This is a preprint of an article accepted for publication in the Int. J. Computational Science and Engineering

Using GPUs to Improve Multigrid Solver Performance on a Cluster

Dominik Göddeke*
Institute of Applied Mathematics, University of Dortmund, Germany
E-mail: dominik.goeddeke@math.uni-dortmund.de
*Corresponding author

Robert Strzodka
Max Planck Center, Computer Science
Stanford University, USA

Jamaludin Mohd-Yusof, Patrick McCormick
Computer, Computational and Statistical Sciences Division
Los Alamos National Laboratory, USA

Hilmar Wobker, Christian Becker, Stefan Turek
Institute of Applied Mathematics, University of Dortmund, Germany

Abstract: This article explores the coupling of coarse and fine-grained parallelism for Finite Element simulations based on efficient parallel multigrid solvers. The focus lies on both system performance and a minimally invasive integration of hardware acceleration into an existing software package, requiring no changes to application code. Because of their excellent price performance ratio, we demonstrate the viability of our approach by using commodity graphics processors (GPUs) as efficient multigrid preconditioners. We address the issue of limited precision on GPUs by applying a mixed precision, iterative refinement technique. Other restrictions are also handled by a close interplay between the GPU and CPU. From a software perspective, we integrate the GPU solvers into the existing MPI-based Finite Element package by implementing the same interfaces as the CPU solvers, so that for the application programmer they are easily interchangeable. Our results show that we do not compromise any software functionality and gain speedups of two and more for large problems. Equipped with this additional option of hardware acceleration we compare different choices in increasing the performance of a conventional, commodity based cluster by increasing the number of nodes, replacement of nodes by a newer technology generation, and adding powerful graphics cards to the existing nodes.

Keywords: parallel scientific computing; Finite Element calculations; GPUs; floating-point co-processors; mixed precision; multigrid solvers; domain decomposition

Reference to this paper should be made as follows: Göddeke et al. (2007) 'Using GPUs to Improve Multigrid Solver Performance on a Cluster', Int. J. Computational Science and Engineering, Vol. x, Nos. a/b/c, pp.1–20.

Biographical notes: Dominik Göddeke and Hilmar Wobker are PhD students, working on advanced computer architectures, computational structural mechanics and HPC for FEM. Robert Strzodka received his PhD from the University of Duisburg-Essen in 2004 and is currently a visiting assistant professor, researching parallel scientific computing and real time imaging. Jamaludin Mohd-Yusof received his PhD from Cornell University (Aerospace Engineering) in 1996. His research at LANL includes fluid dynamics applications and advanced computer architectures. Patrick McCormick is a Project Leader at LANL, focusing on advanced computer architectures and scientific visualization. Christian Becker received his PhD from Dortmund University in 2007 with a thesis on the design and implementation of FEAST. Stefan Turek holds a PhD (1991) and a Habilitation (1998) in Numerical Mathematics from the University of Heidelberg and leads the Institute of Applied Mathematics in Dortmund.

1 INTRODUCTION

The last several years have seen a resurgence of interest in the use of co-processors for scientific computing. Several vendors now offer specialized accelerators that provide high-performance for specific problem domains. In the case of a diverse High Performance Computing (HPC) environment, we would like to select one or more of these architectures, and then schedule and execute portions of our applications on the most suitable device. In practice this is extremely difficult because these architectures often have different restrictions, the time required to move data between the main processor and the co-processor can dominate the overall computation time, and programming in a heterogeneous environment is very demanding. In this paper we explore this task by modifying an existing parallel Finite Element package, with more than one hundred thousand lines of Fortran code, to leverage the power of specialized co-processors. Instead of exclusively focusing on performance gains, we also concentrate on a minimally invasive integration into the original source code. Although the coupling of coarse and fine-grained parallelism is still demanding, we can

- restrict the code changes in the original package to approximately 1000 lines (Section 4.4),
- retain all previous functionality of the package (Section 5.3),
- give the benefit of hardware acceleration to unchanged application code based on the package (Section 4), and
- multiply the performance of the multigrid solvers by a factor of two to three (Section 5).

In this section we discuss the interaction of the hardware, software and economic considerations, and provide the necessary background. Section 2 continues with the description of the test scenario and the involved system components. The specific details of the underlying Finite Element package are outlined in Section 3. Section 4 considers the algorithmic approach and implementation details. Results are discussed in Section 5. Finally, we present our conclusions and plans for future work in Section 6.

1.1 Hardware for HPC

The vast majority of processor architectures are both fast and energy efficient when the required data is located within the processor's local memory. In contrast, the transport of data between system and processor memory is expensive both in terms of time and power consumption. This applies to both the small scales in a processor, as well as at the larger system level.

Even though this memory wall problem [67] has been known for many years, the common choice of processors and systems in HPC often seems to ignore this fact. The problem lies in the performance/cost ratio. While there exist computing devices better suited to address the memory wall problem, they are typically more expensive to purchase and operate. This is due to small production lots, immature or complex software tools, and few experienced programmers. In addition there is always the question of compatibility and the risk of having invested into a discontinued technology, both of which would lead to higher overall costs. On the other hand, standard components are cheaper, readily available, well tested, modular, easier to operate, largely compatible, and they commonly support full development tool chains. In the end, these characteristics tend to favor economic considerations over memory and computational efficiency.

However, if one standard component is easy to operate and maintain, this does not imply that a large number of them is just as easy to handle. Nevertheless, an analysis of the recent TOP500 lists [46] reveals a quickly growing number of clusters assembled from commodity hardware components listed among more traditional HPC systems. Commodity Central Processing Units (CPUs) have made the transition from the field of latency dominated office applications, to the realm of throughput dominated HPC programs. This trend has been partially driven by the performance demands of consumer level multimedia applications. Even with these advancements, commodity processors still suffer from the memory wall problem.

The industry is currently experiencing the beginning of a similar transition of other hardware to the field of HPC, but this time sustained by the mass market of computer gaming. Graphics Processing Units (GPUs) and more recently the Cell [50, 68] and the PhysX [1] processors target this performance hungry entertainment community. In this case, we once again have a situation in which hardware has been developed for an application domain that has very limited relevance to HPC. But similar to today's commodity CPUs, these devices have advanced to the point that their performance/cost ratio makes them of interest to the high-performance community.

A similar development occurs in the market of Field Programmable Gate Arrays (FPGAs). Initially designed as general integrated circuit simulators, current devices have diversified into different types optimized for specific applications areas. Again HPC is not their main market, but some of the specializations target similar processing requirements. Cray's XD1 system is an example of a supercomputer designed to leverage the power of FPGAs [11].

Repeatedly adapting different architectures, especially those designed for other application domains, to the needs of HPC is clearly not an ideal situation. However, the current high-performance market is not large enough to make the enormous costs associated with new hardware design *and* the supporting development environment profitable. While some companies offer accelerator boards for HPC, for example Clearspeed [8], they are typically also suitable for the needs of other application domains. This leads us in an interesting direction, where we no longer try to run everything on the same kind of processor, but rather pick

from among numerous devices those that are best suited for the different parts of our application. This introduces a new imbalance between the theoretical efficiency of the given hardware-software combination and the achievable efficiency, which is likely to degrade with higher numbers of heterogeneous components.

1.2 Software for HPC

In general, the reluctance to complicate application software has been so high that promises of using heterogeneous architectures for significant speedups have met limited success. However, the move towards parallel computing has also reached the CPU arena, while the performance gap of the CPU to parallel co-processors has further widened. Moreover, computing with dedicated co-processors not only increases performance but also addresses the memory wall, the power problem, and can also reduce other indirect costs associated with the size of clusters because fewer nodes would be required. The success of a particular co-processor depends heavily on the software complexity associated with the resulting heterogeneous system.

Therefore, our main focus is the *minimally invasive* integration of the fine-grained parallelism of the co-processor within the coarse-grained parallelism of a Finite Element (FE) package executing on clusters. Our goal is to integrate the hardware alternatives at the cluster node level. The global domain decomposition method does not distinguish between different architectures but rather between different solvers for the sub-domains. The CPU is the most general architecture and supports all types of solvers and grids. The dedicated co-processors offer accelerated backends for certain problem structures, that can be chosen automatically if applicable. The application never has to deal with the different computing and programming paradigms. Most importantly, the interface to the FE package remains the same as in the unaccelerated version.

For this abstraction to work, the initial FE package must fulfill certain criteria. Grid generation for complex simulation domains is a demanding task. While some areas can be easily covered with a regular grid, others may require an unstructured or even adaptive discretization to accurately capture the shape without wasting too many elements. It is important that the FE package maintains the distinction between structured and unstructured parts of the discretization, rather than trying to put everything into a homogeneous data structure, because the parallel co-processors can utilize much more efficient solvers when provided with the additional information about the grid structure. In fact, not only the co-processors benefit from this additional information, the sub-problems with a regular structure also execute much faster on the CPU because of coherent memory access patterns. However, for the CPU this is merely an additional advantage in processing of the data structures, whereas the parallel co-processors depend heavily on certain data movement patterns and we lose several factors in speedup if we ignore them.

Because the co-processors are optimized for different application areas than HPC, we must respect certain restrictions and requirements in the data processing to gain performance. But this does not mean that we are willing to sacrifice *any* software functionality. In particular, we make *absolutely no* compromises in the final accuracy of the result. The hardware restrictions are dealt with by the software which drives the co-processor and where the co-processor is incapable or unsuitable for executing some part of the solver, it falls back to the existing CPU implementation. It is important to note that we are not trying to make everything run faster on the co-processor. Instead each architecture is assigned those tasks that it executes best. The application never has to deal with these decisions, as it sees an accelerated solver with exactly the same interface as the pure CPU version. One assumption we make here is that the tasks to be performed are large enough, so that the costs of configuring the co-processor, and moving data back and forth between CPU and co-processor, can be amortized over the reduced runtime.

In view of the memory wall problem we are convinced that despite the different structures of the co-processors, the computation can be arranged efficiently as long as data transport between host and co-processor is minimized and performed in coherent block transfers that are ideally interleaved with the computation. Therefore, the structure preserving data storage and handling is the main assumption we make about the FE package to enable the coupling of the coarse and fine grained parallelism. All other conditions of the existing software packages are kept to a minimum. By taking this approach, the data structure based coupling can be applied to many parallel applications and executed on different parallel co-processors.

Despite the different internal structure of the parallel co-processors, they share enough similarities on the data flow level to allow for a similar interface to the CPU. Clearly, the device specific solver must be reimplemented and tuned for each architecture, but this requires only the code for a local one node solution, as the coarse grained parallelism is taken care of by the underlying FE package on the CPUs. Due to economic considerations, a CPU-GPU coupling in each node of a cluster is our first choice for a heterogeneous computing platform. We focus on this hardware combination throughout the remainder of the paper.

1.3 GPU background

We aim at efficient interoperability of the CPU and GPU for HPC in terms of both computational and user efficiency, i.e. the user should be able to use the hardware accelerated solvers with exactly the same ease as their software equivalents. In our system, this is achieved by a simple change in parameter files. We assume that both an MPI-based FE package and a (serial) GPU-based multigrid solver are given. While most readers will be familiar with the ideas and concepts of the former, the same is probably less true for the latter. Since we believe the same concept can be applied to other co-processors, we

do not use any GPU specific terminology to explain the interfaces. However, the understanding of the data transport and computation on the GPU in balance with the CPU (discussed in Section 4) requires some insight into the computing paradigm on the GPU.

For an algorithmic CPU-GPU comparison without any graphics terminology we refer to Strzodka et al. [59]. Detailed introductions on the use of GPUs for scientific computing with OpenGL and high level graphics languages *(GPGPU – general purpose computation on GPUs)* can be found in several book chapters [30, 52]. For tutorial code see Göddeke [23]. The survey article by Owens et al. [48] offers a wider view on different general purpose applications on the GPU; and the community web site has a large collection of papers, conference courses and further resources [27].

Our implementation of linear algebra operators and in particular multigrid solvers on GPUs builds upon previous experience in the GPGPU community. Most relevant to our work are implementations of multigrid solvers on GPUs studied by Bolz et al., Krüger and Westermann, Goodnight et al., Strzodka et al. [5, 25, 42, 60] and the use of multiple GPUs for discrete computations by Fan et al., Fung and Mann, and Govindaraju et al. [19, 20, 26]. However, these publications do not focus on the achievable accuracy.

We discuss implementational aspects in Section 4.4.

1.4 Low precision

An important drawback shared by several of the coprocessors is the restriction to single floating-point representation. For instance, GPUs implement only quasi IEEE 754 conformal 32-bit floating point operations in hardware, without denormalization and only round-to-zero. For many scientific computations this is actually an efficiency *advantage* as the area required for a multiplier grows quadratically with the operand size. Thus if the hardware spends the area on single precision multipliers, it offers four times as many of them as double multipliers. For floating-point dominated designs this has a huge impact on the overall area, for cache and logic dominated designs the effects are much smaller, but we ideally want many parallel floating-point units (FPUs). In FPGAs, this benefit is truly quadratic, whereas in the SIMD units of CPUs, the savings are usually reduced to linear because of the use of dual-mode FPUs that can compute in single and double precision. In addition to more computational resources, the use of single precision also alleviates the memory wall problem. For a more thorough discussion of the hardware efficiency of low precision components see Göddeke et al. [24].

Clearly, most numerical applications require high precision to deliver highly accurate (or even correct) results. The key observation is that high precision is only necessary in a few, crucial stages of the solution procedure to achieve the same result accuracy. The resulting technique of mixed precision iterative refinement has already been introduced in the 1960s [45]. The basic idea is to repeatedly gain a little bit of relative accuracy with a low precision solver and accumulate these gains in high precision. While originally this technique had been used to increase the accuracy of a computed solution, it has recently regained interest with respect to the potential performance benefits. Langou et al. evaluate mixed precision schemes for dense matrices in the LAPACK context on a wide range of modern CPUs and the Cell processor [43]. The viability of mixed precision techniques on GPUs for (iterative) multigrid solvers on strongly anisotropic grids and thus matrices with high condition numbers is demonstrated by Göddeke et al. [24]. Both publications emphasize that the achievable accuracy in the results remains identical to computation in high precision alone. Without the mixed precision approach we would need to emulate double precision operations on the parallel devices, thus doubling the required bandwidth and increase the operation count by at least a factor of ten.

The GPU solvers only contribute a small step forward towards the overall solution, and our hope is that the approach is reasonably stable in view of possible false reads or writes into memory. Graphics cards do not utilize Error Correcting Code (ECC) memory, which is one of the remaining issues for their adoption in HPC applications (the other one being the lack of double precision storage and arithmetic, see previous paragraph). By avoiding lengthy computations on the processor our hope is to reduce the probability of memory errors and transient faults being introduced into the calculations. Additionally, our proposed hierarchical solver (see Section 3.2) corrects single-event errors in the next iteration. To date we have not seen evidence of such memory errors affecting our results. For a detailed discussion and analysis of architectural vulnerability for GPUs in scientific computing, as well as some proposals for extending graphics hardware to better support reliable computation, see Sheaffer et al. [55].

1.5 CPU-GPU coupling

If the Finite Element package manages a combination of structured and unstructured sub-problems as explained in Section 1.2, then we want to execute GPU-accelerated solvers for the structured sub-problems. In the context of a (parallel) multigrid solver, this assigns the GPU the task of a local smoother.

For an efficient coupling between the FE package and the GPU solvers, we need to integrate decisions about how to distribute the sub-problems onto the different computational resources. The first choice is simply based on the availability of solvers. The GPU backend (currently) supports only a small subset of the CPU solvers, in particular some sub-problems converge only with an advanced solver which is available only on the CPU. The second choice is more complex and involves the assignment to an architecture based on the type and size of the sub-problem. Two additional factors further complicate the decision process. First, the GPU solver is in fact always a coupled GPU-CPU solver as it requires some CPU support to orchestrate the computation on the GPU, to obtain the double precision

accuracy with iterative refinement, to calculate local contributions to global vectors and to perform the MPI communication with the other processes (we discuss this in more detail in Section 4). Second, due to the memory wall problem, the computation on a principally slower architecture might be faster if less data has to be moved. This means that the ratios between the computational power and memory bandwidth, techniques to bypass the CPU in memory transfers to the GPU and to overlap computation with data transport are crucial.

Overall we have a complicated dynamic scheduling problem which requires a thorough examination, and we do not address it in this paper. For the presented examples we use a simple static or semi-automatic heuristic scheduling based on experience and serial calibration runs.

1.6 Related work

More elaborate discussions on hardware and software trends in HPC for PDEs are presented by Rüde, Keyes, Hoßfeld, Gropp et al. and Colella et al. [9, 28, 34, 37, 51]. Hardware considerations for large-scale computing are elaborated upon by DeHon, Khailany et al., and Dally et al. [12, 14, 38]. General trends and challenges to further uphold Moore's Law are discussed in detail in the annual SEMATECH report [54].

Data locality techniques, especially for multigrid methods, have been extensively studied by Douglas, Rüde et al. and Turek et al. [16, 17, 41, 65]. Optimization techniques for HPC are presented by Garg and Sharapov, and Whaley et al. [21, 66].

Surveys of different parallel computing architectures, especially reconfigurable systems, can be found in Hartenstein, Bondalapati and Prasanna, and Compton and Hauck [6, 10, 32, 33]. Exploitation of different types of parallelism and their relevance are performed by Sankaralingam et al., Guo et al., Taylor et al. [29, 53, 62]. Comparisons of multiple parallel architectures on typical stream kernels and PDE solvers are studied by Suh et al. and Strzodka [58, 61].

Iterative refinement methods are discussed in length by Demmel et al., and Zielke and Drygalla [15, 70]. Applications on the CPU usually focus on efficient extensions of the precision (in intermediate computations) beyond the double precision format as demonstrated by Li et al., and Geddes and Zheng [22, 44]. GPUs and FPGAs and more literature in the context of emulated- and mixed-precision computations are discussed by Göddeke et al. [24].

Related work on the use of GPUs as co-processors is presented in Section 1.3.

2 SYSTEM COMPONENTS

2.1 Test scenario

To evaluate the GPU accelerated solver we focus on the Poisson problem $-\Delta \mathbf{u} = \mathbf{f}$ on some domain $\Omega \subseteq \mathbb{R}^2$, which is justified by the observation that in real-world simulations Poisson problems are often the most time-consuming subtasks. For instance, the solution of the Navier-Stokes equations in Computational Fluid Dynamics (CFD) using projection schemes requires the accurate solution of a Pressure-Poisson problem in every time-step [64].

In our tests, we discretize several two-dimensional domains using conforming bilinear Finite Elements of the Q_1 FEM space. We choose analytic test functions $\mathbf{u}_0$ and define the right hand side as the analytical Laplacian of these functions: $\mathbf{f} = -\Delta \mathbf{u}_0$. Thus we know that $\mathbf{u}_0$ is the exact analytical solution to the continuous PDE, and we can evaluate the integral L_2 error of the discretely computed results against the analytical reference solution $\mathbf{u}_0$ to measure the accuracy.

In the evaluation of this test scenario we are mainly interested in accuracy, absolute performance and weak scalability. See Section 5 for results.

2.2 Hardware

For the solution of the Poisson problem we use up to 17 nodes of two different clusters DQ and LiDO using various configurations. The DQ cluster nodes contain dual EM64T CPUs, a single PCI Express (PCIe) connected graphics card, and an InfiniBand interface that is connected to a full bandwidth switch. The detailed configuration for each node is:

CPU: Dual Intel EM64T, 3.4 GHz, 1 MiB L2 cache, 800 MHz FSB, 600 W power supply.

RAM: 8 GiB[1] (5.5 GiB available), 6.4 GB/s shared bandwidth (theoretical).

POWER: ∼315 W average power consumption under full load without the GPU.

GPU: NVIDIA Quadro FX4500 PCIe, 430 MHz, 114 W max power.

RAM: 512 MiB, 33.6 GB/s bandwidth.

POWER: ∼95 W average power consumption under full load of the GPU alone.

It costs approximately $\$1,400$ to add the FX4500 graphics card to the system. In comparison, a new cluster node with an identical configuration, but without a GPU, costs approximately $\$4,000$, not counting infrastructure such as rack space and free ports of the switches. No additional power supply or cooling unit is required to support the graphics cards in the cluster. Moreover, the cluster has been reliable with the addition of graphics cards that have required only a small amount of additional administration and maintenance tasks. Equivalent consumer-level graphics cards cost approximately \$500.

The chipset used in DQ's EM64T architecture presents a significant performance bottleneck related to a shared memory bus between the two processors. The LiDO cluster

[1] International standard [35]: G= 10^9, Gi= 2^{30}, similarly Mi, Ki.

does not have this limitation and allows us to quantify the resulting benefits of the higher bandwidth to main memory. Each node of LiDO is connected to a full bandwidth InfiniBand switch and is configured as follows:

CPU: Dual AMD Opteron DP 250, 2.4 GHz, 1 MiB L2 cache, 420 W power supply.

RAM: 8 GiB (7.5 GiB available), 5.96 GB/s peak bandwidth per processor.

POWER: ∼350 W average power consumption under full load.

A new cluster node with an identical configuration costs approximately $\$3,800$. Unfortunately, this cluster does not contain graphics hardware, therefore, we cannot test all possible hardware configurations. But the importance of the overall system bandwidth becomes clear from the results presented in Section 5.

2.3 Software

We use the Finite Element package FEAST [3, 4, 65] as a starting point for our implementation. FEAST discretizes the computational domain into a collection of *macros*. This decomposition forms an unstructured coarse grid of quadrilaterals, and each macro is then refined independently into *generalized tensor-product* grids, see Figure 1. On the resulting computational grid, a multilevel domain decomposition method is employed with compute intensive local preconditioners. This is highly suitable for the parallel co-processors, as there is a substantial amount of local work on each node to be done, without any communications interrupting the inner solver. In addition, new local smoothers can be developed and tested faster on a single machine because they are never involved in any MPI calls.

FEAST provides a wide selection of smoothers for the parallel multigrid solver, e.g. simple Jacobi iteration for almost isotropic sub-grids or operators, and ILU or alternating-directions tri-diagonal Gauss-Seidel (ADI-TRIGS) for more anisotropic cases. Thus, the solver for each macro can be chosen to optimize the time to convergence. However, this matching of solvers and macros is a fairly recent feature in FEAST. Currently, two different local solvers can be used concurrently: CPU-based components take advantage of the collection of solvers readily available in FEAST and are applied to local sub-problems that require strong smoothing due to high degrees of anisotropy in the discretization. Our GPU-based multigrid solver currently only offers Jacobi smoothing and is applied to mostly isotropic sub-problems. Implementing stronger smoothers on the GPU in the future will allow us to tackle harder problems with fine grained parallelism. The next section motivates and explains in more detail FEAST's approach to domain discretization and the hierarchical solution.

We use the GotoBLAS and UMFPACK [13] libraries in parts of the CPU implementation, and NVIDIA's Cg language with OpenGL for the GPU code.

3 DATA STRUCTURE AND TRAVERSAL

3.1 Structured and unstructured grids

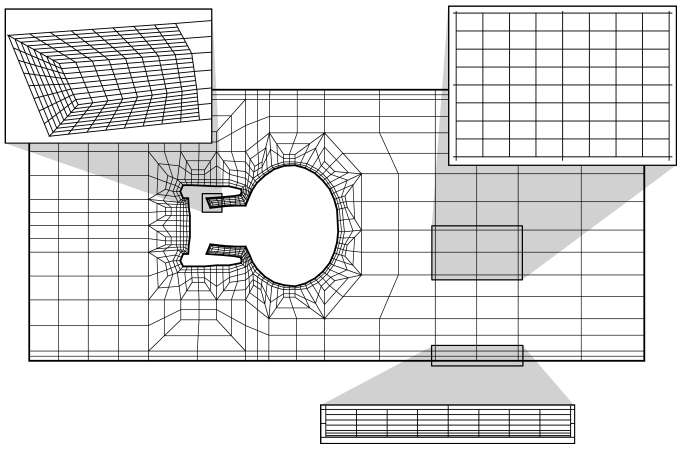

Figure 1: An unstructured coarse grid composed generalized tensor-product macros with isotropic (regular) and anisotropic refinement.

Unstructured grids give unlimited freedom to place grid nodes and connect them to elements, but this comes at a high cost. Instead of conveying the data structure to the hardware, we let the processor speculate about the arrangement by prefetching data that might be needed in the future. Obviously, execution can proceed much faster if the processor concentrates on the computation itself and knows ahead of time which data needs to be processed. By avoiding memory indirections the bandwidth is utilized optimally, prefetching only the required data and maximizing the reuse of data in smaller, higher level caches.

From an accuracy point of view, the absolute freedom of unstructured grids is also not needed or can be achieved with a mild increase in the number of elements. To capture the large-scale form of the computational domain, FEAST covers it with an unstructured coarse grid of macros, each of which is refined into a generalized tensor-product grid (cf. Figure 1 and Section 2.3). There is still a lot of freedom in placing the grid nodes on the finest level, but the regular data structure of the local grid is preserved. To obtain higher resolution in certain areas of the computational domain, the macros can be adaptively refined further, i.e. each macro can have a different number of grid nodes, introducing hanging nodes on the inner boundaries between neighboring macros. Finally, FEAST can also handle r-adaptivity on the macro level by moving existing grid nodes based on an error estimation of intermediate solutions. The global structure of local generalized tensor-product grids is always preserved; only the local problem size and condition number may change.

Discretized PDEs on unstructured grids generally lead to data representations with a sparse matrix structure such as the compact row storage format. Such storage formats generate coherency in the matrix rows but imply an incoherent, indirect memory access pattern for the vector

components. This is one of the main reasons why they perform weakly with respect to the peak efficiency of modern hardware [3, 65]. Generalized tensor-product grids on the other hand lead to banded matrices after PDE discretization, and matrix-vector multiplication can in turn be implemented in a memory coherent way by using blocked `daxpy`-like operations for each band. Although the entire *coarse* grid may be quite complex, for each macro the required data flow in a matrix vector product is known in advance. This allows the optimization of the memory access patterns depending on the target architecture.

Memory coherency is now such an important factor that it favors basically all computationally oriented architectures despite their fundamental differences. The impact is especially high for parallel co-processors which devote a higher percentage of transistors to FPUs rather than logic or cache hierarchies. The parallelism supplied by many FPUs enforces an explicit block-based data transfer model for most of the parallel devices. For example, almost all data transport to the graphics card happens in 1D, 3D or most commonly 2D arrays and the driver rearranges the data to benefit local neighborhood access patterns. The Cell processor requires even a two-level explicit block-based data movement model, and in FPGAs obviously all levels of data storage and transport are handled explicitly, most often in blocks or streams.

In principle, this explicit block-based data communication does not make handling of unstructured data impossible, but it is more complex and loses a significant amount of performance, mainly due to the implied memory indirections, in comparison to the structured case. So instead of trying to implement these irregular structures, much more performance can be gained by accelerating more of the common cases on the co-processors.

3.2 Coupling and decoupling

For large-scale linear equation systems arising from a discretization of PDEs, two general goals typically conflict: For efficient numerical convergence it is advantageous to incorporate a global coupling in the solution process; for efficient parallelization, on the other hand, locality greatly reduces the communication overhead. Multigrid methods that incorporate powerful smoothers such as SOR, ILU or ADI-TRIGS generally deliver very good serial performance, numerical robustness and stability. However, they imply a strong global coupling and therefore global communication due to their recursive character. Non-recursive smoothing operators like Jacobi act more locally, but can lead to poorly converging or even diverging multigrid iterations in the presence of anisotropies. Additionally, the ratio between communication and computation deteriorates on coarser multigrid levels. At the bottom of the hierarchy, the coarse grid solver always implies global communication and can similarly become the bottleneck.

In contrast, *domain decomposition* methods partition the computational domain into several sub-domains and discretize the PDE locally. This introduces artificial inner boundaries for the coupling between the local problems, but retains good parallel scalability since much more computational work is performed on the fine grids. For a detailed introduction on Schur-complement- and Schwarz-methods we refer the reader to the books by Smith et al., Toselli and Widlund, and Kornhuber et al. [40, 56, 63].

The ScaRC (*Scalable Recursive Clustering*, [3, 39]) scheme combines the advantages of these two conflicting approaches in a multilevel domain decomposition technique. On the global level, a domain decomposition technique on the (unstructured) coarse grid is applied, and several macros are grouped in sub-domains (MPI processes). Each macro is then refined into a generalized tensor-product grid. The degree and method of refinement can be different for each macro, some are refined regularly while others are refined anisotropically (see Figure 1). The idea behind this technique is to exploit structured parts of the domain while hiding anisotropies locally, to maximize the robustness and (numerical and computational) efficiency of the overall solution process. On each refined macro, the ScaRC scheme employs a local multigrid solver. This local solver requires only local communication, namely after each step of the outer solver, data is exchanged over the inner boundaries introduced by the decomposition on the macro level. In the easiest case, the global solver is just a straight-forward Richardson defect correction loop. On the next level of complexity, Krylov subspace methods can be applied, for instance the conjugate gradient iterative scheme. Ultimately, smoothing an outer multigrid scheme with local multigrid solvers on each macro yields the best convergence rates and fastest time to solution on large or anisotropic domains. In summary, this approach avoids deterioration of the convergence of the global solver by hiding (encapsulating) anisotropies locally, while exploiting regular structures to leverage high computational efficiency. For a detailed analysis, we refer to Kilian and Becker [3, 39].

Feast is built around the ScaRC approach and uses MPI for inter-node communication. The artificial inner boundary conditions implied by the decomposition are treated automatically. This decouples the local solver from any global communication and global data structures. The local solver only needs to request data for a local sub-problem that comprises several macros, which always have the structure of a generalized tensor-product mesh. The only necessary communication calls are exchange-and-average operations once the local solvers are finished. Thus, new smoothers such as our GPU-based multigrid implementation can be added with a comparatively small amount of wrapper code.

4 IMPLEMENTATION

4.1 Coarse-grained parallelism

Figure 2 presents the algorithm for the outer iteration. In the global loop we execute a biconjugate gradient (BiCG)

```
Assemble all local matrices in double precision

BiCG solver on the fine grid
  Preconditioner:
    MG V-cycle 1+1 on the fine grid
      Coarse grid solver:
        direct LU solver
      Smoother:
        for each macro execute local solver, Figure 3
        internal communication between macros
          in the same sub-domain
        external communication (MPI) between macros
          in different sub-domains
```

Figure 2: Coarse-grained parallelism in the outer solver.

solver preconditioned by the outer, data-parallel multigrid iteration. Enclosing multigrid solvers in Krylov subspace methods significantly improves the convergence and robustness in general, see for example Smith et al. [56]. The stabilizing effect also enables the global multigrid to perform only one pre- and postsmoothing step and to execute an inexpensive V-cycle. The small drawback of this setup is the increased storage requirement by five additional global vectors. The stopping criterion of the outermost BiCG solver is set to reduce the initial residuals by eight digits. On the global coarse grid (comprising the unrefined macros), the outer multigrid uses a direct solver based on an LU decomposition.

The outer multigrid passes its current defect $\mathbf{d}_l$ as a right hand side to the inner solver, $l \in \{10, \dots, 1\}$ is the multigrid level. Hence, the inner solver is a (local) smoother from the outer solver's point of view. The initial guess for the inner solver is always the zero vector as it works on residual systems only, so just one data vector is passed from the outer to inner solver. After smoothing by the inner solver, the correction vector $\mathbf{c}_l$ is passed back to the outer solver and is used to correct the outer defect on the current level l, and the outer multigrid continues with its restriction or prolongation.

Taking into account the minimal overlap of the data shared on *macro edges*, we should point out that the whole scheme is implemented without building up any global matrix. The only global work that is required is the direct solution of the coarse grid problem, comprised of the unrefined macros. The global defects on the different levels of the global multigrid solver are computed by exchanging and averaging values on the macro edges, only after each smoothing step of the outer multigrid. When scheduling two adjacent jobs on the same node (cf. Section 5), we take additional advantage of OpenMPI (in contrast to several other MPI implementations) being able to perform this part of the communication via shared memory. Hence, the strong global coupling of the outer multigrid is reduced to a minimum, since the smoothing is only performed locally. The implementation of the artificial inner boundary conditions in FEAST minimizes communication, but implies slightly increased local smoothing requirements.

With regard to the fine-grained parallel co-processors we target in this paper, this approach strictly distinguishes between local and global problems, and local problems can be scheduled onto the co-processors independently. No additional iterations of the outer multigrid solver are required due to the decoupled local smoothing, except for extremely high anisotropies in the local problems [24].

Depending on the hardware configuration one or more CPU or GPU jobs are scheduled to the nodes. Due to the limited solver capabilities currently implemented on the GPU, the GPU processes can only solve mildly anisotropic local problems. More difficult problems are scheduled as CPU processes. Beside this qualitative restriction, the scheduler also tries to establish a quantitative balance of the runtimes, because the CPU and GPU need a different amount of time for the same problem. Currently, we use a simple a priori scheduling based on the size and type of the sub-domain (anisotropy of the elements). As the CPU and GPU solvers are directly interchangeable, no code changes are required when executing the same program on different clusters. Only some machine parameters must be entered so that the static scheduler knows whether it may use the GPU solvers, and their performance relation to the CPU. In fact, the ratio can be derived from a serial calibration run. This arrangement is very convenient, as it guarantees the execution of the application without dedicated hardware support, but leverages the hardware when present.

Before the computation starts, the FEAST package assembles all local matrices on all multigrid levels in double precision. This is not as critical as it may seem, because all but the finest level consume only one third of the memory required for the finest level matrix. Once all data is present locally, we are able to take full advantage of fine grained parallelism.

4.2 Fine-grained parallelism

```
transform d_l to single precision

if (level l ≥ l_t)
  transfer d_l to GPU
  GPU MG F-cycle 4+4 for levels l ... l_c + 1
    and GPU preconditioned CG direct solution on level l_c
  transfer solution c_l from GPU to CPU
else
  CPU preconditioned CG computes c_l from d_l

transform c_l to double precision and pass to outer solver
```

Figure 3: Fine grained parallelism in the inner solver: GPU multigrid solver on a local macro of level l passed from the outer multigrid, cf. Figure 2.

The GPU uses a multigrid solver with a Jacobi smoother. Due to the moderate anisotropies in our test

cases and due to the slightly increased smoothing requirements by FEAST's domain decomposition approach (see Sections 4.1 and 5.1), we perform four pre- and postsmoothing steps in an F-cycle to increase the solver's robustness and to be relatively independent of the various relaxation and damping parameters in the multigrid algorithm. We found no improvement in running (more expensive) W-cycles. The entire inner solver operates in single precision. This is not a problem, as we only try to reduce the initial residual by two digits, see Section 1.4. Figure 3 provides a pseudo code listing of the inner solver on the GPU. Note that the setup is symmetric to the outer solver. The current defect $\mathbf{d}_l$ to be smoothed is provided by the outer solver and the final correction term $\mathbf{c}_l$ computed by the inner solver is consumed by the outer solver for defect correction and communication of boundary elements. The choice of the coarse grid level l_c and the CPU threshold l_t is discussed in the next section.

While the local CPU solver can operate directly with the double precision data, the GPU solver must first transform the values into a single precision representation, and then transfer them to GPU memory. Here we only discuss how we manage the data vectors, the matrix data is covered in Section 4.3.

Ideally, one would like to take advantage of asynchronous data transfer to and from the video memory, and thus overlap communication and computation. This is especially important because the precision conversion can place a significant strain on the bandwidth available to the CPU(s), in particularly with the shared memory bus in DQ's EM64T-based chipset architecture. On LiDO, the Opteron processors have independent memory buses and this is not a dominant issue. We discuss the impact of memory bus contention further in Section 5.2.

In reality, however, true asynchronous transfers are not easy to achieve. Moving data to the GPU might be asynchronous, but this is controlled by the driver and not exposed to the programmer. Recent driver revisions show a tendency to transfer the data from mapped CPU memory to GPU memory as late as possible, impeding the data flow. Asynchronous readbacks are only asynchronous on the CPU side by design, and extensive tests have revealed that they only yield faster overall performance for image data formats and not for the floating point formats that we use [23, 31]. Clearly, this is not an optimal situation, but much better performance is possible with the emerging GPGPU languages that expose the hardware more directly, see Section 4.4.

4.3 Tradeoffs in parallelism

The use of block-based data transfers and regular access patterns reduces the impact of latency due to cache misses on the CPU and GPU. Avoiding the impact of limited bandwidth is a much more difficult problem to solve in a general way. The primary issue is that a banded matrix-vector product has a very low *computational intensity*. The inner loop of the product repeatedly fetches two data items (the matrix and vector entries per band), performs a single multiply-add instruction and outputs the resulting data item. On almost all systems, the performance of this operation is limited by the memory bandwidth and not the peak computation rate. On a heterogeneous architecture the solution to this bottleneck is to move the matrix data, which is typically read multiple times in an iterative scheme, to the memory system with the highest bandwidth. In the case of our CPU-GPU cluster, the sizes of the caches on the CPU and the memory on the graphics card play a key role in helping us to schedule a particular problem. The texture caches on the GPU are significantly smaller than on the CPU and are optimized for 2D coherent neighborhood lookups, so they do not play an important role in the scheduling, as long as the access patters are coherent.

When the problem data fits easily into the L2 cache, it is assigned to the CPU since the cache has typically a higher bandwidth in comparison to graphics memory; moreover the data has to be transfered to the GPU via the comparably slow PCIe bus first. For larger problems, the decision depends on the existence, size, and bandwidth of an L3 cache. If it is still faster than GPU memory, (once again including data transfer overhead) the task is assigned to the CPU. Large problem sizes are best suited for the graphics card as the video memory is much larger than the CPU caches, and it provides a much higher bandwidth than the main memory (cf. Section 2.2, [57]). Thus, the low-level data transport considerations within our approach are similar for both the CPU and the GPU, first transfer of data to local memory, then processing of data in this higher bandwidth memory. Note, that both the transfer of data from main memory to the local memory/cache and the bandwidth from local memory/cache to the processor is higher for the CPU. However, the GPU has much more local memory (512MiB).

In the inner solver (Figure 3), this idea is implemented by the explicit comparison of the finest level l of the current macro (passed from the outer solver) with a threshold level l_t. Note that level l yields a local problem size of $(2^l+1)^2$, and in our calibration runs, we found that a threshold of $l_t = 6$ corresponding to the local problem size $4,225$ is the the best choice.

Irrespective of the initial size of the macro, the inner solver on the GPU at some stage always has to work on coarser levels in the multigrid cycle, and thus small grids. One could think of rescheduling those small inner multigrid problems that fall below the above threshold back to the faster CPU cache, and in fact the CPU would solve the problem faster. However, the comparatively slow PCIe bus would have to be crossed to reach the CPU cache. Thus, the GPU has to solve these small intermediate problems itself. But on the GPU it does not make sense to operate on very small grids, because the architecture then cannot exploit its parallelism any more. So instead of following the multigrid cycles to the coarsest grids (3x3), we stop earlier and solve a coarse grid problem with $(2^{l_c}+1)^2$ unknowns with a few iterations of a preconditioned conjugate gra-

dient solver. This is overall much faster than continuing with the multigrid transfers down to the trivial problem sizes.

The inner solver description (Figure 3) omits the conversion to single precision and transfer of the *matrices* to the GPU, because it is not performed for each outer iteration. To explain the procedure we model the 512MiB video memory on the graphics card as a 'gigantic' L3 cache for the GPU. We can use it in two different modes.

In the first mode we use *manual* data prefetching. As the video memory of the graphics cards in the cluster can easily hold three complete data sets (matrix hierarchies, iteration vectors, right hand sides etc.) at the same time, interleaving computation on one macro and transfer for the previous and next macros is the most efficient approach. In this mode, we use these available data sets in a round-robin fashion and for increased efficiency, the conversion to single precision is merged with the transfer. The practicability of this mode depends on the existence of asynchronous transfer methods for the data used, and as discussed in the previous section there are none currently for our setup. The benefit of the explicit model is clearly that the memory footprint of the single precision copies is reduced to a minimum.

In the second mode, that we apply for the matrices currently, we use the video memory as a dedicated L3 cache with *automatic* data prefetching. After the CPU has built up all matrices for all levels (see Figure 2), we transform the matrices to single precision and store them in driver-controlled memory. The disadvantage clearly is that the amount of data is much larger than the available video memory. But this approach yields faster accumulated transfer times on our hardware, as the driver can now decide on its own when data is paged in and out of the video memory, overlaying some of the computation with communication which we could not do in the manual mode (cf. Section 4.2).

4.4 Design effort and limitations

As one of our main goals has been a minimally invasive integration of the FEAST package with the GPU-based multigrid solver, it is important to understand the impact this had on the effort and the remaining limitations.

Prior to this work, the GPU application provided a serial standalone multigrid solver for a one-macro configuration. We should note that with the emerging GPGPU languages and programming environments such as Stanford's Brook [7], AMD's *close to the metal* (CTM) [49] and NVIDIA's *compute unified device architecture* (CUDA, which additionally provides a subset of the BLAS [47]), the challenge of implementing Finite Element solvers on the GPU does no longer lie in employing graphics APIs for non-graphics tasks. The real challenge is to devise efficient mappings of existing algorithms to the streaming compute paradigm with 1000s of lightweight hardware threads being executed simultaneously. See Section 1.3 for references relevant in the context of our work.

We first designed a lean interface layer (implemented in C) between FEAST (Fortran) and the GPU package (C++). The GPU package was encapsulated in a library, providing the backend for the interface. This setup allowed us to reuse the GPU code without modification. The interface provides bidirectional passing of configurations and status messages, and multigrid data is exchanged via passing pointers to the FEAST data in memory, allowing the GPU-based solver to read its input data and write back its results directly, without expensive data copying.

The changes to the infrastructure in the FEAST package were few and accounted for approximately 1000 lines of code, taking advantage of the modular design of FEAST's solver and smoother routines. The main obstacle was the coding of the previously unavailable non-uniform partitioning and scheduling of macros to the now heterogeneous parallel jobs.

The challenges we met in this somewhat natural approach were twofold (compiler and library issues are omitted): In matured packages it is more difficult to realize functionality which was not planned in the initial design. For instance, the possibility to reassign a problem from one local smoother to another based on its size was never considered in FEAST before. Thus, if the GPU code rejects a small problem (cf. Section 4.2), the CPU executes a re-implementation of the solver in C++ instead of just calling a FEAST routine. The plan is to make the Fortran CPU smoothers and solvers more accessible to external callers in the future. The main challenge was to design and implement the interface with maximum efficiency. A first functioning version was not too difficult to build, but it involved too many unnecessarily inefficient data transfers. As we strongly emphasized the importance of data transport reduction, we were finally not surprised to see that the biggest speedup from any code improvement (a factor of two) resulted from finally finding an efficient way to interleave the format conversion with the data transfer (cf. Section 4.3). Optimizations of the GPU solvers yielded a 10% improvement, and the size dependent assignment of tasks to the CPU or GPU gained almost another factor of two despite the less carefully optimized re-implementation of the FEAST code in C++.

FEAST also only recently started to offer support for different solvers on different macros. Therefore, we do not have the full flexibility here and the default build supports just two different solvers. One of the problems in implementing more generality in this context is the assignment of the solvers to the macros. While it is not very difficult to initialize more solvers, the problem is to formulate the rules upon which the assignment takes place.

A similar problem occurs with the assignment of jobs within the MPI framework when faced with heterogeneous architectures. In the absence of explicit directives, the assignment of jobs to nodes is clearly sub-optimal in these cases. For 'pure' GPU cases (which still require CPU management, see Section 1.5), as for pure CPU jobs, the assignment is simple since all jobs are equally sized. However, for mixed GPU-CPU runs, the allocation of jobs across nodes

must be managed more carefully, since MPI cannot distinguish between a 'pure' CPU job and a CPU job which is actually managing a GPU-based solver. Thus, before the insertion of explicit assignment directives, these runs often resulted in extreme imbalance between nodes and similarly poor performance.

As a concrete example, for a combined run with CPU and GPU jobs, instead of the desired partitioning of one CPU macro and one GPU macro per node, the default scheduling assigned two CPU macros each to the first half of the nodes and two GPU macros each to the second. Given that each node has only one GPU, this results in both GPU jobs competing for the same co-processor. These clearly incorrect assignments resulted in extremely poor performance, and required significant effort to remedy.

A further complication introduced by the heterogeneous architecture is load balancing between the two different processors. While some of the partitioning is dictated by the solver capabilities or memory considerations (see Section 4.3), there is still considerable leeway in the assignment of macros to either processor. Obviously, the goal is to have all macros complete simultaneously, to minimize wait times. As we shall see from the results in the next Section, the optimal partitioning of macros between GPU and CPU is both problem and size dependent, and requires considerable effort to explore the parameter space.

Here we see the main challenge for the future. FEAST will have to be extended by a dynamic scheduler which knows the characteristics of the different hardware architectures and based on different parameters of the macros decides where to execute them.

5 RESULTS

5.1 Test configurations

In our tests we solve the Poisson problem (Section 2.1) on two different grid formats. The first is a moderately anisotropic Cartesian grid. This allows the use of a the simple Jacobi smoother and exploits the 2-level ScaRC scheme to hide the anisotropies from the global solver, see Section 3.2. The second is an inhomogeneous mixture of regular and anisotropically refined macros which require a more advanced solver for convergence. Starting with some initial guess, the goal in both cases is to gain eight digits of accuracy. In all our tests, we achieve the same accuracy and error reduction irrespective of the hardware architecture used for the computation, i.e. the restriction of the GPU to single precision has no effect on the final result due to our iterative refinement technique. In addition, the number of iterations and (global) convergence rates are identical.

We run the test scenario from Section 2.1 on one master and up to 16 compute nodes of the clusters DQ and LiDO (Section 2.2). For the different test configurations we use the following notation:

$$A\mathbf{g}P\mathbf{m}\_B\mathbf{c}Q\mathbf{m}\_C\mathbf{n}\_\text{Name}\ \ \mathbf{L}L,$$

- $A \in \{0, 1\}$ is the number of GPUs used per node,
- $P \in \{0, \ldots, 12\}$ is the number of macros per GPU process (if $A = 0$ then $P\mathbf{m}$ is omitted),
- $B \in \{0, 1, 2\}$ is the number of CPUs used per node,
- $Q \in \{0, \ldots, 12\}$ is the number of macros for the CPU process(es) (if $B = 2$ then each CPU is assigned $Q/2$ macros, if $B = 0$ then $Q\mathbf{m}$ is omitted),
- $C \in \{2, 4, 8, 16\}$ is the number of compute nodes (there is always one master node in addition),
- Name $\in$ {DQ, LiDO} is the name of the cluster, and
- L is the *level* of the macros, consisting of $(2^L + 1)^2$ degrees of freedom (DOFs), all macros in one configuration have the same size.

The overall number of processes is always equal to $(A + B) \cdot C$, and the global number of unknowns can be derived from $(P + Q) \cdot C$ and the multigrid level, since all macros have the same number of DOFs. For example, 0g_2c8m_16n_LiDO means that 16 compute nodes of the LiDO cluster are used, processes are scheduled to both CPUs and every CPU treats four macros, hence, we have $8 \cdot 16$ macros distributed among $(0 + 2) \cdot 16$ processes; 1g4m_0c_16n_DQ uses 16 compute nodes of the DQ cluster, processes are scheduled only to the GPU and every GPU treats four macros; 1g7m_1c1m_8n_DQ uses eight DQ compute nodes with eight processes being scheduled to the GPUs and eight to the CPUs, treating seven and one macros, respectively. In the 1g4m_0c_16n_DQ example where no CPU is explicitly used for the solution of sub-problems, one CPU per node is implicitly occupied to some extent because they have to support the GPU in the computation and run the outer solver. A 0g_ notation on the other hand means that the GPUs are completely idle.

Recall, that this is a cascaded multigrid scheme (cf. Section 4.1). The outer multigrid solver always runs on the CPUs, and executes the inner multigrid as a smoother on the different levels for each macro. Even if all macros have the same size on the finest level, sub-problems of various sizes are generated during the outer iteration. Depending on the current problem size, GPU processes reschedule their work onto the CPU if the local problems are small (cf. Section 4.3). Thus, even if no processes are scheduled for the CPUs, they will receive more or less work from the GPU processes depending on the value of this threshold. This is also the reason, why we never execute 1g_2c configurations, as this would simply schedule too much work to the CPUs, and also congest the bus to the main memory.

Whenever some values are omitted in the above notation, we refer to all possible completions in the given context at the same time, e.g. we often consider only the hardware configuration, like 0g_2c_DQ, irrespective of the distribution and size of macros.

The test cases are limited because LiDO's nodes do not contain any GPUs, and DQ's nodes cannot accommodate two graphics cards. With newer hardware, we hope to extend the test series to these configurations in the future.

To improve the clarity of the presentation, we use graphs to show the results. The appendix contains detailed tables with all the exact numbers.

5.2 Homogeneous domains

In this section we assume that the macros discretizing the domain are homogeneous and only moderately anisotropic. This configuration is designed to be a prototype, the stronger the anisotropies are, the better the ScaRC scheme is suited compared to standard data parallel multigrid solvers, see Section 3.2 for details.

For such sub-domains a multigrid iteration with a Jacobi smoother converges very quickly, and we need only three outer BiCG iterations (hence, six global multigrid cycles) for all cases in this section, cf. Algorithm 2. A corresponding inner solver is implemented both on the CPU and GPU and, therefore, we have no restrictions in assigning the processes to the architectures, i.e. we can test all available configurations 0g_1c, 0g_2c, 1g_0c and 1g_1c. By varying the number of processors and size of the macros we obtain various problem sizes from 16 Mi to 128 Mi grid nodes.

5.2.1 CPU vs. GPU

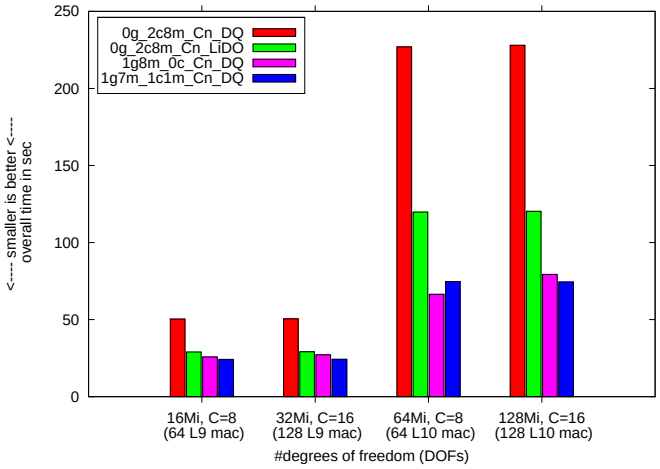

Figure 4: CPU vs. GPU cluster configurations: absolute execution time (see Table 3 in the Appendix for the exact numbers).

Figure 4 compares the performance of the DQ and LiDO clusters using their CPUs and the DQ cluster in two GPU configurations with different macro sizes and processor numbers. Problem sizes vary from 16Mi to 128Mi unknowns. First, we observe that the execution times are more similar for the smaller L9 ($(2^9+1)^2$ DOFs) macros. This is not surprising as smaller macros mean that there is less local work on each node between synchronizations and also a less favorable ratio of computation to communication. For GPUs this effect is particularly large, because, in addition to the above, GPUs have a configuration time for each macro (hardware setup and data transport) that must be amortized over the execution time. The normalized node time per DOF for the same runs (Figure 5) clearly shows that this constant overhead is less significant for the larger L10 macros, and thus the GPUs perform much more efficiently. In addition we see that the nodes perform with similar efficiency in the 8n and 16n runs; weak scalability is discussed in more detail further below.

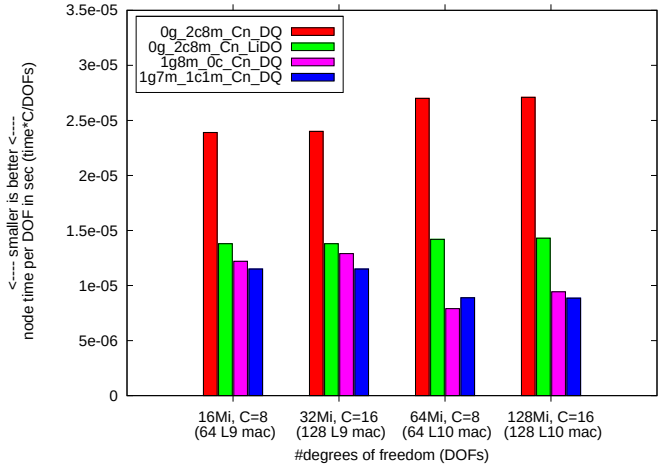

Figure 5: CPU vs. GPU cluster configurations: node time per degree of freedom (see Table 4 in the Appendix for the exact numbers).

Overall, we see that the GPU configurations on the older DQ cluster perform similarly to the newer LiDO cluster for the L9 macros, whereas the difference is clear for the L10 macro configurations. In particular, on the L10 macros the DQ cluster with GPUs solves the test problem around three times faster than without. The reason for the large difference between the CPU configurations of DQ and LiDO and the small difference between the pure GPU (1g_0c_DQ) against the combined GPU-CPU (1g_1c_DQ) configuration is addressed in the following.

5.2.2 System bandwidth

The previous figures featured comparisons of configurations with the same number of nodes. Figures 6 and 7 show execution times for the same number of processes, but not necessarily the same number of nodes. This is achieved by running 0g_2c or 1g_1c_DQ configurations on half the number of nodes than 0g_1c and 1g_0c configurations. The configurations solve problems with the same number of unknowns, but these are distributed onto more nodes in the case of the single CPU (0g_1c) or GPU (1g_0c) configurations.

While for LiDO the 0g_1c4m configuration is slower than 0g_2c8m, because of the additional external communica-

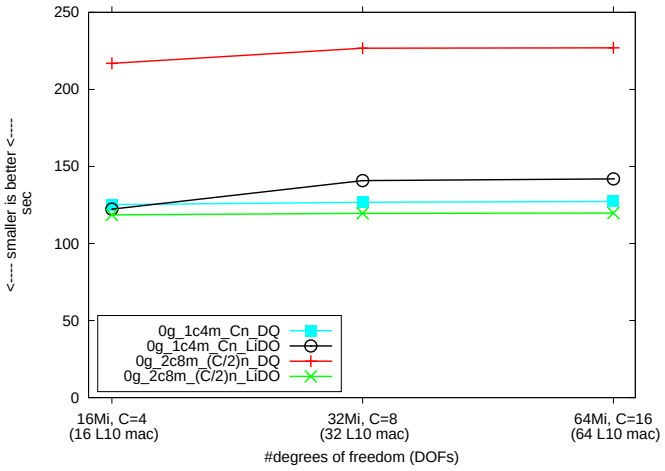

Figure 6: Impact of the system bandwidth and weak scalability (see Table 5 in the Appendix for the exact numbers).

tion (same domain across more nodes), for DQ it is the other way round and 0g_1c4m is much faster than 0g_2c8m, although all problems execute the same number of FLOPs. However, for the data intensive linear algebra operations required in the solution of sparse linear equation systems, the number of FLOPs is less important, instead the bandwidth to the processors is crucial. The chipset in DQ's nodes has a shared memory bus to the processors, such that the 0g_1c4m configuration offers the same amount of bandwidth to one processor as the 0g_2c8m configuration does to two. With twice the number of nodes in the 0g_1c4m configuration we have thus effectively doubled the overall system bandwidth and (minus the additional external communication overhead) it performs accordingly almost twice as fast as 0g_2c8m.

The chipset in LiDO's nodes has separate memory buses for the processors, so that the processors do not diminish each others effective bandwidth. Accordingly we see the opposite effect that executing on fewer nodes is faster. The importance of sufficient bandwidth support in the chipset is emphasized by the fact that LiDO loses the 0g_1c4m comparison with DQ, but resoundingly wins the 0g_2c8m race. The former loss is mainly attributed to DQ's better interconnections. The external communication itself only becomes decisive when the internal system bandwidths are relatively equal. Although the 0g_1c4m_DQ configuration with twice the number of nodes offers more system bandwidth than 0g_2c8m_LiDO ($n \cdot 2 \cdot 6.4 > n \cdot 2 \cdot 5.96$, see Section 2.2), it is slightly slower, because of the doubled amount of external communication.

The same general reasoning about system bandwidth can be applied to adding graphics cards to DQ. Their primary benefit is the sextupling of DQ's system bandwidth and not the compute parallelism in GPUs. In fact, like the CPUs, the GPUs also wait for data most of the time and could perform many more computations in the same time. Unfortunately, the additional bandwidth of the graphics card can only be reached after passing through the PCIe bottleneck. This may sound unfair, as PCIe is much faster than the previous technologies (PCI, AGP), but the unidirectional 4 GB/s (of which less than 2 GB/s are made available by the device driver [57]) are small compared to the 33.6 GB/s on the graphics board. This emphasizes again the necessity of having enough decoupled work for the GPU to take advantage of the faster processing. It also underlines the need for a closer integration of the co-processors into the system, as pursued by the Torrenza [2] and Geneseo [36] projects, see also DRC [18] and XtremeData [69] for Hypertransport couplings of FPGA accelerators and CPUs.

Note, that PCIe is a similar bottleneck to the GPU as the PCI-X based InfiniBand cards with 1.25 GB/s peak (780 MB/s benchmarked) in DQ and LiDO are a bottleneck in accessing a different CPU node. So another interpretation of the graphics cards is to see them as a specialized separate node that can only be connected to a CPU node and not among themselves (in principle graphics cards can be connected directly with DVI). However, the co-processor model discussed in Section 4.3 with the video memory interpreted as a L3 cache for the GPU is more consistent, because the GPUs still need enough support from the CPU, that it seems unjustified to speak of the graphics cards as separate nodes.

Figure 7 shows that the effective gain in system bandwidth through the addition of the GPUs is more than 3.2, approximately half the theoretical value of $6.25 = (33.6+6.4)/6.4$ (cf. Section 2.2). Here, the combined GPU-CPU (1g_1c_DQ) configurations perform only half as well, because they have only half the number of nodes (thus bandwidth) available.

The bandwidth to the CPUs is only a small fraction ($\sim 1/6$) of the system bandwidth once the graphics cards have been added. Thus, only a small advantage can be gained by scheduling some of the work and running the CPU in parallel to the GPU, see Figure 4. On the DQ cluster this advantage is particularly small because the bus is shared, and the second CPU is already using it by supporting the GPU process (cf. Section 4.3). Thus, only the remaining bandwidth fraction can lead to further acceleration. On the LiDO cluster the CPU bandwidth would contribute more to the system bandwidth as the second CPU supporting the GPU process would have its own bus.

The dominance of the contribution to the system bandwidth provided by the graphics cards can be also seen in Figure 7, by realizing that the most extreme balancing of macros towards the GPU (1g7m_1c1m_DQ) is faster than the more balanced 1g6m_1c2m_DQ configuration with two macros on the CPU.

5.2.3 Weak scalability

Figures 6 and 7 demonstrate the weak scalability of our solvers on the two clusters. We increase the problem size from 16Mi to 64Mi degrees of freedom while also doubling the number of nodes in each step. Overall, we see in Figure 6 that all configurations scale very well. While

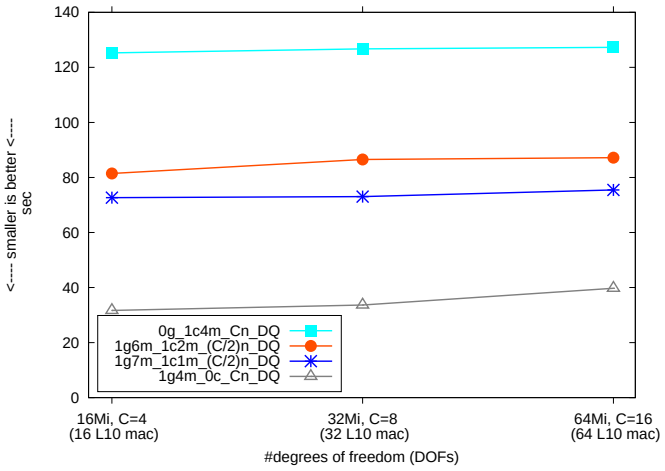

Figure 7: Weak scalability and impact of the system bandwidth (see Table 6 in the Appendix for the exact numbers).

the graph of the 0g_1c4m_DQ configuration is totally flat, the connections in the LiDO cluster are a bit slower and cause a slight increase of execution time in case of the 0g_1c4m_LiDO configurations, where only four macros reside on each node. But when we double the amount of local data (0g_2c8m_LiDO) the LiDO graph is also flat. The 2c_2n setup forms a bit of a special case with only two compute nodes communicating with each other, but only the faster connections on the DQ cluster can exploit this fact for non-proportionally faster execution.

The graphs in Figure 7 demonstrate very good scalability of the GPU configurations from 4 to 16 nodes. This encourages us to look out for opportunities to execute this test scenario on clusters with more GPU equipped nodes in the future.

5.3 Heterogeneous domains

In this test series, we demonstrate the viability of using different hardware solvers for different sub-domains. The idea is to use both CPU and GPU for what they are best suited for, the CPU is assigned few, irregular macros and the GPU is assigned many, regular macros. We thus demonstrate that we do not lose any functionality present in the original FEM package by adding GPUs as co-processors; rather we gain speedups by leveraging the additional bandwidth.

The test domain is square-sized, with anisotropic refinement towards the boundaries, which in practice is often needed to accurately resolve boundary layers, e.g. in fluid dynamics. The adaptive refinement causes the simple Jacobi multigrid to diverge, even with a high amount of smoothing steps. We again solve the Poisson problem as explained in Section 2.1 with L9 and L10 macros, yielding 6Mi to 192Mi degrees of freedom. We use up to 16 compute nodes of the DQ cluster for these configurations.

On the CPU, we employ a powerful alternating directions smoother of type line-wise tridiagonal Gauss-Seidel (ADI-TRIGS) for the inner multigrid. As previous analysis has shown [24], two pre- and post-smoothing steps suffice to treat almost any kind of ill-condition in the local matrices caused by anisotropic mesh refinement, up to the limitations of the double precision floating point format. The GPU continues to use simple Jacobi with four pre- and post-smoothing steps of the inner multigrid, respectively. Note that it is only possible to perform different amounts of smoothing steps per macro because of the strong decoupling, as explained in Section 3.2.

A conventional CPU package would perform this computation with the ADI-TRIGS smoother alone, with an equal distribution of the macros to the jobs. In the graph below, this setup is labeled 0g_2c12m_Cn_DQ.

The equivalent 1g_1c configuration requires very careful load balancing, because a different amount of work is performed on the GPU and CPU.

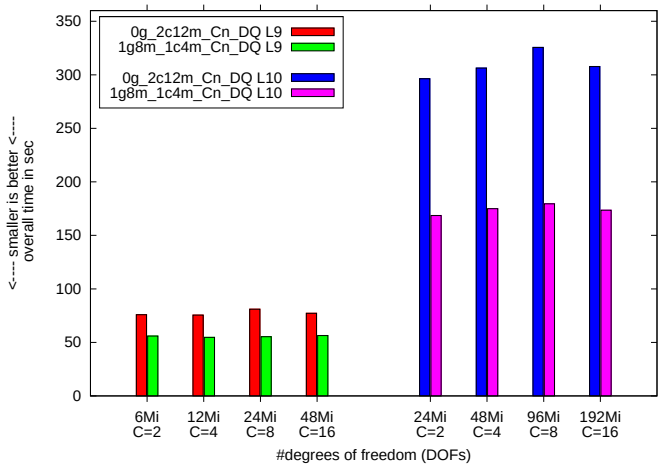

Figure 8: Absolute execution time for the heterogeneous domain, speedup and weak scalability (see Table 7 in the Appendix for the exact numbers).

Figure 8 shows the absolute timing of these two solvers for L9 and L10 macros. Except a small deviation of the eight node runs, both the CPU only and the CPU-GPU configuration show good weak scalability with increasing problem size. The CPU-GPU configuration is 1.4 times faster for L9 macros than the traditional CPU approach. The GPUs excel for the large L10 macros, where we observe an overall speedup by a factor of two, in line with the discussion of the results in Sections 5.2.1, 5.2.2 and 5.2.3.

5.4 Cluster up- and downgrade

The use of GPUs as scientific co-processors demonstrated in the previous sections opens alternatives to traditional approaches of cluster up- and downgrading. In this section we evaluate two practical scenarios not only with respect to performance but also in view of the aspects of costs, space and power requirements. The analysis uses the numbers from Section 2.2. The starting point in both cases is a cluster of eight DQ type of nodes without the GPUs. In

the first scenario we are concerned with strong scalability of the system, in the second with downgrading the size of the cluster (to free up the space and save power) while retaining its performance.

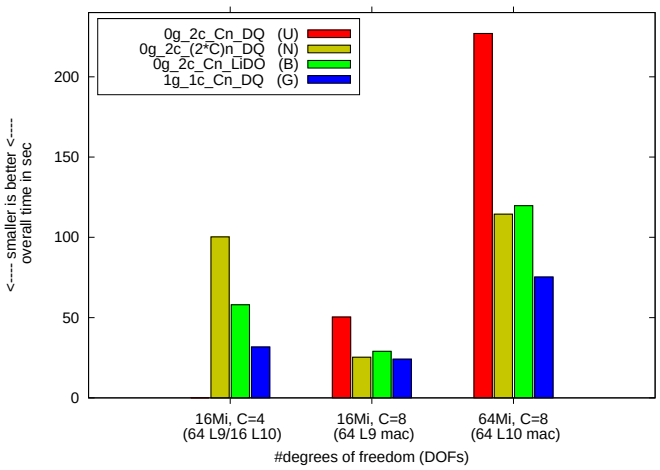

Figure 9: Cluster up- and downgrading: absolute execution time (see Table 8 in the Appendix for the exact numbers).

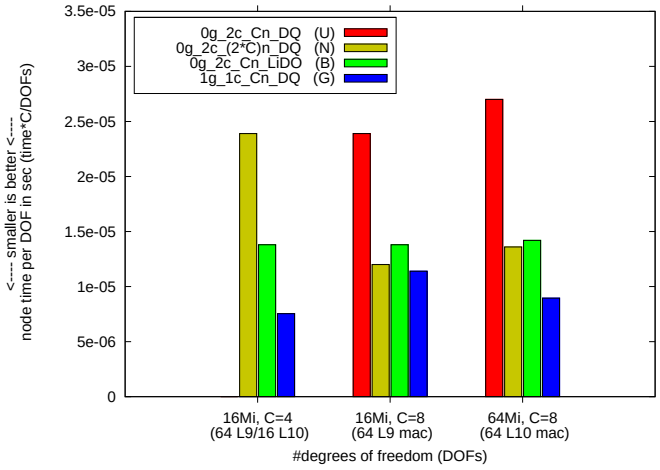

Figure 10: Cluster up- and downgrading: node time per degree of freedom (see Table 9 in the Appendix for the exact numbers).

5.4.1 Upgrade

Given a cluster of eight DQ type of nodes, what is the best procedure to increase its performance on a given set of problems? We analyze the following options:

- **U - unchanged**: The cluster configuration remains unchanged.
- **N - more nodes**: Double the number of the existing nodes.
- **B - better nodes**: Replace the DQ type nodes by LiDO type nodes.
- **G - insert GPUs**: Insert powerful GPUs into the existing nodes.

The middle and right comparisons in Figures 9 and 10 (absolute and normalized numbers respectively) show the impact on the performance of the approaches on different problem sizes with L9 (16Mi DOFs) and L10 macros (64Mi DOFs), respectively. As discussed in Section 5.2.1, the smaller problems (L9 macros) with less local data and shorter times between synchronizations produce more relative overhead and offer less opportunities for the GPU to reduce the overall runtime. Still the G option already surpasses the configuration with doubled nodes (N) and more clearly outperforms the B option, with better nodes (Figures 9 and 10 in the middle).

On large problem sizes with L10 macros the GPUs gain additional speedups. Now, there is enough local work to justify the data transfer through the PCIe bottleneck to the superior bandwidth bus on the graphics card, and the G option is clearly better, 3.02 times faster than the original cluster. The N options performs (not surprisingly) twice as fast, and the B option comes close to it by improving the speedup to a 1.90 factor. Figure 10 clearly shows how the efficiency of the GPU improves for the larger problems while it hardly changes for the CPU configurations.

Table 1: Evaluation of the different upgrade options (Section 5.4.1) under various aspects. Arrows indicate if larger or smaller numbers are better, boldface indicates the best case.

Aspect	**better**	**U**	**N**	**B**	**G**
Performance on small problems	↑	1.00	1.99	1.74	**2.09**
Performance on large problems	↑	1.00	1.98	1.90	**3.02**
Space requirements	↓	**1.00**	2.00	**1.00**	**1.00**
Acquisition costs	↓	**0.00**	1.00	0.95	0.35
Average power under full load	↓	**1.00**	2.00	1.11	1.30
Av. performance / cost	↑	-	1.99	1.92	**7.03**
Av. performance / power	↑	1.00	1.00	1.64	**1.97**

Performance is only one aspect in the consideration of a cluster upgrade. Table 1 puts some other aspects in relation, with bold numbers denoting the best value in each aspect. We see that the G option wins in most categories and often with a very significant margin. Even doubling the number of nodes (N) cannot match the performance of the GPUs, and in terms of acquisition costs N and B are clearly worse. Only the the power consumption would be

slightly smaller with the newer nodes and performance on small problems is similar on twice as many nodes. However, the comparable GPU performance applies only to the strict strong scalability when exactly one problem has to be solved fast. As soon as two or more of the small problems are to be solved independently (e.g. different problem parameter settings), then there would be more local data, and using half of the GPU nodes for each of the problems would deliver a clearly superior performance as demonstrated in the next section.

When we analyse the combined benefits through the performance/cost and performance/power ratios in Table 1 the advantage of the G option for the upgrade becomes even more obvious. Note, that the performance/cost ratio refers to an upgrade scenario, i.e. the eight DQ nodes are already present and only the additional costs are considered. If a completely new system is assembled then one would obviously opt for the faster, newer nodes and evaluate the GPUs there, but we could not test this because LiDO has no GPUs.

5.4.2 Downgrade

The scenario discussed in this Section is to downgrade the size of the cluster while minimizing the effect on its performance. Given a cluster of eight DQ type of nodes, how much performance can we achieve with only four nodes? Again we look at three options.

- **U - unchanged**: The cluster configuration remains unchanged.
- **N - fewer nodes**: Reduce the number of nodes in the cluster by half.
- **B - better nodes**: Replace the nodes with half the number of LiDO type nodes.
- **G - insert GPUs**: Remove half the cluster and place powerful GPUs in the remaining nodes.

The middle and left comparison in Figures 9 and 10 shed light on the performance of the downgraded systems according to the above options. Obviously, once we have downgraded the cluster to fewer nodes, we will have less memory available and will not be able to solve the large problems anymore. Therefore, we compare the performance of the new configurations with the original L9 results (red bar in the middle of figures). Note, that in the previous section option N was doubling the number of nodes and now it is halving them, thus in case of the left comparison (C=4) the notation 0g_2c_(2*C)n_DQ in the legend with 2*C suggesting a node doubling is inappropriate, all configurations in the left comparison use four nodes.

Simply halving the number of nodes degrades the performance by almost the same factor. The B option does a better job and achieves 87% of the previous performance. The G option, however, actually outperforms the original system by 59%, despite having only half the number of nodes. This is possible because the efficiency on the larger macros is high, see Figure 10. For the CPU only runs we tried both possible macro configurations giving 16Mi DOFs on four nodes: 16 L9 macros or 4 L10 macros per node and use the former which is slightly faster.

Table 2: Evaluation of the different downgrade options (Section 5.4.2) under various aspects. Arrows indicate if larger or smaller numbers are better, boldface indicates the best case.

Aspect	**better**	**U**	**N**	**B**	**G**
Performance on small problems	↑	1.00	0.50	0.87	**1.59**
Space requirements	↓	1.00	**0.50**	**0.50**	**0.50**
Acquisition costs	↓	**0.00**	**0.00**	0.48	0.18
Average power under full load	↓	1.00	**0.50**	0.56	0.65
Performance / cost	↑	-	-	1.81	**8.83**
Performance / power	↑	1.00	1.00	1.55	**2.45**

As in the upgrading case the G option wins most categories in the comparison (Table 2). For very low acquisition costs it is the only configuration that can preserve and even surpass the previous performance level of the original cluster with twice as many nodes. The very high performance/cost ratio emphasizes this fact. As explained in Section 5.4.1 we only consider additional costs, so not buying anything gives an infinite ratio, but this would fail the main goals of reducing the space requirement and retaining the performance. Overall power consumption would be slightly lower with the better nodes (B), but the performance/power ratio is much in favor of the GPU solution again (G).

5.4.3 Massively parallel systems

We understand that the above discussion would be more exciting if the initial configuration did not consist of eight nodes, but rather 128 or even 4096 and we were discussing the corresponding upgrades or downgrades by factors of two. If we assume the same performance relations as discussed above then the aspects of costs, space and power have a much larger impact on these scales, as they clearly stand out of the noise present in these aspects and influences of additional factors. A problem in performing such a study is the lack of large clusters equipped with current graphics hardware technology. On the other hand, studies such as this one are necessary to demonstrate that this type of approach to system acceleration is feasible and would warrant an application on large scales. Increasingly larger GPU systems are already being employed and we will consider larger scale computations in the future.

6 CONCLUSIONS AND FUTURE WORK

We have tackled the difficult problem of efficient utilization of heterogeneous processors with different computing paradigms in a commodity based cluster. More and more hardware accelerators based on standard hardware components offer very high performance in their application areas, but the challenge lies in the integration of these advantages into established software environments. We have accepted the challenge and designed an interface to graphics processors (GPUs) for the parallel Finite Element package FEAST. Rather than sacrificing modularity for ultimate performance, a minimally invasive integration of the GPU as a scientific co-processor has been pursued.

With a lean interface to the FEAST package we obtain a modular and extensible framework which delivers speedups of two to three for sufficiently large problems. Detailed analysis of the system bandwidth as the most important performance factor for linear system solvers is provided and supported by results. Although the GPU itself is fairly restricted in data handling and precision, we maintain the same functionality and result accuracy as in the original FE package. The restrictions are encapsulated in the implementation of the interface and do not affect the application programmer who benefits from the hardware accelerated solvers by simple changes to configuration files, without the necessity for any code changes.

We have analyzed an up- or downgrade of a cluster through GPUs in comparison to standard approaches of varying the number or substituting newer nodes. The GPUs are clearly dominant in the performance and costs aspects, but also perform comparably well with respect to space and power requirements. While only modified software can benefit from the additional GPU power, we have demonstrated that providing hardware acceleration for a large software package ($>$ 100.000 lines of code) may be achieved by touching 1% of the code basis, and providing a device-specific implementation of the (inner) solver.

The most challenging task in the future will be to devise a dynamic scheduling scheme, which takes into consideration the condition of the sub-problems, the expected runtime, the availability and suitability of the co-processors and the bandwidth of and between the different processors.

We plan on extending the GPU-based code with more advanced smoothing operators and coarse grid solvers. More tradeoffs in parallelism, in particular a better balance between the bandwidth and processing power of GPU's, are necessary. Ultimately, more of the emerging commodity floating point co-processors need to be evaluated and integrated into our system.

ACKNOWLEDGMENT

We thank Sven Buijssen, Matthias Grajewski and Thomas Rohkämper for co-developing FEAST and its tool chain. Thanks to Michael Köster, David Daniel, John Patchett, Jason Mastaler, Brian Barrett and Galen Shipman for help in debugging and tracking down various cluster software and hardware issues. Also thanks to NVIDIA and AMD for donating hardware that was used in developing the serial version of the GPU backend.

This research has been partly supported by a Max Planck Center for Visual Computing and Communication fellowship, by the German Science Foundation (DFG), project TU102/22-1 and by the U.S. Department of Energy's Office of Advanced Scientific Computing Research.

DETAILED TEST RESULTS

Table 3: CPU vs. GPU cluster configurations: absolute execution time.

#DOFs	C	#macros	0g_2c8m_Cn_DQ	0g_2c8m_Cn_LiDO	1g8m_0c_Cn_DQ	1g7m_1c1m_Cn_DQ
16Mi	8	64 (L9)	50.36	28.98	25.75	24.12
32Mi	16	128 (L9)	50.51	29.12	27.19	24.21
64Mi	8	64 (L10)	226.96	119.72	66.40	74.69
128Mi	16	128 (L10)	227.98	120.25	79.29	74.37

Table 4: CPU vs. GPU cluster configurations: node time per degree of freedom.)

#DOFs	C	#macros	0g_2c8m_Cn_DQ	0g_2c8m_Cn_LiDO	1g8m_0c_Cn_DQ	1g7m_1c1m_Cn_DQ
16Mi	8	64 (L9)	2.39e-05	1.38e-05	1.22e-05	1.15e-05
32Mi	16	128 (L9)	2.40e-05	1.38e-05	1.29e-05	1.15e-05
64Mi	8	64 (L10)	2.70e-05	1.42e-05	7.90e-06	8.89e-06
128Mi	16	128 (L10)	2.71e-05	1.43e-05	9.43e-06	8.85e-06

Table 5: Impact of the system bandwidth and weak scalability.

#DOFs	C	#macros	0g_1c4m_Cn_DQ	0g_1c4m_Cn_LiDO	0g_2c8m_(C/2)n_DQ	0g_2c8m_(C/2)n_LiDO
16Mi	4	16 (L10)	125.25	122.29	216.88	118.58
32Mi	8	32 (L10)	126.71	140.83	226.73	119.55
64Mi	16	64 (L10)	127.26	141.90	226.96	119.72

Table 6: Weak scalability and impact of the system bandwidth.

#DOFs	C	#macros	0g_1c4m_0gCn_DQ	1g6m_1c2m_(C/2)n_DQ	1g7m_1c1m_(C/2)n_DQ	1g4m_0c_Cn_DQ
16Mi	4	16 (L10)	125.25	81.45	72.53	31.68
32Mi	8	32 (L10)	126.71	86.54	73.03	33.64
64Mi	16	64 (L10)	127.26	87.20	75.47	39.74

Table 7: Absolute execution time for the heterogenous domain, speedup and weak scalability.

#DOFs	C	#macros	0g_2c12m_Cn_DQ	1g8m_1c4m_Cn_DQ
6Mi	2	24 (L9)	75.9	55.9
12Mi	4	48 (L9)	75.5	54.7
24Mi	8	96 (L9)	81.0	55.3
64Mi	16	192 (L9)	77.2	56.4
24Mi	2	24 (L10)	296.3	168.5
48Mi	4	48 (L10)	306.3	174.9
96Mi	8	96 (L10)	325.6	179.5
192Mi	16	192 (L10)	307.7	173.5

Table 8: Cluster up- and downgrading: absolute execution time.

#DOFs	C	#macros	0g_2c_Cn_DQ (U)	0g_2c_(2·C)n_DQ (N)	0g_2c_Cn_LiDO (B)	1g_1c_Cn_DQ (G)
16Mi	4	64 (L9)/ 16 (L10)		100.25	57.96	31.68
16Mi	8	64 (L9)	50.36	25.29	28.98	24.09
64Mi	8	64 (L10)	226.96	114.42	119.72	75.23

Table 9: Cluster up- and downgrading: node time per degree of freedom.

#DOFs	C	#macros	0g_2c_ Cn_DQ (U)	0g_2c_ (2·C)n_DQ (N)	0g_2c_ Cn_LiDO (B)	1g_1c_ Cn_DQ (G)
16Mi	4	64 (L9) / 16 (L10)		2.39e-05	1.38e-05	7.54e-06
16Mi	8	64 (L9)	2.39e-05	1.20e-05	1.38e-05	1.14e-05
64Mi	8	64 (L10)	2.70e-05	1.36e-05	1.42e-05	8.95e-06

REFERENCES

[1] AGEIA Technologies, Inc. (2006). AGEIA PhysX processor. http://ageia.com/physx/index.html.

[2] AMD, Inc. (2006). Torrenza technology. http://enterprise.amd.com/us-en/AMD-Business/Technology-Home/Torrenza.aspx.

[3] Becker, C. (2007). *Strategien und Methoden zur Ausnutzung der High-Performance-Computing-Ressourcen moderner Rechnerarchitekturen für Finite Element Simulationen und ihre Realisierung in FEAST (Finite Element Analysis & Solution Tools)*. PhD thesis, Universität Dortmund, Fachbereich Mathematik.

[4] Becker, C., Kilian, S., and Turek, S. (2007). FEAST – Finite element analysis and solution tools. http://www.feast.uni-dortmund.de.

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

[6] Bondalapati, K. and Prasanna, V. K. (2002). Reconfigurable computing systems. *Proceedings of the IEEE*, 90(7):1201–1217.

[7] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786.

[8] ClearSpeed Technology, Inc. (2006). ClearSpeed Advance Accelerator Boards. www.clearspeed.com/products/cs_advance/.

[9] Colella, P., Dunning, T. H., Gropp, W. D., and Keyes, D. E. (2003). A science–based case for large–scale simulation. Technical report, DOE Office of Science. http://www.pnl.gov/scales.

[10] Compton, K. and Hauck, S. (2002). Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210.

[11] Cray Inc. (2006). Cray XD1 supercomputer. www.cray.com/products/xd1.

[12] Dally, W. J., Hanrahan, P., Erez, M., Knight, T. J., Labonté, F., Ahn, J.-H., Jayasena, N., Kapasi, U. J., Das, A., Gummaraju, J., and Buck, I. (2003). Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35.

[13] Davis, T. A. (2004). A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):165–195.

[14] DeHon, A. (2002). Very large scale spatial computing. *Lecture Notes in Computer Science, Proceedings of the Third International Conference on Unconventional Models of Computation*, 2509:27–36.

[15] Demmel, J., Hida, Y., Kahan, W., Li, X. S., Mukherjee, S., and Riedy, E. J. (2006). Error bounds from extra-precise iterative refinement. *ACM Transactions on Mathematical Software*, 32(2):325–351.

[16] Douglas, C. C., Hu, J., Karl, W., Kowarschik, M., Rüde, U., and Weiß, C. (2000a). Fixed and adaptive cache aware algorithms for multigrid methods. In Dick, E., Riemslagh, K., and Vierendeels, J., editors, *Multigrid Methods VI*, volume 14, pages 87–93. Springer.

[17] Douglas, C. C., Hu, J., Kowarschik, M., Rüde, U., and Weiß, C. (2000b). Cache optimization for structured and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40.

[18] DRC Computer Corporation (2006). DRC Reconfigurable Processor Units. http://www.drccomputer.com/drc/modules.html.

[19] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. (2004). GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47.

[20] Fung, J. and Mann, S. (2004). Using multiple graphics cards as a general purpose parallel computer: Applications to computer vision. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004)*, volume 1, pages 805–808.

[21] Garg, R. P. and Sharapov, I. (2001). *Techniques for Optimizing Applications: High Performance Computing*. Sun Microsystems Inc.

[22] Geddes, K. O. and Zheng, W. W. (2003). Exploiting fast hardware floating point in high precision computation. In *ISSAC '03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 111–118.

[23] Göddeke, D. (2006). GPGPU coding tutorials. Technical report, University of Dortmund, Institute of Applied Mathematics and Numerics. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/.

[24] Göddeke, D., Strzodka, R., and Turek, S. (2007). Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):221–256.

[25] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. (2003). A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, pages 102–111.

[26] Govindaraju, N. K., Sud, A., Yoon, S.-E., and Manocha, D. (2003). Interactive visibility culling in complex environments using occlusion-switches. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D Graphics*, pages 103–112.

[27] GPGPU (2007). General-purpose computation using graphics hardware. `http://www.gpgpu.org`.

[28] Gropp, W. D., Kaushik, D. K., Keyes, D. E., and Smith, B. F. (2001). High performance parallel implicit CFD. *Parallel Computing*, 27:337–362.

[29] Guo, Z., Najjar, W., Vahid, F., and Vissers, K. (2004). A quantitative analysis of the speedup factors of FPGAs over processors. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 162–170.

[30] Harris, M. (2005). Mapping computational concepts to GPUs. In Pharr, M., editor, *GPUGems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 31, pages 493–508. Addison-Wesley.

[31] Harrison, O. and Waldron, J. (2007). Optimising data movement rates for parallel processing applications on graphics processors. In *Proceedings of the 25th International Conference Parallel and Distributed Computing and Networks (PDCN 2007)*, pages 251–256.

[32] Hartenstein, R. (2001). A decade of reconfigurable computing: A visionary retrospective. In *Design, Automation and Test in Europe 2001, Proceedings*, pages 642–649.

[33] Hartenstein, R. (2003). Data-stream-based computing: Models and architectural resources. In *International Conference on Microelectronics, Devices and Materials (MIDEM 2003)*.

[34] Hoßfeld, F. (2001). Perspektiven für Supercomputer–Architekturen. Technical report, FZ Jülich - Zentralinstitut für Angewandte Mathematik.

[35] IEC (2000). *Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*, second edition.

[36] Intel, Inc. (2006). Geneseo: PCI Express technology advancement. `http://www.intel.com/technology/pciexpress/devnet/innovation.htm`.

[37] Keyes, D. E. (2002). Terascale implicit methods for partial differential equations. *Contemporary Mathematics*, 306:29–84.

[38] Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Owens, J. D., and Towles, B. (2003). Exploring the VLSI scalability of stream processors. In *Proceedings of the Ninth Symposium on High Performance Computer Architecture*.

[39] Kilian, S. (2001). *ScaRC: Ein verallgemeinertes Gebietszerlegungs-/Mehrgitterkonzept auf Parallelrechnern*. PhD thesis, Universität Dortmund, Fachbereich Mathematik.

[40] Kornhuber, R., Periaux, J., Widlund, O. B., Hoppe, R., Pironneau, O., and Xu, J., editors (2005). *Domain Decomposition Methods in Science and Engineering*, volume 40 of *Lecture Notes in Computational Science and Engineering*. Springer.

[41] Kowarschik, M., Weiß, C., Karl, W., and Rüde, U. (2000). Cache–aware multigrid methods for solving poissons equation in two dimensions. *Computing*, 64:381–399.

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

[43] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., and Dongarra, J. (2006). Tools and techniques for performance – exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 113.

[44] Li, X. S., Demmel, J. W., Bailey, D. H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S. Y., Kapur, A., Martin, M. C., Thompson, B. J., Tung, T., and Yoo, D. J. (2002). Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software*, 28(2):152–205.

[45] Martin, R., Peters, G., and Wilkinson, J. (1966). Handbook series linear algebra: Iterative refinement of the solution of a positive definite system of equations. *Numerische Mathematik*, 8:203–216.

[46] Meuer, H., Strohmaier, E., Dongarra, J. J., and Simon, H. D. (2007). Top500 supercomputer sites. `http://www.top500.org/`.

[47] NVIDIA Corporation (2007). NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda`.

[48] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113.

[49] Peercy, M., Segal, M., and Gerstmann, D. (2006). A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications*.

[50] Pham, D., Asano, S., Bolliger, M., Day, M. N., Hofstee, H. P., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., and Yazawa, K. (2005). The design and implementation of a first-generation CELL processor. In *Solid-State Circuits Conference 2005, Digest of Technical Papers*, pages 184–592 Vol. 1.

[51] Rüde, U. (1999). Technological trends and their impact on the future of supercomputers. In Bungartz, H.-J., Durst, F., and Zenger, C., editors, *High Performance Scientific and Engineering Computing*, volume 8 of *Lecture notes in Computational Science and Engineering*, pages 459–471.

[52] Rumpf, M. and Strzodka, R. (2005). Graphics processor units: New prospects for parallel computing. In Bruaset, A. M. and Tveito, A., editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 89–134. Springer.

[53] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W., and Moore, C. R. (2003). Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *ACM SIGARCH Computer Architecture News*, 31(2):422–433.

[54] SEMATECH (2006). International technology roadmap for semiconductors (ITRS). `http://www.sematech.org/corporate/annual`.

[55] Sheaffer, J. W., Luebke, D. P., and Skadron, K. (2007). A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In Aila, T. and Segal, M., editors, *Graphics Hardware 2007*, pages 55–64.

[56] Smith, B. F., Bjørstad, P. E., and Gropp, W. D. (1996). *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press.

[57] Stanford University Graphics Lab (2006). GPUbench – how much does your GPU bench? `http://graphics.stanford.edu/projects/gpubench/results`.

[58] Strzodka, R. (2004). *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University of Duisburg-Essen.

[59] Strzodka, R., Doggett, M., and Kolb, A. (2005). Scientific computation for simulations on programmable graphics hardware. *Simulation Modelling Practice and Theory, Special Issue: Programmable Graphics Hardware*, 13(8):667–680.

[60] Strzodka, R., Droske, M., and Rumpf, M. (2003). Fast image registration in DX9 graphics hardware. *Journal of Medical Informatics and Technologies*, 6:43–49.

[61] Suh, J., Kim, E.-G., Crago, S. P., Srinivasan, L., and French, M. C. (2003). A performance analysis of PIM, stream processing, and tiled processing on memory-intensive signal processing kernels. In DeGroot, D., editor, *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, Computer Architecture News, pages 410–421.

[62] Taylor, M. B., Kim, J. S., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S. P., and Agarwal, A. (2002). The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35.

[63] Toselli, A. and Widlund, O. B. (2004). *Domain Decomposition Methods - Algorithms and Theory*, volume 34 of *Springer Series in Computational Mathematics*. Springer.

[64] Turek, S. (1999). *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach*. Springer, Berlin.

[65] Turek, S., Becker, C., and Kilian, S. (2003). Hardware–oriented numerics and concepts for PDE software. *Future Generation Computer Systems*, 22(1-2):217–238.

[66] Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

[67] Wilkes, M. (2000). The memory gap (keynote). In *Solving the Memory Wall Problem Workshop*. `http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf`.

[68] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., and Yelick, K. (2006). The potential of the Cell processor for scientific computing. In *CF '06: Proceedings of the ACM International Conference on Computing Frontiers*, pages 9–20.

[69] XtremeData, Inc. (2006). The XD1000 FPGA Coprocessor Module for Socket 940. `http://www.xtremedatainc.com/Products.html`.

[70] Zielke, G. and Drygalla, V. (2003). Genaue Lösung linearer Gleichungssysteme. *GAMM-Mitteilungen*, 2(1):7–107.