In *Proceedings of ASIM 2005 - 18th Symposium on Simulation Technique*, Sept. 2005.

Accelerating Double Precision FEM Simulations with GPUs

Dominik Göddeke
dominik.goeddeke@math.uni-dortmund.de
Universität Dortmund, Fachbereich Mathematik
Vogelpothsweg 87, 44 227 Dortmund

Robert Strzodka
strzodka@caesar.de
caesar research center
PO Box 120 260, 53 044 Bonn

Stefan Turek
ture@featflow.de
Universität Dortmund, Fachbereich Mathematik
Vogelpothsweg 87, 44 227 Dortmund

Abstract

In visualization and computer graphics it has been shown that the numerical solution of PDE problems can be obtained much faster on graphics processors (GPUs) than on CPUs. However, GPUs are restricted to single precision floating point arithmetics which is insufficient for most technical scientific computations. Since we do not expect double precision support natively in graphics hardware in the medium-term, we demonstrate how to accelerate double precision iterative solvers for Finite Element simulations with current GPUs by applying a mixed precision defect correction approach. Our prototypical algorithm already runs more than two times faster than a highly tuned pure CPU solver while maintaining the same accuracy. We present a series of tests and discuss multiple optimization options.

1 Introduction

Over the last two years, programmable commodity graphics processors (GPUs) have gained a lot of interest outside the field of computer graphics because of their astonishing memory bandwith and tremendous floating point computational power. A new field of research commonly called *general purpose computation on graphics hardware* (GPGPU) has emerged. We refer the reader to [3] for a survey of this research field with extended references, and to [2] for online material with introductions, example codes and further information.

The number of different applications presented in the survey shows the attractiveness of the GPU as a numerical co-processor. Especially in the computationally intensive field of

partial differential equations (PDEs) various problem types such as the diffusion equation, the wave equation, the Poisson problem or the Navier-Stokes equations have been successfully implemented [3]. GPUs are particularly suitable for these computations as Finite Difference or (structured) Finite Element Method (FEM) schemes map naturally to the data-stream based paradigm that GPUs employ. However, in real-world problems which require a highly accurate solution, GPUs have not been used so far due to the restricted computational precision. Current GPUs support quasi-IEEE 754 conformal *half* (s10e5) and *single* (s23e8) precision floating point formats, and we do not expect native hardware support for *double* (s52e11) precision arithmetics in the medium-term. Double arithmetics can be emulated with single floats, also on GPUs[1]. But such emulations increase more than tenfold the operation count. This is only acceptable in otherwise bandwidth bound operations.

In this paper we, therefore, propose the revitalization of mixed precision defect correction approaches that have been known for almost 100 years: By iteratively computing residuals of a single precision approximate solution to a linear system in double precision, only few correction steps suffice to reduce approximation errors close to machine accuracy. In the context of scientific computing using GPUs, this approach translates to correcting a GPU result with just a few CPU-based iterations. In this way we obtain the *full accuracy* of CPUs with the *high speed* of GPUs. In particular, GPUs can now be used to accelerate technical applications which require the computational results to be sufficiently accurate in terms of quantities like e.g. drag and lift values. Obviously, this task goes far beyond the real-time visualization of qualitatively correct solution profiles.

To estimate possible performance gains we have first benchmarked FEM building blocks such as vector-vector operations and banded matrix-vector multiplication on the CPU. Due to inherently cache-unfriendly memory access patterns, most Finite Element codes exhibit a massive decline in (computational) efficiency [4] once the number of unknowns exceeds the cache capacity of common CPUs. Next we have studied and performed our own GPU benchmarks[2]. The results show that GPU performance is asymptotically converse: For small problems, CPUs outperform GPUs by far, but for the more interesting large problem sizes, GPUs are able to deliver several GFLOP/s of sustained performance. For the tested FEM building blocks the GPU outperforms highly optimized cache-aware CPU implementations by a factor of 5 to 20. Thus the GPU is an ideal co-processor in a mixed precision defect correction method applied to FEM simulations.

This technique of CPU–GPU–coworking maps harmoneously into our FEAST [5] framework. By applying domain decomposition, the global solution of a huge problem is split into a sequence of local subproblems (each with up to 10^6 unknowns) which we plan to solve on the GPU. By using generalized tensorproduct meshes' in contrast to regular cartesian grids (each inner node has to be incident to exactly 4 cells, but the cells can be arbitrarily shaped), we can resolve complex geometries and deform the mesh to concentrate cells based on a-posteriori error estimation [1] while still being able to apply highly optimized fast data structures for regular meshes and use the GPU.

[1] Emulated double precision in the Gaia project at the Lawrence Livermore National Laboratory: `www.ll.mit.edu/HPEC/agendas/proc04/powerpoints/Talks-OPEN/Tues/johnson.ppt`

[2] Workshop: *GPUs as FEM co-processors*, `www.mathematik.uni-dortmund.de/~goeddeke/`

The remainder of this paper is organized as follows. Section 2 describes and analyzes our approach; Section 3 provides a thorough evaluation w.r.t. robustness, accuracy and applicability. We finish with conclusions and future work in Section 4.

2 CPU–GPU–Solver

We present a mixed precision defect correction algorithm for the iterative solution of linear equation systems. The core idea of the algorithm is to split the solution process into a computationally intensive but less precise inner iteration running in 32 bit on the GPU and a computationally simple but precise outer correction loop running in 64 bit on the CPU. Our approach can be easily implemented on top of an existing GPU-based single precision iterative solver in applications where higher precision is necessary. The algorithm requires two input parameters, ϵ_{inner} and ϵ_{outer} as stopping criteria for the inner and outer solver respectively. Let A denote the (sparse) coefficient matrix, $\mathbf{b}$ the right hand side, $\mathbf{x}$ the initial guess for the solution and a scaling factor α. Subscript $_{32}$ indicates single precision vectors stored in GPU memory and $_{64}$ indicates double precision vectors stored in CPU memory.

1. Set initial values: $\alpha_{64} = 1.0$, $\mathbf{x}_{32} = \mathbf{x}_{64} = \mathbf{0}$.
2. Iterate inner solver until $||\mathbf{b}_{32} - A_{32}\mathbf{x}_{32}|| < \epsilon_{inner}$.
3. Transfer inner solution to CPU and update outer solution: $\mathbf{x}_{64} \leftarrow \mathbf{x}_{64} + \alpha_{64}\mathbf{x}_{32}$.
4. Calculate defect in double precision: $\mathbf{d}_{64} = \mathbf{b}_{64} - A_{64}\mathbf{x}_{64}$.
5. Calculate norm of defect: $\alpha_{64} = ||\mathbf{d}_{64}||$
6. Check for convergence ($\alpha_{64} < \epsilon_{outer}$); otherwise scale defect: $\mathbf{d}_{64} = \alpha_{64}^{-1}\mathbf{d}_{64}$.
7. Set new initial guess and RHS: $\mathbf{x}_{32} = \mathbf{0}$, $\mathbf{b}_{32} \leftarrow \mathbf{d}_{64}$, goto step 2

Note that only a single matrix-vector multiplication and norm calculation is required on the CPU in each iteration. This is the basic form of the algorithm. We are currently testing different variants and extensions and will report on them in the future. For example, a simple but effective performance increase can be obtained by not checking for convergence of the inner solver after every step, but checking heuristically at regular intervals depending on the expected convergence behaviour of the inner solver, e.g. a full-scale multigrid is converging much better than a conjugate gradient approach and thus requires less inner iterations and more frequent convergence checks. Our experiments (cf. Section 3.3) indicate that this can be controlled effectively by adjusting the inner stopping criterion ϵ_{inner}. Additionally, asynchroneous transfers for convergence checks are worth further examination. Instead of flushing the GPU pipeline for a read-back to the CPU of the norm of the defect, we may start the (potentially superfluous) next iteration on the GPU and wait for an asynchronous read-back to finish while the GPU is computing. Finally, we can exploit the fact that the GPU is idle when the defect correction is performed on the CPU.

Although here we are concerned with a 32/64 bit combination to obtain double precision results, it is worth mentioning that this defect correction method can also be used as a 16/32

bit combination on the GPU. Half precision results can be updated to single precision, while taking advantage of the 50% bandwidth reduction and the implied performance increase.

3 Evaluation and Results

3.1 Test description

For a given test function, we solve the Poisson equation $-\Delta \mathbf{u} = \mathbf{f}$ with Dirichlet boundary conditions on $\Omega = [0,1]^2$. Conforming bilinear Finite Elements (Q_1) are used for the spatial discretization for different levels of refinement ($N = 3^2$ to $N = 1025^2$ unknowns). Physically, this can be interpreted as calculating the deflection of a membrane fixed at its boundaries and stressed with the force distribution given by the right hand side. For simplicity, we use an unpreconditioned conjugate gradient[3] solver. The error analysis below is performed with the test function $\mathbf{u}_0(x,y) := x(1-x)y(1-y)$. The error shown in Tables 1 and 2 is measured nodewise with the scaled l_2 norm (approximate L_2) of the analytic solution and our computed results. Reference data on the CPU is obtained using the cache-aware highly optimized FEAST simulation package [5] on an Opteron 250 Linux node (4 GFLOP/s LinPack). GPU timings were taken on a GeForce 6800 graphics card (AGP).

3.2 Influence of input data precision

In the first test series, we analyze the influence of the input data precision on the overall error. The right hand side is therefore computed as the discrete Laplacian $\mathbf{f} := -\Delta_h \mathbf{u}_0$ to avoid discretization errors and then used in double, single or half[4] float precision. The computation itself is performed in the CPU reference solver and the GPU-based defect correction solver with full accuracy.

$\mathbf{f}_{16}$ (CPU)	$\mathbf{f}_{16}$ (GPU-CPU)	$\mathbf{f}_{32}$ (CPU)	$\mathbf{f}_{32}$ (GPU-CPU)	$\mathbf{f}_{64}$ (CPU)	$\mathbf{f}_{64}$ (GPU-CPU)
$2.333 \cdot 10^{-6}$	$2.333 \cdot 10^{-6}$	$7.718 \cdot 10^{-10}$	$7.717 \cdot 10^{-10}$	$2.750 \cdot 10^{-13}$	$2.806 \cdot 10^{-13}$
$1.008 \cdot 10^{-6}$	$1.008 \cdot 10^{-6}$	$7.726 \cdot 10^{-10}$	$7.725 \cdot 10^{-10}$	$1.051 \cdot 10^{-12}$	$1.049 \cdot 10^{-12}$

Table 1: Error caused by varying input data precision ($\mathbf{f}_{16}$, $\mathbf{f}_{32}$ and $\mathbf{f}_{64}$) for a pure CPU solver and our combined GPU-CPU approach, both using double precision to solve until the norm of the residual drops below 10^{-12}. Problem size $N = 257^2$ and $N = 513^2$.

We observe that representing all involved vectors in double precision is essential. Although the computation is always performed in 64 bit, the reduction of the right hand side to 32 bit already costs approximately 3 digits of accuracy. The furthergoing reduction to 16 bit costs additional 3 digits. This shows that it does not make sense to improve the internal computational precision of the graphics pipeline without introducing a higher precision format for input and output. Even if the kernel programs on the GPU could compute in double precision this would be of little use if the input and output formats remained single float.

[3] The (unoptimized) implementation is based on the 'outdated' OpenGL pBuffer technique which implies additional performance penalties. We expect better performance with the new 'framebuffer object' technique.

[4] A CPU implementation of the `half` data type is available as part of the OpenEXR library.

Our defect correction approach, however, is very suitable for utilizing the reduced precision of GPUs in a way which does not harm the final result. We see that the GPU-CPU approach is hardly any worse than the pure computation on the CPU. Consequently, current GPUs may not be used for direct high precision computations, but are good at improving a double precision result via defect corrections.

3.3 Overall error and performance results

In this test we analyse the overall error of the continuous problem and the performance of the different approaches. The right hand side $\mathbf{f} := -\Delta \mathbf{u}_0$ is computed with the continuous Laplacian to obtain the continous problem $-\Delta \mathbf{u} = \mathbf{f}$ in Ω with the analytically known solution $\mathbf{u} = \mathbf{u}_0$. We run the pure CPU and the GPU-CPU solver at half, single and double precision, with the input data in the corresponding precision and solve until $\epsilon_{outer} < 10^{-12}$. Table 2 summarizes the computed differences to the analytical reference solution for different refinement levels (number of unknowns).

N	CPU16	GPU-CPU16	CPU32	GPU-CPU32	CPU64	GPU-CPU64
3^2	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$	$5.208 \cdot 10^{-3}$
5^2	$1.444 \cdot 10^{-3}$	$1.509 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$	$1.440 \cdot 10^{-3}$
9^2	$2.675 \cdot 10^{-4}$	$7.556 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$	$3.869 \cdot 10^{-4}$
17^2	$4.047 \cdot 10^{-4}$	$2.332 \cdot 10^{-3}$	$1.015 \cdot 10^{-4}$	$1.016 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$	$1.015 \cdot 10^{-4}$
33^2	$2.120 \cdot 10^{-3}$	$2.253 \cdot 10^{-3}$	$2.611 \cdot 10^{-5}$	$2.648 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$	$2.607 \cdot 10^{-5}$
65^2	noise	noise	$6.464 \cdot 10^{-6}$	$8.324 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$	$6.612 \cdot 10^{-6}$
129^2	noise	noise	$1.656 \cdot 10^{-6}$	$8.554 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$	$1.666 \cdot 10^{-6}$
257^2	noise	noise	$5.927 \cdot 10^{-7}$	$2.781 \cdot 10^{-5}$	$4.181 \cdot 10^{-7}$	$4.181 \cdot 10^{-7}$
513^2	noise	noise	$2.803 \cdot 10^{-5}$	$1.119 \cdot 10^{-4}$	$1.047 \cdot 10^{-7}$	$1.047 \cdot 10^{-7}$
1025^2	noise	noise	$7.708 \cdot 10^{-5}$	$4.463 \cdot 10^{-4}$	$2.620 \cdot 10^{-8}$	$2.620 \cdot 10^{-8}$

Table 2: Errors for different solver accuracies both on the CPU and on the GPU.

First let us note that in a given visualization of the results, there is no discernible difference for $N \geq 17^2$, whereas the quantitative differences are obvious from Table 2. The impact of the solver accuracy is clearly visible: Despite the simplicity of the test function, half and single precision are insufficient to approximate it using a Finite Element discretization. For the first few levels, the discretization error dominates and decreases by the expected factor of 4 in each refinement (increase in problem size N). From level 3 and 6 on (half and single precision respectively), the lack of precision dominates the error and even worse, further refinement of the computational mesh increases the error due to worse matrix conditioning. Additionally, observe the difference in accuracy in half and single precision between the CPU and the GPU. On the other hand, the defect correction algorithm yields the same accuracy as the reference solver running completely on the CPU in double precision, independent of the inner threshold.

For the performance comparison, we only take $N \geq 257^2$ into account, since the CPU with its large cache naturally outperforms the GPU for smaller problems. Table 3 summarizes runtimes and iteration counts depending on the inner stopping criterion. It shows the overall GPU timings including the necessary data transfer and the CPU corrections.

N	Iterations CPU	Time CPU	Iterations GPU	Time GPU	Iterations GPU	Time GPU
257^2	303	2.32s	926 (6)	5.53s	1151 (4)	6.68s
513^2	601	19.47s	1885 (6)	8.34s	2744 (5)	11.59s
1025^2	1191	146.90s	4378 (7)	60.16s	6385 (6)	86.84s

Table 3: Performance results for the double precision solver: CPU reference solver, GPU-CPU solver with $\epsilon_{inner} = 10^{-2}$, GPU-CPU solver with $\epsilon_{inner} = 10^{-8}$. Results are summed over all inner iterations, the number of outer iterations is listed in brackets.

We see that the combined GPU-CPU solver with a high inner stopping criterion (10^{-2}) is fastest. This requires more outer iterations on the CPU, but they are relatively cheap. The 7 defect correction steps for the largest problem take 2.41sec CPU computing time and 0.26sec transfer time. So in addition to the optimizations discussed in Section 2, the choice of ϵ_{inner} and the inner solver will have to be examined closely for an optimal GPU-CPU interplay. But even without these future optimizations the current solution is already 2.3 times faster than the pure CPU solver, if the vector size exceeds the CPU cache size.

4 Conclusions

We have presented a preliminary approach to accelerate double precision Finite Element simulations using the GPU as a co-processor. The main virtue of the defect correction algorithm is the equally good accuracy as compared to a double precision CPU solver. The second goal of significantly improved performance has also been achieved. We have outlined that without impairing the accuracy further performance gains may be expected through a better controlling of the algorithm's parameters. We will follow these lines to provide an efficient embedding of the GPU as a numerical co-processor for established double precision PDE packages on parallel computers.

References

[1] *Grajewski, M., Köster, M., Turek, S.:* Numerical Analysis and Practical Aspects of a Robust and Efficient Grid Deformation Method in the Finite Element Context. To appear.

[2] *Harris, M.J.:* General Purpose Computation on GPUs, `http://www.gpgpu.org`

[3] *Owens, J.D., Luebke, D., Govindaraju, N., Harris, M.J., Krüger, J.,Lefohn, A.E., Purcell, T.J.:* A Survey of General-Purpose Computation on Graphics Hardware. Eurographics 2005, State of the Art Reports (2005), pp. 21-51.

[4] *Turek, S., Becker, Ch., Killian, S.:* Consequences of modern hardware design for numerical simulations and their realization in FEAST. Proceedings of Euro-Par '99 (1999), pp. 643-650.

[5] *Turek, S., Becker, Ch., Killian, S.:* Hardware-oriented Numerics and concepts for PDE software. Special Journal Issue for PDE Software FUTURE 1095 (2003), pp. 1-23.