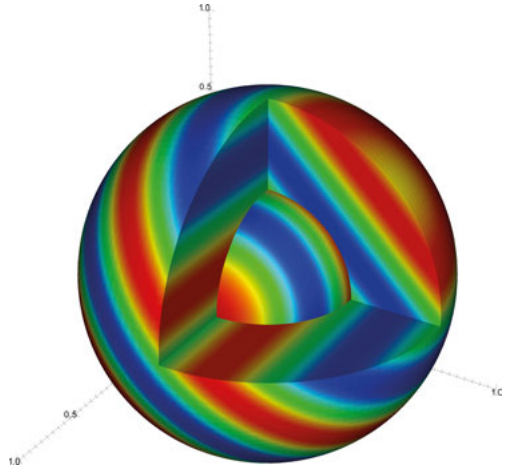


Figure 2.8. Solution of Poisson's equation on spherical shell for a mesh with 0.2 million second-order elements (with one octant removed).

In Table 2.2, which shows the convergence of the multigrid solver for the spherical shell problem, we see that now the error decays by a factor of about 1.5 which each doubling of the problem size. This is lower than the factor for the L-shaped domain, which is expected both due to the higher dimensionality where more points are needed for an equivalent level of resolution, but also due to the fact that adaptive refinement uses the unknowns more efficiently.

Table 2.2. *Convergence with uniform refinement when solving Poisson's equation on a spherical shell using second order elements.*

Cells	DoFs	L_2 error	Iterations
384	3474	1.880e-01	10
3072	26146	4.933e-02	11
24576	202818	1.403e-02	13
196608	1597570	3.907e-03	14
1572864	12681474	1.043e-03	15

Also, we see that in contrast to the L-domain problem, the number of multigrid iterations are not constant with mesh refinement. The explanation for this is that optimal convergence of the multigrid algorithm depends on the coefficient to be well resolved on all levels. As we refine the mesh, the difference between coefficient on the finest and coarsest levels grows.

In the final example, we consider the Minimal Surface Equation,

$$\begin{aligned} -\nabla \cdot \left(\frac{1}{\sqrt{1+|\nabla u|^2}} \nabla u \right) &= 0 \quad \text{in } \Omega, \\ u &= g \quad \text{on } \partial\Omega, \end{aligned} \tag{2.27}$$

which is a nonlinear version of Poisson's equation describing the surface of a soap film spanned a by a wire loop [62, Ch. 10]. We tackle the nonlinearity using Newton's iteration in combination with a backtracking line search method, yielding solutions such as the one in Figure 2.9.

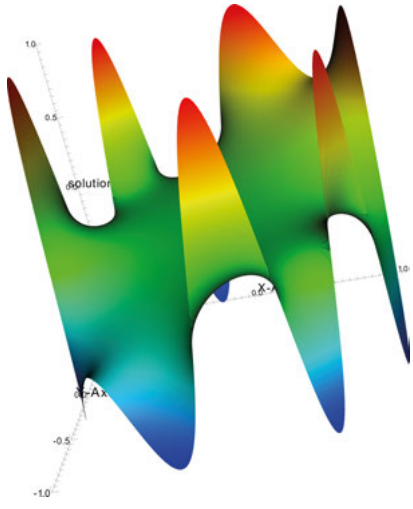


Figure 2.9. Solution of the Minimal Surface Equation for a mesh with 80 000 $\mathcal{Q}_4$ elements.

2.5.2 Mixed Precision

In all numerical computations, cancellation, rounding, and representation errors due to the finite precision of floating point numbers lead to a more or less imprecise result. While the size of each individual error is determined by the number of decimals used, the size of the overall error is decided by how these are amplified by the computation under consideration.

In [28], Göddeke and collaborators show that when solving linear systems from finite element discretizations, double precision arithmetic must be used in order for the arithmetic to be accurate enough to produce a satisfactory solution. In fact, if single precision is used, not only does the number of linear iterations grow more than expected with mesh refinement, but the convergence control itself fails due to inaccurate approximations of the residual norm.

In Paper IV we consider a mixed-precision approach where a single-precision multigrid preconditioner is combined with double precision for the outer

conjugate gradient iteration. As can be seen in Table 2.3, neither the accuracy nor the number of iterations are affected compared to if double precision is used exclusively. Also see Section 3.5.2, where we return to mixed precision algorithms in the context of graphics processors, for which lower-precision arithmetic can have several performance benefits.

Table 2.3. *Convergence for mixed and full double precision in multigrid solve of the Spherical shell Poisson problem with rapidly variable coefficient from Paper IV.*

				Double		Mixed	
		Cells	DoFs	L_2 Error	Iterations	L_2 Error	Iterations
2D	$\mathcal{Q}_1$	10M	10M	3.31e-06	8	3.31e-06	8
	$\mathcal{Q}_3$	2.6M	2.4M	1.29e-06	8	1.29e-06	8
3D	$\mathcal{Q}_1$	1.6M	1.6M	2.89e-03	14	2.89e-03	14
	$\mathcal{Q}_3$	0.20M	5.4M	4.07e-03	16	4.07e-03	16

3. Finite Element Methods on 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 [54]. 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 [33]. 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 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 at any given time.

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

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		$x \leftarrow x_2$

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 one indivisible entity, which can only appear to the memory as a whole, then it can safely be executed concurrently. For simple critical sections, such as 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 sections and are limited to the available atomic instructions. 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 [34].

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 lock-based technique, where we always perform the locking even if no actual conflicts occurred, thus being a more *pessimistic* approach. This optimistic approach of transactional memory 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. With transactions, on the other hand, 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 [34]. In the following years, several studies investigating possible transactional-memory implementations in hardware or software were published [3, 66, 12, 49]. Around 2008, Sun Microsystems announced the Rock processor; the first major processor intended for commercial use to feature transactional memory in hardware [18]. 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 [31], whereas Intel has introduced transactional memory in the form of the Transactional Synchronization Extensions in their Haswell processor series [36]. For both of these systems, relatively good performance when applied to scientific computing has been demonstrated [46, 74].

The Rock Processor

All the experiments in Paper I were performed on 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 [18].

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.

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 [21].*

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

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 failure 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 information in [22]. 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 [22], 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.

Listing 3.1. *Strategy for handling failed transactions*

```
1 while transaction fails:
2   if cps == conflict:
3     back-off
4   else if cps == store:
5     compare-and-swap data
6   else if cps == load:
7     load data
8   retry transaction
```

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

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.*

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%

3.3.1 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 conversions 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. 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.

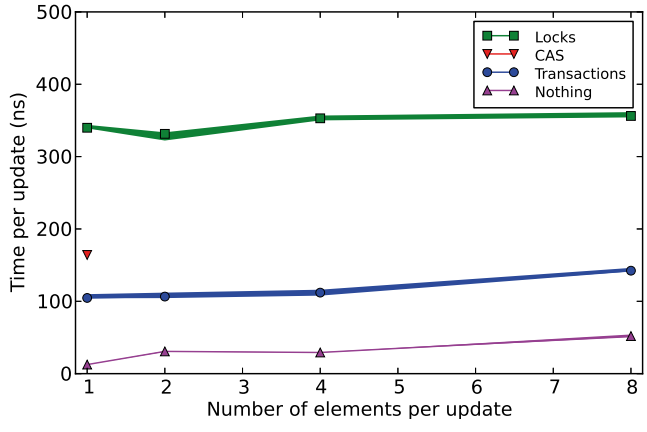


Figure 3.1. Result of the **overhead** benchmark. The width of the lines shows the variation from 100 runs, measured as lower to upper quartile.

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`.

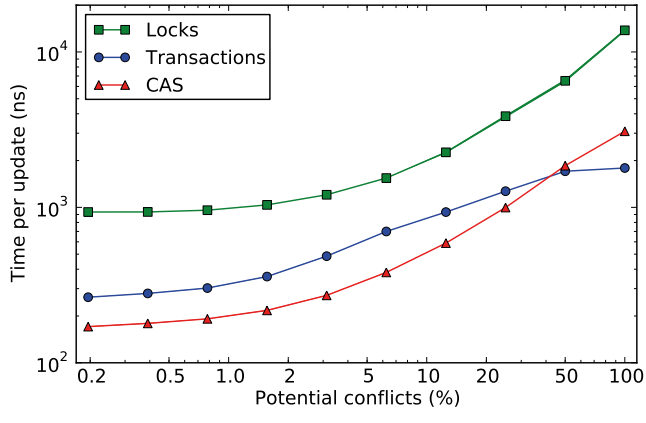


Figure 3.2. Results of the **contention** benchmark, presented as the median of 100 runs. The (barely visible) variable width of the lines shows the variation from lower to upper quartile.

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 possible (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.

3.3.2 FEM Assembly Experiment

In Paper I, we study the performance of the transactional memory system of the Rock processor for performing floating point updates in the assembly of a finite-element matrix. 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 (2.8). 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 affected 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 [35].

Figure 3.3 shows the results of the finite-element assembly benchmark. 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 – a factor of 3.3 at best for the compare-and-swap implementation – which can be explained by the fact that the assembly algorithm is memory bound.

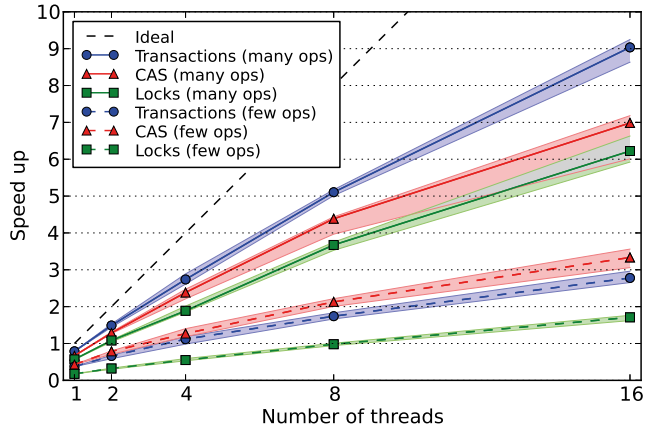


Figure 3.3. Median speedup over serial for the different finite-element assembly approaches. The shaded areas show the variation from lower to upper quartile.

Considering the amount of aborted transactions (Table 3.3), we see that there was a relatively high amount of aborts – higher than in any of the microbenchmarks. However, looking at the distribution of causes, we see that there was in fact almost no conflicts. Rather, the brunt of the failures 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.

Conclusion

In summary, we conclude that transactions were faster than both locks for both versions of the assembly, and faster than compare-and-swap when updates are less frequent. Furthermore, in the compute-intensive version of the matrix assembly, we saw that for certain applications, transactions allows for very naive algorithms to be used with a decent speedup. 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 2017, the processing power (Gflop/s) of Intel processors increased by about 35% annually, whereas corresponding increase in the bandwidth (GB/s per socket) was only about 13%.

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 3.4, 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.

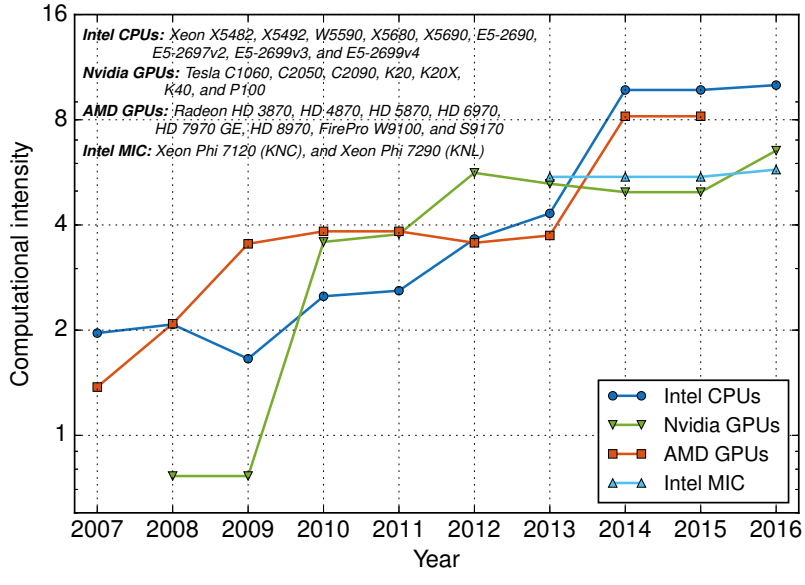


Figure 3.4. Change in computational intensity for processors during the last six years for double precision operations. Graph based on a blog post by Karl Rupp [64].

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 nearly expect to reap the full benefit of new architectures.

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 much larger for many GPUs. For instance, for most new consumer graphics cards, double precision is a full 24 times slower than single precision. 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.

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.*

Processor	Cores	Gflops	GB/s	Gflops/W	Gflops/$
Intel Core i7 6950X	10	480 (960)	77	3.4 (6.9)	0.28 (0.56)
Intel Xeon E5-2699v4	22	774 (1 549)	77	5.3 (10.7)	0.19 (0.38)
Nvidia Titan Xp	3 840	703 (11 244)	548	2.8 (45)	0.59 (9.4)
Nvidia Tesla K40	2 880	1 431 (4 291)	288	6.1 (18)	0.26 (0.78)
Nvidia Tesla P100	3 584	4 036 (8 071)	720	16 (32)	0.71 (1.4)
AMD FirePro S9150	2 816	2 534 (5 069)	320	11 (22)	0.77 (1.5)
Intel Xeon Phi 7290	72	3 456 (6 912)	600	14 (28.2)	0.55 (1.1)

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, not all 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 [47], a solution of the compressible Navier-Stokes equations [45], and a multigrid solver [13]. A summary of early work in GPGPU can be found in the survey paper by Owens et al. [60]. However, the many restrictions and the fact that a programming model originally intended 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, each executed on a single SM in groups of 32 threads called *warps*, independently of the other blocks. Within a thread block or an SM, there is a piece of shared memory and also a small L1 cache. 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 at kernel launch boundaries. In CUDA, there are atomic intrinsics similar to the compare-and-swap operation discussed in Section 3.2. On earlier generations of Nvidia GPUs, atomic operations came with a substantial overhead, but on the Kepler architecture and later, these are performed by a single instruction with a "fire-and-forget" semantic. This means that the instruction returns immediately and conflict resolution is taken care of by the cache system (explained in detail in Section 5.2.11 in [73]). Furthermore, these fully native atomic operations were previously limited to single precision arithmetic, but on the recent Pascal architecture, support is added also for double-precision values [56].

Since the total cache size of GPUs is much smaller than for CPUs, it is typically not enough to hide the latency to the main GPU memory. Instead the GPU relies on massive parallelism for this, where new parallel threads can be switched in while waiting for long-latency instructions such as memory accesses to complete. The higher the number of resident thread warps in flight per SM or the *occupancy*, the better are the possibilities for the scheduler to hide the latency.

An important feature of CUDA, and arguably the most crucial aspect to taken into account 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 [58]. Examples of fields where CUDA has been successfully utilized include molecular dynamics simulations [4], fluid dynamics [25], wave propagation [53], sequence alignment [50] and, Monte Carlo simulations of ferromagnetic lattices [63].

In response to CUDA and the popularity of GPU programming, OpenCL was launched by the consortium Khronos Group in 2008 [68]. 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 [37, 26].

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. How-

ever, 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 manycore processors, and the 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 [13, 30, 19, 23]. 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 [61], 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 [17], 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 [24]. Markall et al. study what implementations of FEM assembly are appropriate for many-core processors such as GPUs compared to multi-core CPUs [51].

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 high-order solution of elliptic and parabolic PDEs. In [42], Komatitsch et al. study a high-order spectral-element method applied to a seismic simulation on a single GPU. In [41], 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 [38]. 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.

For linear simplex elements $\mathcal{P}_1$, where the shape function gradients are constant within each element, it is possible to factor out all the element-dependent data out of local-element integration, leading to a generalized version of the local-matrix approach valid for general meshes. In [39], Knepley et al. apply this approach on GPUs within the FEniCS library.

3.5.2 Matrix-Free Finite Element Experiments

In Papers II-IV, we explore matrix-free finite-element methods for numerical solution of PDEs on graphics processors. We consider a finite-element discretization of Poisson's equation, which provides a simple model problem, while at the same time is representative of more complex problems as it captures many of the difficulties that can arise there. First, in Paper II, we consider the case of a Cartesian mesh and study the performance of a matrix-free evaluation of the Laplace operator acting on a vector. In Paper III, we generalize the set up to include general meshes and problems with variable coefficients, and develop a matrix-free implementation of the Laplace operator evaluation based on sum factorization. Finally, Paper IV extends these results to a full PDE solver using multigrid preconditioning.

As discussed in Section 2.1, the matrix-vector product constitutes the majority of the computational work in an iterative linear solver. Therefore, we initially consider only this operation and its performance when executed on a GPU.

Operator Evaluation with Local Matrix

As mentioned in Section 2.3.1, when the mesh is uniform and no variable coefficient is used, the local matrices for each element are identical and a single copy can be precomputed and used during the operator evaluation, with a very small memory footprint as a consequence (see Figure 2.2). This approach is evaluated in Paper II, were we study the performance of a CUDA parallelization of the local-matrix algorithm. We introduce CUDA blocks of 256 threads and let each thread process one element each. Two strategies for how to safely perform the shared-variable updates in the reduction loop are considered. The first one is based on the graph coloring method described in Section 2.3.3, where the Cartesian mesh allows for a trivial coloring of elements. The second version uses CUDA atomic intrinsics operations.

Our implementation of the matrix-free algorithm is contrasted to a matrix-based version where the multiplication implementation is provided by CUSPARSE, the official sparse-matrix library of Nvidia [55]. The matrix is stored

in the *compressed sparse row* (CSR) matrix format, since this is efficient for sparse matrix-vector products and other row-wise operations. We also explored using the hybrid (HYB) format [9], but found that this did not perform better than the CSR format for our case.

Since the focus here is on the Laplace operator evaluation as it would appear in an iterative solver, we only benchmark this operation, and exclude any time taken for set up, since this can be done once initially. To avoid noisy time measurements, we take the minimum time of 20 runs of 20 operator applications. The benchmark computer system features an Nvidia Tesla K20 GPU, an eight-core Intel Xeon E5-2680, and 64 GB DRAM, and runs Linux 2.6.32, with GCC 4.4 and CUDA 5.5. All the experiments are for double precision arithmetic.

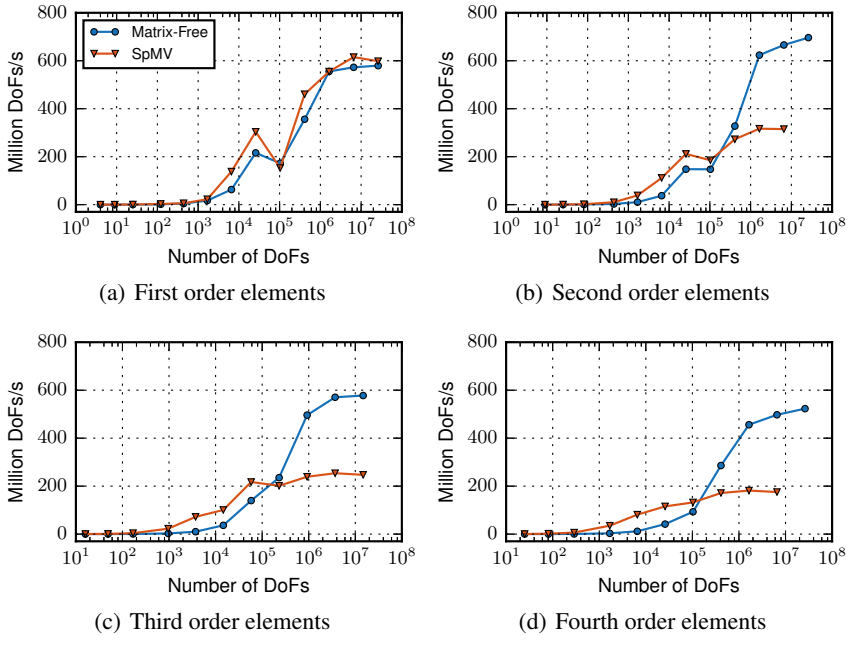
In Figures 3.5 and 3.6 we see the throughput in million unknowns per second of the matrix-free and matrix-based algorithms, and how this scales with the problem size for a Cartesian mesh set up in 2D and 3D, respectively. Looking at Figure 3.5, we firstly see that performance increases with problem size since the GPU needs enough parallelism to be fully utilized. Secondly, we see that the matrix-based algorithm falls behind for elements of order two and higher, due to its bandwidth usage which increases rapidly with element order (see Figure 2.2), with the matrix-free algorithm being about two times faster overall. Furthermore, we see that for $\mathcal{Q}_2$ and $\mathcal{Q}_4$ elements, the matrix-based algorithm could not fit the matrix in memory as indicated by the one missing data point for those curves.

Looking at the 3D results in Figure 3.6 and comparing to the 2D case in Figure 3.5, we see that the situation looks similar for first and second-order elements, with the matrix-free algorithm coming out ahead with a speedup of about two. However, this pattern is broken for third and fourth order elements in 3D, where we see a large drop in the performance of the matrix-free version. The reason for this can be seen if we consider the number of elements in the local matrix which for elements of order p in d dimensions is equal to $(p+1)^{2d}$. The resulting number of entries for different finite-element configurations are listed in Table 3.5.

Table 3.5. *Number of entries in the local matrix for various element configurations.*

	$\mathcal{Q}_1$	$\mathcal{Q}_2$	$\mathcal{Q}_3$	$\mathcal{Q}_4$
2D	16	81	256	625
3D	64	729	4096	15625

If we compare these numbers the size of the L1 cache of the Tesla K20, which is equal to 32 kB or 4096 doubles, it is clear that for third-order elements in 3D, this can no longer accommodate the local matrix, if the additional contribution from various other data structures like index mappings are taken into account. This means that the threads in a block can no longer share the local matrix in the cache, implying that the actual bandwidth footprint far ex-



(a) First order elements

(b) Second order elements

(c) Third order elements

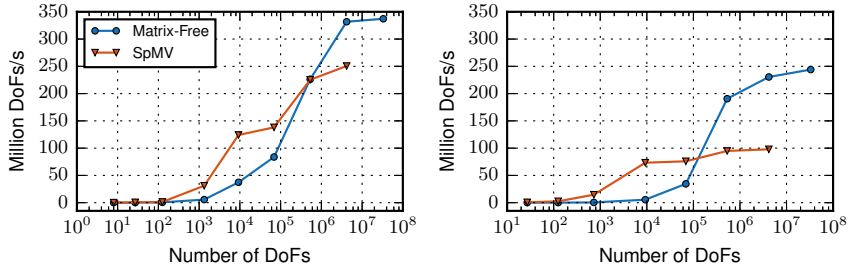
(d) Fourth order elements

Figure 3.5. Performance versus problem size for the 2D experiments in Paper II.

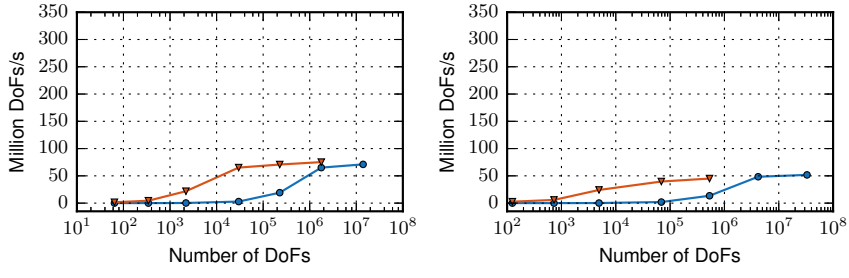
ceeds the theoretical ideal bandwidth, explaining the decrease in performance. Finally, we see that as for the 2D experiments, the matrix-based algorithm had trouble fitting the largest problems in memory. In fact, for fourth order elements, almost two magnitudes larger problems could be solved with the matrix-free approach.

In Figure 3.7, we see a comparison of the performance of the atomics and coloring methods for handling updates of shared variables, for the largest problems solved. Here, we also include a third version "Ideal", without any protection yielding an estimate of the ideal performance.

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 ideal reference version. For the coloring version, the overhead is roughly 15% for all configurations, except for $\mathcal{Q}_2$ elements in 2D where the Ideal version shows extraordinarily high performance with an overhead of about 24% as a consequence. On the other hand, for the version using atomics, the overhead is much more inconsistent and varies between 11% and 28% for most configurations. The exception to this is for the two configurations which are affected by the caching problem described above, i.e. third- and fourth-order elements in 3D, where overhead is negligible. The explanation for this is that the slower execution hides the overhead of the atomic operations by making the updates less frequent. The coloring



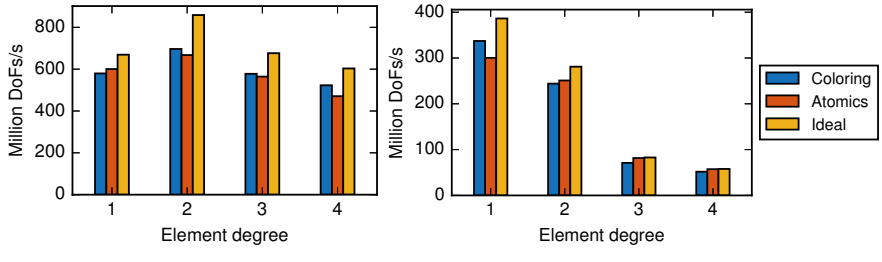
(a) First order elements (b) Second order elements



(c) Third order elements (d) Fourth order elements

Figure 3.6. Performance versus problem size for the 3D experiments in Paper II.

approach on the other still has to process the colors separately, and thus still have roughly the same overhead as for the other configurations.



(a) 2D (b) 3D

Figure 3.7. Performance for the largest problems solved using coloring, atomics, or no protection for the shared-variable updates.

Operator Evaluation using Sum-Factorization

Motivated by the promising performance results for the local-matrix approach in Paper II, we next consider the fully matrix-free method based on sum-factorization described in Section 2.3.2. This method is applicable for a much wider range of cases, allowing for variable coefficients, curved geometry, and non-uniformly refined mesh. It also has the potential of providing more robust performance since even if it uses more distinct memory in total (see Fig-

ure 2.2), each thread reads less data so it does not rely as heavily on caching of large data structures.

In Paper III, we present a GPU parallelization of the sum-factorization algorithm in CUDA based on the `deal.II` framework [5]. One of the key differences to the implementation in Paper II using a local matrix is that the sum factorization approach needs to store a considerable amount of temporary data for each element. During the evaluation of the local operator, the gradient at each quadrature point needs to be saved, which amounts to $d(p+1)^d$ values for $\mathcal{Q}_p$ elements in d dimensions. This is an issue since using a high number of registers for each CUDA thread will lead to a severe limitation on the occupancy, i.e. number of resident threads per streaming multiprocessor. Because of this, we opt for a different parallelization compared to for the local-matrix method. As we saw in Section 2.3.2, the sum-factorization technique leads to a series of dense tensor contractions of tensors of size $(p+1)^d$. Since these are essentially small dense matrix-matrix multiplications and can be parallelized over the entries in the tensor, we introduce one CUDA thread per local unknown and assign each element to one CUDA block. We then let the threads in the block cooperate in performing the tensor reductions using CUDA shared memory. Also, the threads can cooperate in reading the unknowns from the global vector, and in writing them back after the evaluation.

For optimal utilization of bandwidth to global memory, we layout the per-element data structures in a *structure-of-array* format, so that, e.g., the Jacobian are stored one component at a time. For the data structures that are common to all elements, such as the value and gradients of one-dimensional shape function at quadrature points – referred to as ψ_μ^α and χ_μ^α in Section 2.3.2 – we utilize CUDA *constant* memory in order to leverage the dedicated cache for warp-broadcast read-only access. Finally, we use non-caching loads using the `__ldg` function for reading the local DoF values from the global vector since these have a very irregular pattern and cannot be expected to be coalesced.

Just as in Paper II, we consider both the mesh coloring approach and the one atomic operations for performing the reduction of element contributions back to the global vector. In this case, the same straightforward coloring which was used for the Cartesian mesh cannot be used, and a full graph-coloring algorithm is necessary. We use the one available in `deal.II` described in Section 2.3.3.

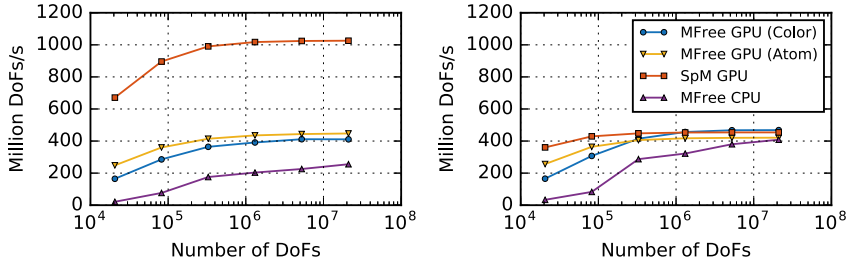
To evaluate the performance of our method, we run a benchmark similar to the one used in Paper II, where we repeatedly evaluate the Laplace operator of a Poisson-type problem. We use a variable coefficient defined by

$$A(\mathbf{x}) = \frac{1}{(0.05 + 2\|\mathbf{x}\|^2)}$$

and a unit hyper-ball domain in 2D and 3D. In the first benchmark, we consider a uniformly refined mesh which is created by repeatedly refining a coarse base obtained using the `GridGenerator::hyper_ball` function in `deal.II`.

We compare our GPU implementation of the matrix-free sum-factorization method to two competitors; one version corresponding CPU implementation of the same method, and one version for the GPU using an explicit sparse matrix. The matrix-free CPU reference is a highly optimized implementation based on previous work by Kronbichler et al. [44], and is parallelized using a combination of Intel Threading Building Blocks and vector intrinsics. Just as in Paper III, we use Nvidia CUSPARSE in the sparse-matrix version for GPU [55].

We use two different computer systems for the experiments; a system with an Nvidia Tesla K40, an Intel Core i5-3550 quad-core, 16 GB DDR3 RAM, and CUDA 8 RC for the GPU experiments, and a system with two Intel Xeon E5-2680 eight-core processors, and 64 GB DDR3 RAM for the CPU experiments. It should be stressed that the K40 GPU and the two Xeon processors have a comparable power consumption, with a TDP of 235 W for the K40, and a combined TDP of 260 W for the CPUs. Thus, a comparison of performance can be considered fair.



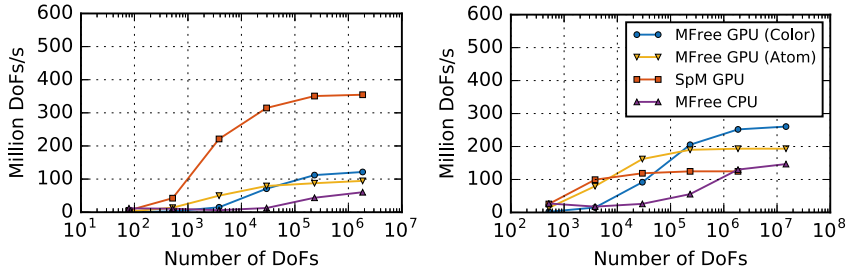
(a) First order elements (b) Second order elements

(c) Third order elements (d) Fourth order elements

Figure 3.8. Throughput vs problem size for the 2D uniformly refined mesh.

In Figures 3.8 and 3.9, we see the results in 2D and 3D, respectively, again reported as throughput in number of unknowns per time. As expected from the discussion in Section 2.3.1, the matrix-based algorithm is more efficient for first order elements since it uses less data in that case (see Figure 2.2). On the other hand, for elements of order two and higher, the matrix-free method (using coloring) is faster than the sparse-matrix method, and with speedups

of between two and eight in all cases, except for $\mathcal{Q}_2$ elements in 2D, where the speedup was negligible. Furthermore, when comparing the CPU and GPU versions of the matrix-free algorithm, we see that the GPU is always faster; up to 61% faster in 2D, and up to more than two times faster in 3D.



(a) First order elements

(b) Second order elements

(c) Third order elements

(d) Fourth order elements

Figure 3.9. Throughput vs problem size for the 3D uniformly refined mesh.

Next, if we compare the performance of version using the mesh coloring to the one using atomic operations, we see that except for $\mathcal{Q}_1$ elements in 2D, the coloring approach is faster for large problems. However, we see that atomics version is faster for smaller problem sizes, especially in 3D, which can be explained by the fact that the coloring approach has less parallelism since only the elements of one color at a time are processed in parallel, and thus needs more elements to be fully entertained.

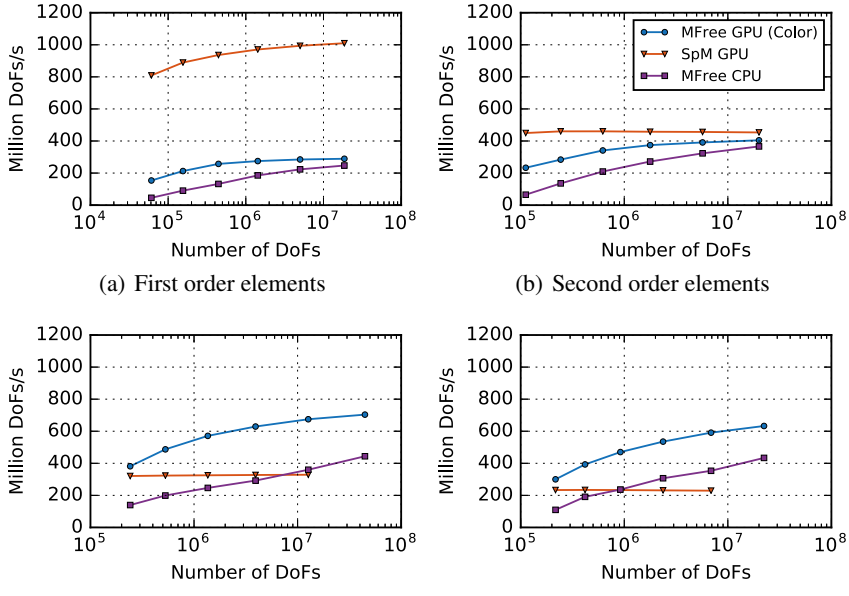
Lastly, we again see that the implementation using an explicit sparse matrix cannot fit the largest problems in memory for elements of order three and higher in 2D, and for elements of order two and higher in 3D. The effect is not as strong as for the local-matrix approach, since the tensor-product approach needs to store considerably more data due to Jacobian mappings, especially for non-Cartesian meshes.

Resolving Hanging Node Constraints

In Paper III, we also introduce a novel method for resolving hanging-node constraints on the GPU. As noted in Section 3 of Paper III, the fine-grained parallelization used for the sum-factorization approach prohibits resolving the

hanging node constraints one by one in serial. To utilize the existing parallel threads as much as possible, we instead let all threads on a given constrained face or edge cooperate in resolving all the corresponding constraints in parallel. Furthermore, since the constraints on a face in 3D has the same tensor-product structure as the in the local operator evaluation, the same sum-factorization method can be used there too.

To evaluate the performance of the method for resolving hanging-node constraints, we rerun the benchmarks from before on a non-uniformly refined mesh. By taking the same coarse initial hyper-ball mesh from the uniform case, and refining it along a series of non-aligned spherical shells, we generate a mesh with highly localized refined regions of elements. Using this method, we obtain meshes of similar size to the uniform ones, with about 3–5% hanging nodes in 2D, and about 20–38% hanging nodes in 3D, of which an example can be seen in Figure 2.6 in Section 2.4.



(a) First order elements

(b) Second order elements

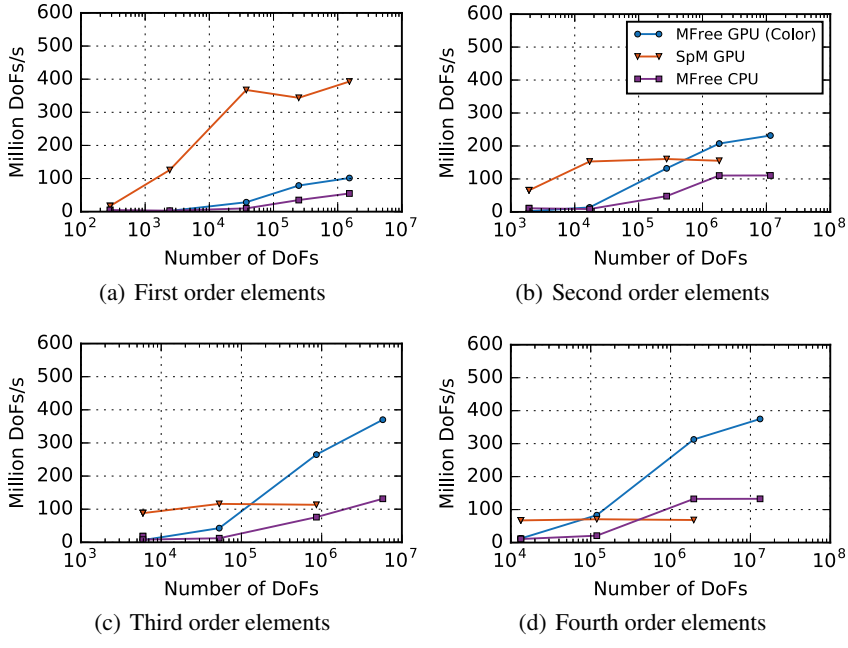
(c) Third order elements

(d) Fourth order elements

Figure 3.10. Throughput vs problem size for the 2D adaptively refined mesh.

The results can be seen in Figures 3.10 and 3.11, where we first note that there is a slight drop in performance for the matrix-free methods compared to in the experiments with uniform meshes in Figures 3.8 and 3.9. For the GPU version the overhead of the hanging-node treatment is about 10–40%, whereas the CPU version had a higher overhead reaching up to 70% for some meshes. On the other hand, the matrix-based version performs identically as expected since all hanging nodes are eliminated once at assembly. However, the matrix-free algorithm is still faster for third order and higher in 2D and

second order and higher in 3D, due to the much more efficient algorithm. All in all, the matrix-free approach is at least twice as fast for third order elements and higher in 2D, and at least three times faster for third order elements in 3D.



(a) First order elements (b) Second order elements

(c) Third order elements (d) Fourth order elements

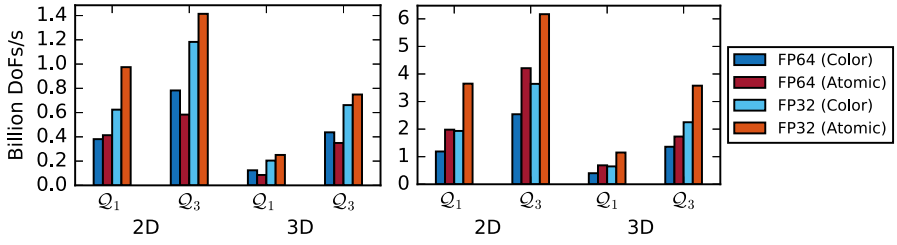
Figure 3.11. Throughput vs problem size for the 3D adaptively refined mesh.

Performance Impact of Floating-Point Precision

In Paper IV, we briefly revisit the matrix-free Laplace operator evaluation. Motivated by the observations concerning the performance of single and double precision arithmetic on graphics processors in Section 3.5, as well as the discussion on the viability of a mixed precision approach for multigrid methods in Section 2.5.2, we study how performance changes with the precision of the floating-point numbers, and also how this has changed on the most recent GPU hardware.

We consider a similar setup to the one in Paper III; the variable coefficient spherical shell problem described in Section 2.5.1. In Figure 3.12(a), we see the performance of a matrix-free Laplacian evaluation for 32-bit and 64-bit arithmetic when executed on an Nvidia Tesla K40. Just like we saw previously, the coloring approach is faster than atomics when using 64-bit numbers, except for first order elements in 2D. However, this changes drastically when going to 32-bit arithmetic where the version using atomics is consistently faster, due to the fact that the single-precision `atomicAdd` is performed in a single instruction. Also, we see that there is a solid improvement in performance of the single-precision version over the one using double precision. When using

atomics, the speedup is between two and three, whereas the speedup for the coloring version is about 50–60%.



(a) Tesla K40 (b) Tesla P100

Figure 3.12. Performance in throughput of unknowns per time for mesh coloring and atomics approaches, when using single and double precision numbers. The meshes contained 10–24 million DoFs in 2D, and 1.6–5.4 million DoFs in 3D. Results for a) the previous Kepler generation GPU, and b) the current Pascal generation GPU.

Looking at Figure 3.12(b), where we have rerun the benchmarks on an Nvidia Tesla P100 from the new Pascal architecture, we see that the version using atomics is now faster also for double precision, which is due to the added support for single-instruction `atomicAdd` also for 64-bit floating point numbers. Due to this, the speedup of single precision over double precision is reduced slightly for that case, yielding speedups of between 40-65% for the version using mesh coloring, and 47% to a factor of two when using atomic operations.

Solution of Poisson's Equation using Multigrid

Next, we consider the performance of the full multigrid-based iterative solution of Poisson's equation discussed in Section 2.5, when executed entirely on the GPU. In Paper IV, we develop a CUDA implementation of the building blocks of a multigrid method. For the prolongation and restriction grid-transfer operations, we apply the same sum-factorization method as was used for the evaluation of the local operator and for resolving hanging nodes. This is possible since the interpolation between coarse and fine cells have the same tensor-product structure as the other operations. One important difference, however, is that the number of inputs and outputs of the grid transfer interpolation is not equal; there are $(p+1)^d$ local DoFs on the coarse level and $(2p+1)^d$ local DoFs on the fine level. We solve this by using one thread per coarse DoF, and let each thread be responsible for multiple fine-level DoFs.

In addition to the evaluation of the Laplace operator and the prolongation and restriction operations, a full iterative solver also requires implementation of vector operations such as element-wise vector updates and the scalar product. Here, rather than a straightforward parallelization with one thread per vector entry, we let each thread handle a chunk of eight entries in order to utilize the ILP of the streaming multiprocessor which can only issue two inde-

pendent instructions for each group of four warps per cycle [72, 57]. A CUDA block size of 512 was found to yield the best performance. The parallel reduction of the scalar product is performed in shared memory for each block, using the techniques discussed in Harris [32].

We first solve the spherical shell problem described in Section 2.5.1. Based on the promising single-precision performance of the Laplace operator, we first compare the performance of a mixed-precision solver using single-precision arithmetic for the multigrid preconditioner and double precision for the outer conjugate gradient iteration, to one which only uses double precision. Since atomic operations was found to perform better than coloring for all configurations on the Tesla P100, we only consider that case for the remainder of the experiments. Looking at Figure 3.13, we see that the mixed precision version is about 1.5–1.8 times faster than the version using double precision exclusively.

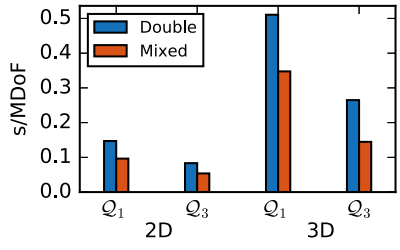
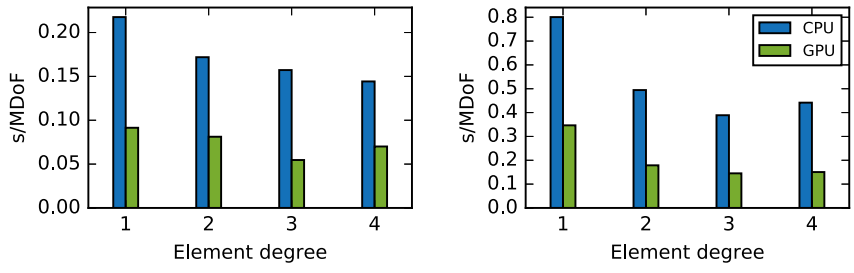


Figure 3.13. Time for a multigrid solve of the hyper shell problem when using a mixed precision approach or only double precision. The results are normalized to the problem size to allow for comparison of different setups, which are the same as in Figure 3.12.

We then continue with a comparison of the performance of our GPU implementation with a corresponding CPU implementation parallelized using MPI in combination with vector intrinsics. We again try to match the GPU and CPU hardware both in terms of power consumption and architectural generation. While the GPU version is executed on the Tesla P100 (300 W), the CPU version is run on a system featuring two 14-core Intel Broadwell Xeon E5-2690v4 processors (2×135 W). From Figure 3.14, we see that the GPU version is between two and three times faster than the CPU version for all element configurations.

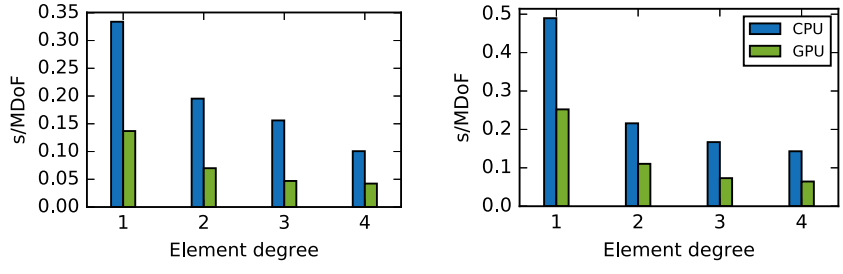
Next, the adaptively refined L-shaped domain discussed in Section 2.5.1 is considered. To include performance for adaptive meshes in 3D, we also include a three-dimensional generalization in the form a of a “hyper-L”; a cube where one octant is cut away. As discussed in Section 2.3.1, for Cartesian meshes, considerable bandwidth can be saved by only storing a single value per element for the inverse Jacobian. Since the L domain only gives rise to such elements, we employ this optimization in the implementation for the benchmarks. Looking at the results in Figure 3.15 we see that the GPU version



(a) 2D (b) 3D

Figure 3.14. Comparison of GPU and CPU performance for the Shell domain problem on meshes with about 10–24 million DoFs in 2D, and about 2–13 million DoFs in 3D.

is again two to three times faster than the CPU version, also when including hanging nodes in the mesh.



(a) 2D (b) 3D

Figure 3.15. Performance for the L-shaped domain on meshes with about 3 million DoFs for first-order elements, and between 20 and 30 million DoFs for elements of order two to four.

Finally, we evaluate the performance of the matrix-free multigrid solver applied to the nonlinear minimal surface equation (see Section 2.5.1). Table 3.6 shows the results for the CPU and GPU versions when fourth-order elements are used. Here, we note that the speedup (66%) is slightly smaller than for the linear example problem, which can be expected since the parallelism in the nonlinear line-search procedure is slightly lower than in linear solver.

Also, we see that while the number of nonlinear Newton iterations stays roughly constant with mesh refinement, the inner multigrid solver needs an increasing number of iterations. Just like for the shell example, this is due to the poor resolution of the coefficient on coarser grids, which is relatively steep close to the boundary for the boundary condition used in this set up.

It is perfectly possible to generalize the minimal surface equation to 3D, albeit with a somewhat unclear physical interpretation. In Table 3.7, we see the results when applying our solver to this problem. Here, the speedup reaches a factor of three, which is considerably larger than for the 2D case. We conclude

Table 3.6. *Performance results for the minimal surface equation using $\mathcal{Q}_4$ elements.*

			CPU		GPU	
Cells	DoFs	Nonlin. iter.	Lin. Iter.	Time solution	Lin. Iter.	Time solution
5.1k	82k	12	91	0.159	91	0.727
20k	0.33M	13	131	0.527	129	1.47
82k	1.3M	12	163	2.77	161	3.85
0.33M	5.2M	12	193	17.4	186	10.5

that the larger number of computations for the 3D elements is enough to make the system solution dominate the runtime, and hide the overhead from the nonlinear iteration.

Table 3.7. *Performance results for the 3D minimal surface equation using $\mathcal{Q}_4$ elements.*

			CPU		GPU	
Cells	DoFs	Nonlin. iter.	Lin. Iter.	Time solution	Lin. Iter.	Time solution
450	30k	8	23	0.0874	23	0.151
3.6k	0.23k	9	39	0.301	40	0.390
29k	1.8M	9	65	3.70	68	2.06
0.23M	15M	11	111	50.3	112	16.2

Conclusion

Through Papers II–IV, we have shown that the graphics processors can be used successfully for high-order finite-element computations, and that the matrix-free approach is essential for an efficient utilization of the processor. We have presented a GPU implementation with excellent performance, both compared to equivalent matrix-free implementations running on comparable CPUs, and compared to matrix-based implementations for the GPU. Furthermore, we have demonstrated the applicability of our method in a wide range of situations including curved geometry, adaptively refined mesh, and non-linear PDEs. Finally, using the matrix-free approach, no assembly is needed, and we are able to more efficiently use the limited memory of the GPU and solve problems about an order of magnitude larger.

4. Outlook

This thesis contributes a number techniques for efficient utilization of recent processors for finite element computations. We have shown that hardware transactional memory can perform competitively for concurrent updates of shared floating-point variables, while simplifying programming with the proper library support in place. Furthermore, we have shown that the matrix-free approach, which is crucial for leveraging higher-order elements, lends itself well for implementation on graphics processors.

The good performance of the matrix-free GPU implementation in various Poisson model problems demonstrates it has the potential of speeding up several important applications. These include simulation of turbulent fluid flows, where solution of Poisson's equation using high-order methods is central. Also, the application of a high-order discrete Laplace operator constitutes the main computational work in simulation of wave propagation.

As demonstrated in our benchmark experiments, the more efficient architecture of graphics processors can reduce the power consumption of a given simulation by a factor of two to three, which can have a big impact on large-scale simulations where the cost of electricity is substantial. More broadly, with the continuing growth in usage of throughput-oriented manycore processors such as graphics cards and the Intel Xeon Phi, we can expect bandwidth-efficient massively parallel algorithms such as our matrix-free finite element approach to be relevant for the hardware of the future.

5. Summary in Swedish

Datorbaserade simuleringar är idag en central del av samtliga fält inom naturvetenskap och teknik. När experiment är för dyra, farliga, eller på annat sätt omöjliga att genomföra i praktiken, kan matematisk modellering av det betraktade systemet vara ett alternativ. Ett exempel på detta från vår vardag är väderprognoser, där detaljrika atmosfärsfysikaliska modeller matade med stora mängder mätdata används för att förutsäga hur dagens väder ska utveckla sig framöver. De flesta dylika matematiska modeller är baserade på partiella differentialekvationer, för vilka lösningen beskriver det modellerade systemet. För många realistiska modeller är det dock ofta svårt eller till och med omöjligt att finna en exakt lösning till differentialekvationen. Istället används numeriska beräkningsmetoder för att ta fram en uppskattning av lösningen. I forskningsområdet beräkningsvetenskap studerar vi dels hur dessa beräkningsmetoder kan göras noggrannare så att en meningsfull lösning fås, och dels hur de kan göras effektivare så att beräkningarna kan slutföras inom rimlig tid. Om vi återvänder till exemplet väderprognoser handlar det om att se till att prognosen gör en korrekt förutsägelse, och att prognosen kan räknas ut i god tid innan vädret som förutsägs faktiskt inträffar. I den här avhandlingen är det främst effektivitetsaspekten som undersöks.

För många matematiska modeller resulterar de numeriska metoderna i en såpass stor mängd numeriska beräkningar att en lösning med papper och penna är fullständigt orealistisk. I dessa fall tar vi istället hjälp av datorer som programmeras att utföra beräkningen. På grund av den exponentiella utveckling av beräkningskraften hos datorprocessorer som är känd som Moores lag kan dessa idag utföra många miljarder additioner och multiplikationer per sekund, vilket tillåter oss att utföra simuleringsberäkningar med en detaljrikedom som var fullständigt otänkbar för bara några år sedan. Diverse trendbrott i hur processorer är designade skapar dock utmaningar när vi ska skriva våra beräkningsprogram så att den nya beräkningskraften tas tillvara fullt ut.

På grund av problem med bland annat värmeutvecklingen i processorn har vi de senaste åren sett en övergång från ökningar i processorns beräkningsfrekvens till ökningar i processorns parallellism, dvs en ökning i hur många beräkningar som kan utföras samtidigt. Moderna processorer har nu upp till tjugotalet parallella kärnor, vilka kan utföra en oberoende operation var per tidsenhet. Detta har gjort programmeringen mer komplex eftersom beräkningarna måste delas upp i oberoende delar som kan utföras parallellt. Vi har också sett att utvecklingen av beräkningskraft har dragit ifrån ökningen i minnesbandbredd. För många beräkningstillämpningar gör detta att beräkningskraften inte utnyttjas till fullo, då antalet beräkningsoperationer per minnesåtkomst

är för litet. Båda dessa trender har lett till att nya beräkningsmetoder har utvecklats som lättare går att parallellisera och som kräver färre minnesoperationer – ofta till ett pris av fler beräkningsoperationer.

En mycket viktig trend inom beräkningsvetenskap de senaste tio åren består i den ökande användningen av grafikkort för numeriska simuleringar. På grund av sin specialiserade design inriktad på 3D-grafik har grafikkort en fundamentalt mer resurseffektiv arkitektur jämfört med vanliga CPUer, både vad gäller strömförbrukning och inköpspris. Detta har lett till ett starkt intresse för att utforska möjligheterna att dra nytta av grafikkort för annat än grafik. Då många simuleringsberäkningar uppvisar en stor likhet med grafikberäkningar har det i flera fall varit mycket framgångsrikt och möjliggjort helt nya beräkningar. Grafikkorten har dock med sin grafikcentrerade design många begränsningar som utgör utmaningar för flera typer av beräkningar där strukturen inte är identisk den hos 3D-grafik.

I den här avhandlingen undersöks hur nya flerkärnesprocessorer och grafikkort kan utnyttjas på ett effektivt sätt för vetenskapliga beräkningar.

En av de viktigaste numeriska metoderna för lösning av differentialekvationer är finita elementmetoden som bygger på en uppdelning av beräkningsområdet i små diskreta delar, element, inom vilka lösningen antas bete sig som ett polynom. Vid det klassiska sättet att utföra finita elementberäkningar görs först en assemblering där man bestämmer ett linjärt ekvationssystem, som sedan löses i ett andra steg, vanligen med en iterativ lösningsmetod. Då lösningssteget vanligen utgör majoriteten av körtiden har mycket arbete lagts på effektivisering av det. Assembleringen är å andra sidan en mer komplex operation som involverar komplicerade databeroenden och parallella skrivningar till gemensamt minne.

I Manuskript I undersöks prestandan hos en parallell implementation av assembleringsfasen utförd på flerkärnesprocessorer. Vi betraktar transaktionsminne, som är en ny processorteknik för att hantera samtidiga skrivningar, och jämför den med konventionella tekniker såsom lås och atomiska operationer. Vi kör våra prestandatest på en experimentell processorprototyp som har hårdvarustöd för transaktionsminne, och visar att transaktionsminne är det snabbaste sättet att uppdatera delade flyttalsvariabler för beräkningsintensiva elementuppsättningar, samtidigt som det har potential att minska programmeringsmödan genom att förenkla hanteringen av delat minne.

I Manuskript II–IV betraktar vi en matrisfri variant av finita elementmetoden, och undersöker hur denna presterar vid exekvering på grafikkort. Ett kritiskt problem hos den konventionella tvåstegsvarianten av finita elementmetoden är att koefficientmatrisen för ekvationssystemet tar stor plats att lagra i minnet, vilket begränsar storleken på de modeller som kan simuleras, i synnerhet i tre dimensioner och för högre ordnings element. Vidare är matrisvektorprodukten, som står för majoriteten av arbetet i lösningsfasen, mycket dåligt anpassad för moderna processorer då den enbart utför omkring två beräkningsoperationer för varje minnesåtkomst, medan dagens processorer kräver upp

mot 80 operationer per minnesåtkomst för full utnyttjandegrad. Genom att slå ihop assembleringen med lösningsfasen, och därigenom göra sig av med matrisen, erhålls en matrisfri variant av matrisvektorprodukten som har väsentligt lägre utnyttjande av minnesbandbredden.

I Manuskript II presenterar vi en matrisfri implementation av matrisvektorprodukten för kartesiska beräkningsnät och visar att denna är överlägsen en matrisbaserad implementation för andra ordningens element eller högre, så länge som grafikkortets cacheminne räcker till för att lagra viktiga datastrukturer. I Manuskript III utvidgar vi implementationen till att även vara tillämpbar på fall med allmän geometri, adaptivt förfinat beräkningsnät, samt med modelldata som varierar över beräkningsområdet. Här visar våra prestandatest återigen att den matrisfria varianten är mycket snabbare än en matrisbaserad implementation samt kan angripa avsevärt större problem. Vidare är grafikkortsvarianten snabbare än en motsvarande matrisfri implementation för sedvanliga processorer, vilket visar på att grafikkort har en mer effektiv arkitektur för dylika beräkningar. I Manuskript IV sätter vi in den matrisfria matrisvektorprodukten i en fullständig lösare av partiella differentialekvationer och drar nytta av multigridmetoder för effektiv iterativ lösning av ekvationssystem. Genom tre olika exempelproblem visar vi att det matrisfria angreppssättet tillåter effektivt utnyttjande av grafikkort för fullständiga finita elementberäkningar involverande realistiska beräkningsnät. Här ser vi genomgående mellan två och tre gånger högre prestanda för grafikkort jämfört med en likvärdig uppsättning av flerkärnesprocessorer.

Våra prestandaresultat motiverar användning av den matrisfria metoden för realistiska simuleringar av ett flertal viktiga tillämpningar, såsom noggrann simulering av turbulenta vätskeflöden, och simulering av vågutbredning i elastiska medier. Vidare är energieffektiviteten lovande för framtidens datorer där vi förväntar oss att grafikkort och andra liknande massivt parallella processorer kommer spela en nyckelroll i att hålla ned strömförbrukningen i storskaliga beräkningar.

Acknowledgments

First of all, I would like to thank my main advisor Jarmo Rantakokko for your help and encouragement, and my co-advisors Gunilla Kreiss and Sverker Holmgren for valuable experience, contacts, and research context. Secondly, I want to thank my main collaborator, co-author, and general mentor in research, Martin Kronbichler. I hope we get the chance to more inspiring collaborations in the future. I also thank my other co-authors and collaborators, Bruno Turcksin, Martin Tillenius, David Black-Schaffer, Martin Karlsson, and Elisabeth Larsson. Moreover, I thank Wolfgang Bangerth for accepting when I invited myself to visit Texas A & M University during spring 2016, and Rene Gassmöller and Juliane Dannberg for taking care of me during my stay.

Furthermore, I am grateful for my colleagues and friends at TDB for providing a friendly atmosphere and making me always enjoy going to work. You are too many to be named, but should all take this personally: *Thank you!* A special thanks goes to Fredrik Hellman for proofreading this comprehensive summary, and for being a great friend. I also want to express my gratitude for the excellent leadership and management skills of Lina von Sydow and Michael Thuné that make it a pleasure to work at TDB and the IT department.

On a personal note, I want to thank my parents for always believing in me and for collaboration in making me who I am. I also thank Malungs folkhögskola, V-dala spelmanslag, and Klättercentret for seeing to my well-being in my life outside of academia during these years.

The work that resulted in Paper I was carried out within the Linnaeus center of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center, supported by the Swedish Research Council, which also provided computer systems for experiments in Paper II–IV. Funding for travel has been gratefully received from Ångpanneföreningens Forskningsstiftelse, Anna-Maria Lundins stipendiefond, Stiftelsen Lars Hiertas Minne, H F Sederholms stipendiestiftelse, and Knut och Alice Wallenbergs stiftelse. I finally acknowledge the support from the Advanced Parallelization and Algorithms Performance Research Unit of Trinidad and Tobago in classified activities related to the thesis.

References

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

[2] M. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss–Seidel. *Journal of Computational Physics*, 188:593–610, 2003.

[3] 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*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[5] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 8.5. *submitted*, 2017.

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

[7] W. Bangerth, R. Hartmann, and G. Kanschat. deal. II – A General Purpose Object-Oriented Finite Element Library. *ACM Transactions on Mathematical Software*, 33(4):24/1–24/27, 2007.

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

[9] 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, pages 18–1, New York, NY, USA, 2009. ACM.

[10] M. Berger, M. Aftosmis, and G. Adomavicius. Parallel multigrid on Cartesian meshes with complex geometry. In *Parallel Computational Fluid Dynamics*, pages 283–290, Amsterdam, 2001. North Holland.

[11] 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*, pages 41–46, 1982.

[12] 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*, pages 81–91, 2007.

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

[14] A. Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of Computation*, 31:333–390, 1977.

[15] 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*, 43(1, SI):23–28, 2011.

[16] G. F. Carey and B. Jiang. Element-by-element linear and nonlinear solution schemes. *Communications in Applied Numerical Methods*, 2(2):145–153, 1986.

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

[18] 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*, 29(2):6–16, MAR-APR 2009.

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

[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*, 9(5):256–268, Oct 1974.

[21] 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*, pages 157–168, New York, NY, USA, 2009. ACM.

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

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

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

[25] E. Elsen, P. LeGresley, and E. Darve. Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics*, 227(24):10148–10161, 2008.

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

[27] C. Farhat and L. Crivelli. A General-Approach to Nonlinear Fe Computations on Shared-Memory Multiprocessors. *Computer Methods in Applied Mechanics and Engineering*, 72(2):153–171, 1989.

[28] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations. *Int. J. Parallel Emerg. Distrib. Syst.*, 22(4):221–256, January 2007.

[29] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Towards realistic performance bounds for implicit CFD codes. In *Proceedings of Parallel CFD'99*. Elsevier, 1999.

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

[31] 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*, 32(2):48–60, March 2012.

[32] M. Harris. Optimizing cuda. *SC07: High Performance Computing With CUDA*, 2007.

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

[34] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[35] T. Y. Hou and X. Wu. A Multiscale Finite Element Method for Elliptic Problems in Composite Materials and Porous Media . *Journal of Computational Physics* , 134(1):169–189, 1997.

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

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

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

[39] M. G. Knepley and A. R. Terrel. Finite element integration on gpus. *ACM Trans. Math. Softw.*, 39(2):10:1–10:13, February 2013.

[40] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357–397, JAN 20 2004.

[41] 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*, 229(20):7692–7714, 2010.

[42] 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*, 69(5):451–460, 2009.

[43] D. Komatitsch and J. Tromp. Introduction to the spectral element method for three-dimensional seismic wave propagation. *Geophysical Journal International*, 139(3):806–822, DEC 1999.

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

[45] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, JUL 2003.

[46] 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)*, pages 1326–35, 2013.

[47] E. S. Larsen and D. McAllister. Fast Matrix Multiplies Using Graphics Hardware. In *Supercomputing, ACM/IEEE 2001 Conference*, pages 43–43, 2001.

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

[49] 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.

[50] S. A. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(2), 2008. Annual Meeting of the Italian-Society-of-Bioinformatics, Naples, ITALY, APR 26-28, 2007.

[51] 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*, 71(1):80–97, JAN 10 2013.

[52] J. M. Melenk, K. Gerdes, and C. Schwab. Fully discrete hp-finite elements: fast quadrature . *Computer Methods in Applied Mechanics and Engineering* , 190(32–33):4339–4364, 2001.

[53] D. Michéa and D. Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[54] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

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

[56] NVIDIA Corporation. GP100 Pascal Whitepaper. Technical report, 2016. Version 1.1.

[57] NVIDIA Corporation. *Kepler Tuning Guide*, 2016.

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

[59] S. A. Orszag. Spectral methods for problems in complex geometries . *Journal of Computational Physics* , 37(1):70–92, 1980.

[60] 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*, pages 21–51, 2005.

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

[62] Y. Pinchover and J. Rubinstein. *An Introduction to Partial Differential Equations*. Cambridge University Press, 1 edition, 2005.

[63] 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*, 228(12):4468–4477, JUL 1 2009.

[64] 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.

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

[66] B. Saha, A. 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*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[67] Y. Shapira. *Matrix-Based Multigrid: Theory and Applications*. Springer Publishing Company, Incorporated, 2nd edition, 2010.

[68] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering*, 12(3):66–72, MAY-JUN 2010.

[69] M. Tillenius. SuperGlue: A shared memory framework using data-versioning for dependency-aware task-based parallelization. Technical Report 2014-010, Department of Information Technology, Uppsala University, April 2014.

[70] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Elsevier Academic Press, London, 2001.

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

[72] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC*, volume 10, page 16. San Jose, CA, 2010.

[73] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[74] 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*, page 11, 2013.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1512

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-320147

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2017