Integrating GPUs as fast co-processors into the existing parallel FE package FEAST

Dominik Göddeke*
Universität Dortmund
dominik.goeddeke@math.uni-dortmund.de

Christian Becker*
christian.becker@math.uni-dortmund.de

Stefan Turek*
ture@featflow.de

Abstract

We report on our experiences with integrating GPUs as fast, parallel floating-point co-processors into the parallel FE package FEAST. Since a full re-implementation of such a package is not feasible, we identify the smoothing of an outer domain-decomposition multigrid solver as a natural entry-point for a minimally invasive integration of GPUs. We address the issue of limited computational precision with a mixed precision iterative refinement approach and present preliminary timing results on a commodity cluster enhanced with GPUs.

1 Introduction

Commodity graphics processors (GPUs) have evolved into a very attractive hardware platform for general purpose computations due to their extremely high floating point processing performance, huge memory bandwidth and their comparatively low cost [OLG+05]. However, modern GPUs have several limitations: While the on-chip bandwidth is comparable to L2 cache bandwidth on CPUs, the off-chip bandwidth over the PCIe bus to the main memory can be a significant bottleneck, delivering sustained rates of one to two GB/s only. Programming GPUs requires significant background knowledge in computer graphics, and algorithms have to be adapted and reformulated to the data-stream based programming model. Finally, GPUs only provide quasi-IEEE 32-bit single floating point precision which is insufficient for most applications.

Typical FE packages, especially parallel ones, consist of up to several hundred thousand lines of code, and a complete redesign to incorporate GPUs is out of question. We therefore provide a *minimally invasive integration* of the fine-grained parallelism of GPUs as co-processors into the coarse-grained parallelism of the existing MPI-based FE package FEAST [BKT03].

*Angewandte Mathematik und Numerik, Fachbereich Mathematik, D-44 227 Dortmund, Germany

2 Background

Designed for the solution of large-scale PDE problems, FEAST combines the advantages of *multigrid* and *domain decomposition* methods into a generalised solver scheme called SCARC (*Scalable Recursive Clustering*). The computational domain is discretised into a collection of so-called *macros*, forming an unstructured coarse grid of quadrilaterals, and each macro is subsequently refined independently into *generalised tensor-product* grids. On the resulting computational grid, a global parallel multigrid solver is executed, implemented using MPI. FEAST provides a wide selection of smoothers, e.g. simple Jacobi iteration for cartesian sub-grids, and alternating-directions tri-diagonal Gauss-Seidel (ADI-TRIGS) for more anisotropic ones. For large-scale problems FEAST allows to recursively smooth the defects of the global multigrid with local multigrid solvers acting on the refined macros only. The main idea behind this approach is to avoid deterioration of the convergence of the global solver by hiding (encapsulating) anisotropies locally, while exploiting regular structures to leverage high computational efficiency. We refer to Turek et al. for more details [BKT03].
GPUs only provide quasi-IEEE single precision floating point storage and arithmetic. For the kind of FEM problems we are interested in, this low precision is insufficient. Instead of emulating double precision on the GPU through a double-single approach, we apply a mixed precision iterative refinement technique: While all global data is held in double precision on the CPU, the inner multigrid is restricted to single precision and the result of the smoothing step is accumulated in double precision to the global result. We refer to our hardware–oriented study of these techniques in the FEM context [GST06], and note that such an approach does not deteriorate the accuracy of the computed results, especially for very ill-conditioned problems due to operator or mesh anisotropies.

3 Integration into FEAST

The key observation for our intended integration of GPUs into FEAST is that FEAST maintains a clear distinction between local and global problems, and the local (sub-) problems are all structured and therefore suitable to be offloaded to co-processor hardware. More generally, this approach can similarly be applied to every FE package that maintains a similar distinction (which most FE packages do for performance reasons). Consequently, our GPU-based multigrid implementation is integrated into FEAST as a new type of smoother for the outer multigrid, with a narrow interface: FEAST builds up and maintains all global and local data, and a local problem (including a full multigrid matrix hierarchy) is passed to the GPU along with the current right hand side and some configuration data. This means the data is duplicated into GPU memory, and the GPU-based solver then smoothes this defect by means of multigrid. Finally, the defect correction term is transferred back to the CPU which accumulates it to the local defect before communicating and continuing with the global multigrid.
A first prototype of the admittedly straightforward idea to delegate the smoothing of the outer, global multigrid to a GPU-based inner, local multigrid performed rather poorly compared to what we expected after benchmarking the individual components separately. We

identified three main bottlenecks in the initial design which we address subsequently:
GPUs perform poorly for small problem sizes, mainly because of the overhead their use implies and due to unsaturated pipelines with too few threads working concurrently. As the outer multigrid employs F-cycles, most of the calls to its smoother are performed for small problem sizes. We therefore introduced a dynamic CPU-GPU switch: If a local problem is too small to be smoothed efficiently on the GPU, we simply reschedule it to the CPU. Dynamically exchanging the local smoother was however not considered in FEAST before, so we implemented a single-precision version of the GPU code on the CPU as well to hide the target decision from FEAST. The actual choice of the threshold depends on the cache sizes on the CPU and transfer overhead of the GPU.
Our GPU-based multigrid currently only offers Jacobi smoothing, whereas FEAST provides a wide range of smoothers that robustly handle deformations and anisotropies in the underlying mesh. Since the data and control flow for e.g. an ADI-TRIGS smoother is non-trivial to implement on a GPU, this adds an additional level of complexity to the already hard scheduling problem: The coarse grid macros need to be partitioned into parallel blocks based on the availability and applicability of smoothers and the (expected) run-time on each node. We did not address this hard dynamic scheduling problem yet, and confine ourselves to static partitions of exemplary test domains in this paper.
The last bottleneck is that the GPU is idle when the CPU performs the accumulation and update steps, and the computation is stalled during transfers from and to the co-processor. We addressed this issue by implementing a streaming solution scheme for all sub-problems (macros) that are scheduled onto the same node: While sub-problem i is smoothed on the GPU, the data for sub-problem $i+1$ is transferred to the GPU memory using DMA, and the result vector of problem $i-1$ is read back to the host and accumulated to the local solution. This streaming technique of interleaved transfers and computation also reduces the impact of the transfer over the comparatively slow PCIe bus.

4 Preliminary results

In our tests we solve the Poisson problem $-\Delta\mathbf{u} = \mathbf{f}$ with zero Dirichlet boundary conditions on two different grids. All tests are executed on a cluster with 32 compute nodes and one master node. Each node comprises a dual Intel EM64T 3.4 GHz and a NVIDIA Quadro FX1400 PCIe (mid-range) graphics card. The nodes are fully connected via Infiniband.
The first mesh is a simple cartesian grid which allows to use fast simple solvers. The second is a mixture of regular and generalised tensor product macros which require a more advanced solver for convergence. 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.

4.1 Regular grids

On square-like grids a multigrid cycle with a Jacobi smoother converges very quickly, and a corresponding inner solver is available on both on the CPU and GPU. Thus, we can

schedule a task to the CPU or GPU depending solely on the problem size on the current outer multigrid level. This is done statically for the macros, and again dynamically based on the CPU-GPU switch described above. We evaluate six different configurations:

1x16p CPU: 16 nodes, one CPU each, one CPU process

1x16p GPU: 16 nodes, one GPU each, one GPU process

2x16p CPU: 16 nodes, two CPUs each, two CPU process

2x16p GPU: 16 nodes, two CPUs and one GPU each, one CPU and one GPU process

1x32p CPU: 32 nodes, one CPU each, one CPU process

1x32p GPU: 32 nodes, one CPU and one GPU each, one GPU process

Currently, we do not have nodes in which we can use two graphics cards in parallel, but this will be possible in the future. The grid we use comprises 16x16 macros. All macros have the same size $(2^L + 1)^2$, where L is the *level* of the macro. By varying the level we obtain overall problem sizes from ≈1M to ≈67M grid nodes.
In figure 1 (left) we compare the performance of the cluster with one or two CPUs per node and with or without a graphics card. The first diagram shows the absolute execution time, the second the time normalised by the number of unknowns per macro. Comparing the 1x16p CPU against the 2x16p CPU configuration, we see that the second CPU process per node gains approximately 20% in performance. This number is fairly low because our computations are very data intensive and the CPUs share a common bus to the main memory.
Comparing the combined CPU-GPU solvers against the CPU solver, we see at first a surprising up and down movement, which is particularly clear in the normalised graph. There are two causes for this behaviour. For small levels almost all work is scheduled to the CPU (cf. section 3) because the GPU is only efficient on large macros. Therefore, the first levels basically compare the C++ against the (much more optimised) Fortran multigrid solver. The former wins at first due to the lower bandwidth requirements (single precision). On the highest level we finally see the acceleration by the GPU, which performs very well on large problems. This advantage is large enough for the 1x16p GPU version to outperform the 2x16p CPU version. From an economic point of view this means that equipping the cluster nodes with one processor plus a mid-range graphics card can already outperform a dual processor node for this kind of application. Clearly, this is not a general statement, but we expect the advantage to grow for higher levels, which we could not verify on time due to scheduling problems with the Linux kernel and unoptimised MPI buffered transfers over the Infiniband interconnects.
Figure 1 (centre) compares a 16 node two processor against a 32 node one processor configuration. Again we show the absolute numbers on top and the normalised times at the bottom. We see a similar behaviour as in the previous graph. The significantly better performance of the 1x32 configurations shows once again the importance of high bandwidth in the system. Both configuration types 2x16 and 1x32 utilise the same number of CPUs, but the latter has more bandwidth available because the CPUs are not sharing a bus. This advantage clearly outweighs the additional communication necessary. Because the entire

bandwidth is now available to the CPU, the 1x32 GPU version is fastest on the highest level by a smaller margin than before.

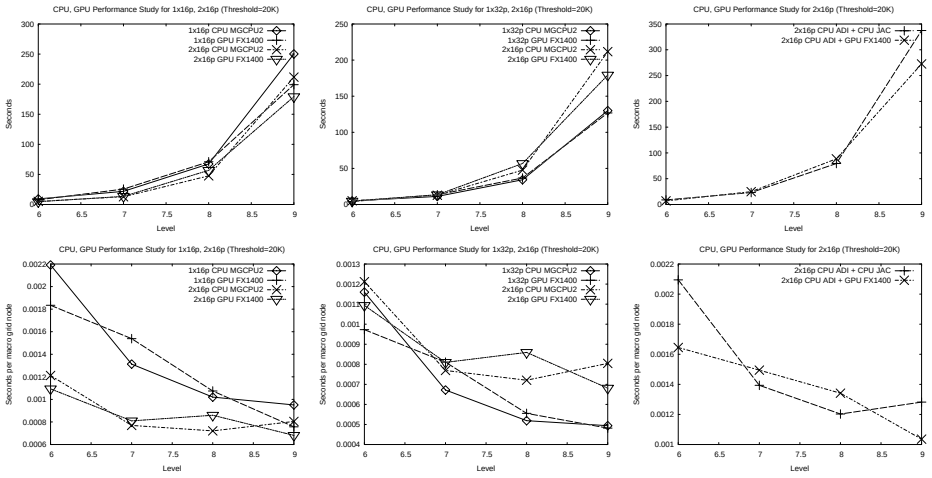

Figure 1: Absolute (top) and normalised (bottom) cluster performance: Left: 16 nodes, regular grid, one or two CPUs and zero or one GPU per node. Centre: 16 or 32 nodes, regular grid, one or two CPUs and zero or one GPU per node. Right: 16 nodes, coarsely adapted grid, one or two CPUs and zero or one GPU per node.

4.2 Coarsely adaptive grids

Here we test the situation for a grid containing generalised tensor-product macros which require a more advanced solver, currently not available on the GPU. The idea is to schedule the anisotropic macros to an advanced CPU solver and utilise the GPU for the quick solution of several isotropic macros in the meantime. The GPU smoother continues to use the size dependent switch to divert small problems to the CPU. The timings are again obtained with a static partitioning of the domain. Figure 1 (right) shows the absolute and normalised execution time for the Poisson problem on this domain. We use two configurations:

> **2x16p CPU ADI + CPU JAC:** 16 nodes, two CPUs and two CPU processes each. The first process handles the anisotropic macros with an ADI-TRIGS multigrid, the second several of the simple ones with a Jacobi multigrid solver.
>
> **2x16p CPU ADI + GPU:** 16 nodes, two CPUs and one GPU each, one CPU and one GPU process. The CPU handles the anisotropic macros with the ADI-TRIGS multigrid, the GPU several of the simple ones with a Jacobi multigrid solver.

The results are consistent with the previous graphs. Once the problem size becomes large, the GPU can process the data in parallel quicker than the second CPU in the 2x16p CPU ADI+CPU JAC configuration. An additional advantage of the GPU processing is that we

put less strain on the main memory system, whereas in the CPU-CPU configuration both processors compete for bandwidth. Nevertheless, there is still potential to further accelerate the 2x16p CPU ADI+GPU configuration, since tests on a single node show that the GPU can perform up to 5 times as much work as the CPU [GST06].

5 Summary and future work

We have demonstrated a prototypical approach to leverage the bandwidth and compute power of GPUs in the parallel Finite Element package FEAST. Our integration is minimally invasive in the sense that apart from some control code, the original FE package remains untouched, and the GPU is included as a new type of smoother. The adapted iterative refinement method solves one key disadvantage of GPUs, namely the restriction to single precision floating point computation: In our results, we gain the same accuracy as if the entire computation was performed in double precision.
Our results are preliminary in two ways: First, we did not have the time to include computations on level 10, on which we expect the speedup of the GPUs to be most significant. Second, we did not address the hard dynamic scheduling problem yet. Despite these two drawbacks, our results demonstrate the potential of enhancing commodity clusters with GPUs, not only for pure speedup, but also in view of the total cost of ownership: Including GPUs is a much more cost-efficient enhancement than adding more compute nodes to increase the performance of an existing cluster.

Acknowledgments

This work is – in parts – a collaboration with Robert Strzodka and Patrick McCormick. We would like to thank the FEAST developers, Sven Buijssen, Matthias Grajewski, Susanne Kilian and Hilmar Wobker, for their support. This work has been in part supported by the German Science Foundation (DFG), project TU102/22-1.

References

[BKT03] Ch. Becker, S. Kilian, and S. Turek. Hardware–oriented numerics and concepts for PDE software. In *FUTURE 1095*, pages 1–23. Elsevier, 2003. International Conference on Computational Science ICCS2002, Amsterdam.

[GST06] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 2006. to appear.

[OLG+05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics, State of the Art Reports*, pages 21–51, September 2005.