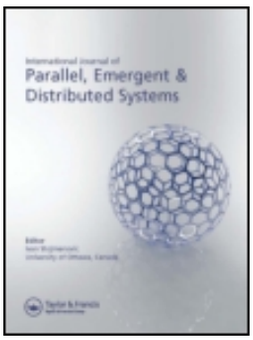

International Journal of Parallel, Emergent and Distributed Systems

ISSN: 1744-5760 (Print) 1744-5779 (Online) Journal homepage: https://www.tandfonline.com/loi/gpaa20

Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations

Dominik Göddeke , Robert Strzodka & Stefan Turek

To cite this article: Dominik Göddeke , Robert Strzodka & Stefan Turek (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, DOI: 10.1080/17445760601122076

To link to this article: https://doi.org/10.1080/17445760601122076

The International Journal of Parallel, Emergent and Distributed Systems,
Vol. 22, No. 4, August 2007, 221–256

Taylor & Francis
Taylor & Francis Group

Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations

DOMINIK GÖDDEKE†*, ROBERT STRZODKA‡ and STEFAN TUREK†

†Universität Dortmund, Fachbereich Mathematik, Vogelpothsweg 87, 44 227 Dortmund, Germany
‡Stanford University, Max Planck Center, 353 Serra Street, Stanford, CA 94305, USA

(Received December 2006; revised August 2006; in final form October 2006)

In this survey paper, we compare native double precision solvers with emulated- and mixed-precision solvers of linear systems of equations as they typically arise in finite element discretisations. The emulation utilises two single float numbers to achieve higher precision, while the mixed precision iterative refinement computes residuals and updates the solution vector in double precision but solves the residual systems in single precision. Both techniques have been known since the 1960s, but little attention has been devoted to their performance aspects. Motivated by changing paradigms in processor technology and the emergence of highly-parallel devices with outstanding single float performance, we adapt the emulation and mixed precision techniques to coupled hardware configurations, where the parallel devices serve as scientific co-processors. The performance advantages are examined with respect to speedups over a native double precision implementation (time aspect) and reduced area requirements for a chip (space aspect).

The paper begins with an overview of the theoretical background, algorithmic approaches and suitable hardware architectures. We then employ several conjugate gradient (CG) and multigrid solvers and study their behaviour for different parameter settings of the iterative refinement technique. Concrete speedup factors are evaluated on the coupled hardware configuration of a general-purpose CPU and a graphics processor. The dual performance aspect of potential area savings is assessed on a field programmable gate array (FPGA). In the last part, we test the applicability of the proposed mixed precision schemes with ill-conditioned matrices. We conclude that the mixed precision approach works very well with the parallel co-processors gaining speedup factors of four to five, and area savings of three to four, while maintaining the same accuracy as a reference solver executing everything in double precision.

Keywords: Mixed precision iterative refinement; Emulated precision; Graphics hardware; Reconfigurable hardware; Large sparse linear equation systems; FEM

1. Introduction

After a long period of steady growth, desktop computer architecture has reached a turning point. Further progress is no longer enabled by growth in core clock rates, but by growth in parallelism. Physical limitations, in particular thermal restrictions, generate a trend towards more *parallelism on a chip*: Double-core processors are already on sale and multi-core processors with much more parallelism are in the making. The parallel arrangements distribute the heat more evenly, avoiding dangerous hot spots.

*Corresponding author. Email: dominik.goeddeke@math.uni-dortmund.de

The International Journal of Parallel, Emergent and Distributed Systems
ISSN 1744-5760 print/ISSN 1744-5779 online © 2007 Taylor & Francis
http://www.tandf.co.uk/journals
DOI: 10.1080/17445760601122076

In the attempt to uphold a simple sequential programming model, the CPU has only recently opted for the parallelisation instead of enhancement of its inner structures. Other more application specific chips have followed this path from the beginning. For instance, Field Programmable Gate Arrays (FPGAs) and Graphics Processor Units (GPUs) are made up of a large number of Boolean or arithmetic processing elements (PEs). In these highly-parallel architectures the PEs consume a high percentage of the overall transistor count, in contrast to the cache-dominated general purpose CPUs. If we use the PEs for floating point computations then the required precision has a strong impact on the size and costs of the chip. Hardwired PEs in most highly-parallel architectures are therefore only single precision (e.g. in the GPU), while reconfigurable architectures (e.g. FPGA) can be configured for double precision operations, but these arithmetic units consume so many resources that efficient parallelisation is only possible with low precision units.

For memory dominated processors the change from low precision to high precision PEs has only a small impact on the overall number of transistors and thus CPUs can more easily afford to compute in high precision in the Floating Point Unit (FPU) than other more parallel devices. However, higher precision FPUs not only increase the transistor count but also require wider data paths through the chip and make it more difficult to meet timing constraints. So even the CPU would waste too many resources by concentrating on double precision alone, and the SSE units in CPUs include a dedicated-optimisation for the single float format, enabling the processing of twice as many operations on single than on double floats.

We see that current hardware architectures offer a lot of computation power in single float precision but scientific applications do not take advantage of these capabilities, as the accuracy requirements apparently force them to use double precision operations exclusively. In fact, for many problems the restriction of the *computational precision* to single float would mean an unacceptable loss of the *result accuracy*, see for example, the PDE case in Section 5.3. The many parallel single precision PEs can be exploited by emulating high precision operations and for a completely general algorithm this is the only possibility to gain more accuracy with low precision PEs. However, for many algorithms we do not require high precision arithmetic for all intermediate computations to gain highly-accurate final results. The knowledge about the error propagation in the algorithm can be used to confine the use of high precision computations to only few relevant places. We obtain a *mixed precision method* which utilises low and high precision computations in different parts of the algorithms.

Given these three approaches of native high precision, emulated high precision and mixed precision implementations, we compare the resulting performance and accuracy in this paper. The main focus lies on the mixed precision technique for the solution of PDEs. The performance aspect is studied in view of the existing parallel co-processors that execute single precision operations extremely fast. The accuracy is examined with respect to wide applicability of these approaches for problems of high condition.

1.1. Mixed precision

The idea behind mixed precision methods is obviously to perform a *large* part of the computation in low and only a small part in high precision, thus allowing the hardware to save on transistors by offering many low precision and only few high precision PEs. The low and high precision PEs may reside in different chips, e.g. a general purpose CPU with double float arithmetic and a parallel co-processor with single float arithmetic. Section 4 discusses available architectures that are particularly suitable for this kind of processing.

This simple idea of utilising different precision formats in the same algorithm has surprisingly many beneficial properties:

Accuracy

- For many algorithms the mixed precision method can obtain exactly the same final accuracy as though the entire computation were performed in the high precision format.
- For certain algorithms, like the solution of a linear equation system, the above savings are particularly large, because even 99% of the operations can be performed in the low precision format, without affecting the final accuracy.
- Since we use less precision in the computation it is always possible to construct problems for which the mixed precision approach must break down, however, we show that even for very badly conditioned practical cases of PDE problems there is no loss in accuracy.

Computation

- The number of transistors required to implement a hardware multiplier grows quadratically with the size of the operands (bit length). Consequently, instead of one high precision multiplier we can use the same number of transistors for four low precision multipliers if the low precision format is half the size of the high precision format.
- As a consequence of the above, the number of operations required in the emulation of a high precision format with low precision arithmetic grows quadratically with the quotient of the high and low precision number format sizes. For floating point operations even more emulation operations are required, because of the special treatment of the exponent.
- Mixed precision techniques are used extensively in reconfigurable architectures where low precision PEs directly translate into low resource consumption and thus more parallelism. In this case, PEs of different precision can be configured on the same chip.
- Hardwired processors gain even more from mixed precision algorithms, because hardwired PEs are far more transistor efficient than reconfigurable ones. Thus, we may utilise a highly parallel low precision co-processor for the majority of the computations and upgrade the accuracy with few high precision computations on a slower general purpose processor.
- The decision to produce a highly-parallel architecture of limited precision does not limit the accuracy of the final results. In combination with a general purpose processor which supports various number formats and can emulate even more, the same parallel architecture can be used to obtain double, extended or even variable accuracy results.

Memory

- The use of low precision formats requires fewer resources not only for the PEs but also for the memory. This is particularly important for the very expensive and limited memory that resides next to the PEs (register file, L1 cache) and the second level memory on the same die (embedded memory, L2 cache). For optimal processing, large data sets are often split into blocks that match the size of these local memories. Smaller data formats allow the division into larger blocks which increases the efficiency of the entire algorithm.
- The use of low precision formats reduces the bandwidth requirement in computations. This is much in favour of mixed precision methods as memory transfers are often the bottleneck in computations and also account for more power consumption than the computation itself.

In summary, we can say that mixed precision methods have wide applicability and benefit both computation and bandwidth limited algorithms. For bandwidth limited algorithms with low arithmetic intensity (ratio of operations per memory access), we measure almost the theoretical factor of two in speedup when transitioning from double to single float precision. More generally, in the exponential development of computer technology in the last 10 years we observe that memory performance is increasing much slower than computational performance, because the integration of additional PEs into a processor is much cheaper both in costs and power than the widening of the data bus for additional bandwidth. This leads to the so-called *memory wall* [1], the inability to provide enough bandwidth for the PEs, thus limiting the overall performance of the system by the memory performance rather than the peak computational performance. Using low precision number formats in most intermediate computations of an algorithm cuts the bandwidth requirements almost in half and thus alleviates the memory wall problem.

If a particular algorithm is computation limited despite the general memory wall problem, then the savings obtained from mixed precision methods can be even larger than in the memory limited case, since the area of a multiplier grows quadratically with the bit length of the operands. Thus, cutting the number format size in half allows up to four times as many PEs on the same area. The factor is not exactly four, as there is some overhead associated with each unit and for floating point numbers the ratio of the exponent to the mantissa has additional impact. Also, in hardwired architectures this quadratic advantage cannot be exploited so easily since the data paths are fixed. Therefore, hardwired *dual-mode* FPUs, like the SSE unit in current CPUs, have only a linear and not quadratic efficiency, offering twice and not four times as many operations in single than double precision. In reconfigurable architectures where we have control over the data paths, we do indeed benefit directly from the quadratic savings and can configure four single instead of one double multiplier; see Section 7.

To avoid the inefficiency of dual-mode FPUs we could think of producing chips with four times as many single float units and emulate double precision when needed. However, once a chip has been fabricated with single precision FPUs, emulation of double precision costs much more than the factor of four, because we are confined to a fixed instruction set and cannot use the transistors in the most efficient way for the emulation, see Section 2.2. Thus, we are in a dilemma: a dual-mode FPU offers only two instead of four single precision operations. However, if the same transistors are invested in four single precision FPUs, we double the single precision performance, but also half or even quarter double precision performance due to the inefficient emulations. To resolve this problem the chips simply concentrate on the precision that is most often used in their application domains and take into account the fact that operations in the other precision will be fairly inefficient in comparison to the other architecture. With mixed precision methods we actually take advantage of this specialisation for a certain precision and divide the algorithm into two parts that execute best on one or the other architecture, see Section 3.

In reconfigurable hardware the tradeoff between the precision and area of a multiplier is truly quadratic, but here we pay a price *a priori*, as the area of a configured FPU is approximately six times larger than that of a hardwired FPU. This is the reason why newer FPGAs contain hardwired integer multipliers in high numbers. They can be efficiently used for the treatment of the mantissa of floating point numbers. Integer multipliers have the advantage of allowing efficient emulations of higher precision integer operations, see Section 2.2. But even with embedded hardwired multipliers there still remains a configuration overhead for the data paths. The optimal balance between hardwired and reconfigurable elements, in particular for floating point operations, is discussed in detail by Ho *et al.* [2].

1.2. Paper organisation

The paper is roughly divided in four parts. After this introductory section, we discuss the theoretical background of high precision floating point emulation and the mixed precision iterative refinement technique in Sections 2 and 3. Section 3.2 describes our general template of the iterative refinement algorithm for the solution of large, sparse linear equation systems. The next section is devoted to hardware configurations particularly suitable for the discussed algorithms. The third part of the paper contains our main numerical and benchmarking results. After a brief introduction to the test procedure that we apply in the finite element context, we present tests to analyse the performance and accuracy of the iterative refinement solvers in Section 5. Section 6 then reports on achievable speedups with both the emulation and the refinement techniques using a GPU as a co-processor to the general purpose CPU. We analyse the potential resource savings on a FPGA in Section 7. The last section of the paper returns to the numerical examination of the iterative refinement schemes with respect to increasingly ill-conditioned matrices. There, we employ operator and mesh anisotropies as a practical approach to examine such influences in the FEM context. We conclude with a concise summary, plans and suggestions for future work. The appendix contains tables with the actual performance and accuracy numbers, while in the main text we use mostly diagrams for better readability.

Given the combination of diverse research areas ranging from numerical analysis to hardware systems we do not present related work in one place, but rather discuss related approaches in the individual sections. This allows us to discuss the techniques, their background and application to our algorithms in one context.

2. High precision emulation

2.1. Exact and redundant emulation

For unsigned integers the emulation of higher precision arithmetic with lower precision units is straightforward. An addition of two m-bit operands generates a $m + 1$-bit, and a multiplication a $2m$-bit result. Integer units typically provide a carry bit that stores the additional bit of the sum, and subsequent assembly commands can include this carry bit in the addition, such that an arbitrary long chain of m-bit words can be processed to emulate a $k{\cdot}m$-bit format. In case of the multiplication, the hardware either provides the entire $2m$-bit result of a $m \times m$-bit multiplication and thus the product of two $k{\cdot}m$-bit operands results from the addition of $k(k + 1)/2$ mixed $m \times m$-bit products; or the multiplier delivers only a m-bit result, then each m-bit word must be split into two $m/2$-bit words and we have four times as many mixed products to process.

The emulation of higher precision floating point numbers follows the same lines, but there is an additional difficulty as the bits belonging to the exponent must be treated differently. So while an emulated $k{\cdot}m$-bit integer has exactly the same precision as a native $k{\cdot}m$-bit integer (*exact emulation*), floating point emulations with the same property are too expensive in software due to the treatment of rounding modes, denormalisation, etc. Instead, a partially-redundant number representation with less precision is chosen in favour of efficiency (*redundant emulation*). This technique has been introduced by Møller, Knuth, Dekker and other researchers in the late 1960s [3–5]. In the literature, it is referred to with a variety of different names, *double–single*, nativepair arithmetic or double-length floating point. We will use the first nomenclature.

For example, if the hardware provides only s23e8 single float arithmetic, we can emulate a higher precision format by spreading the mantissa over two single float values and use the

exponents to align the two mantissas. The resulting precision equivalents approximately a s46e8 format, but the effective range of the exponent is reduced by 23. In this respect the double–single format is even less precise than an exact s46e8 format and is clearly inferior to a full s53e11 IEEE double. However, the two separate exponents allow us to represent some numbers, e.g. $1 + 2^{-100}$, that are rounded-off even in the double format. So the comparison to a s46e8 format is only approximative.

With native double (s52e11) or extended precision (s64e15) support in current CPUs, this approach is most often used to construct higher precision formats, e.g. by combining two native double precision values into a *quad precision* format [6,7] or even arbitrary precision [8,9]. These approaches are applicable if the underlying floating point format satisfies certain minimal conditions, e.g. faithful rounding and the existence of guard bits.

2.2. Double–single operations

Arithmetic operations on combinations of low precision floating point values require a careful treatment of overflows and underflows between the high and low order parts, using only low precision arithmetic operations. In this section, we will briefly present the algorithms for addition and multiplication. For reference implementations and more details, we refer to Bailey *et al.* and the GNU Multiple Precision Arithmetic Library [10,11].

Addition of two double–single values $c = a + b$ is straightforward. The high-order parts are added, then the low order parts are added including the error from the high order addition. Finally, the overflow from the low part is included in the high part. In pseudocode (using the notation `.hi` and `.lo` for the two components), we get:

Double–single addition $c = a + b$, cf. [10]:

(i) Compute high-order sum and error:
```
t1 = a.hi + b.hi
 e = t1 - a.hi
```
(ii) Compute low order term, including error and overflows:
```
t2 = ((b.hi - e) + (a.hi - (t1 - e))) + a.lo + b.lo
```
(iii) Normalise to get final result:
```
c.hi = t1 + t2
c.lo = t2 - (c.hi - t1)
```

We note that the emulated arithmetic for addition requires 11 native operations.

Multiplication is slightly more complicated. The emulation is significantly cheaper on systems that provide a fused multiply–add ($y = a + b*c$), i.e. rounding and normalisation in the underlying floating point system are not performed until the final addition (the product $b*c$ is not rounded before the accumulation to a). If no fused multiply-add is available, the low and high order products during the multiplication have to be split separately.

Double–single multiplication $c = a*b$, cf. [10]:

(i) Compute initial high-order approximation and error:

- If a fused multiply-add is available:
```
c11 = a.hi*b.hi
c21 = a.hi*b.hi - c11
```

- If no fused multiply-add is available:

```
cona = a.hi*8193.0
conb = b.hi*8193.0
  a1 = cona - (cona - a.hi)
  b1 = conb - (conb - b.hi)
  a2 = a.hi - a1
  b2 = b.hi - b1
 c11 = a.hi*b.hi
 c21 = (((a1*b1 - c11) + a1*b2) + a2*b1) + a2*b2
```

(ii) Compute high-order word of mixed term:

```
  c2 = a.hi*b.lo + a.lo*b.hi
```

(iii) Compute (c11, c21) + c2 using Knuth's trick, including low-order product:

```
  t1 = c11 + c2
   e = t1 - c11
  t2 = ((c2 - e) + (c11 - (t1 - e))) + c21 + a.lo*b.lo
```

(iv) Normalise to get final result:

```
c.hi = t1 + t2
c.lo = t2 - (c.hi - t1)
```

We note that 18 native arithmetic operations are required, 32 if no fused multiply–add is available.

Variations of these algorithms have been published; it is for example possible to save a few operations and trade them for an additional branch. We refer to the literature for more details [6,9].

2.3. Normalisation

Both the addition and the multiplication perform a normalisation step at the end of the computation. In an effort to reduce the instruction count in the emulation one can skip this step, thus trading performance for precision. Before normalisation the lower term `t2` in the addition of two numbers a, b has an exponent of at most ExponentOf(max(`a.hi`, `b.hi`))$-23+2$. If `a.hi` and `b.hi` have the same sign, then `t1` has an exponent of ExponentOf(max(`a.hi`, `b.hi`)) or one higher, so the mantissas are already almost aligned, and we could skip the normalisation. In the worst case this leads to the loss of 2 bits of precision in a subsequent operation. However, if `a.hi` and `b.hi` have different signs, then the normalisation step is much more important, as the exponent of `t1` can greatly vary. In the worst case `t1` evaluates to zero and the normalisation step is crucial to transfer the mantissa from the lower to the upper term. In multiplications we are generally on the safe side, since the high order term `a.hi*b.hi` dominates the result and thus its exponent.

The general effect of fewer normalisations is an increase of the redundancy of the format, because now we allow for a double–single c that ExponentOf(`c.lo`) $>$ ExponentOf (`c.hi`) $-$ 23. This is just a specific example of the general trade-off between the latency of operations and the redundancy of the operand representation in arithmetic hardware circuits.

3. Mixed precision iterative refinement

3.1. Background and related work

Iterative refinement methods have been known for more than 100 years already. They gained rapid interest with the arrival of computer systems in the 1940s and 1950s. Wilkinson *et al.* [12–14] combined the approach with accumulated inner products as a means to assess and increase the accuracy of computed results for linear system solvers, and provided a solid theoretical foundation of these methods. The core idea is to use the residual of a computed solution as a right hand side to solve a correction equation. This process is iterated, and it is essential that the residuals are computed with higher than working precision (see below). The algorithm in its original form to solve $\mathbf{Ax} = \mathbf{b}$ for a quadratic $N \times N$ matrix $\mathbf{A}$ reads:

$$\begin{aligned}
&\mathbf{x}^{(0)} = 0 \\
&\mathbf{d}^{(s)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(s)} && \text{compute residual in } \textit{high} \text{ precision} \\
&\mathbf{A}\mathbf{c}^{(s)} = \mathbf{d}^{(s)} && \text{solve equation system in } \textit{low} \text{ precision} \\
&\mathbf{x}^{(s+1)} = \mathbf{x}^{(s)} + \mathbf{c}^{(s)} && \text{accumulate solution in } \textit{high} \text{ precision}
\end{aligned}$$

Wilkinson *et al.* showed that the computed solution $\mathbf{x}^{(s^\star)}$ is the exact solution of $(\mathbf{A} + \delta\mathbf{A})\mathbf{x} = \mathbf{b}$ where $\delta\mathbf{A}$ is dependent on $\mathbf{b}$ and uniformly bounded. The upper bound m of $\|\delta\mathbf{A}\|_2$ depends on the details of the floating point system used, in particular on the intermediate rounding modes of accumulations. $\mathbf{A}$ is "too ill-conditioned" [13] for the precision of the computation if $m\|\mathbf{A}^{-1}\|_2 \geq 1$. In this case $\mathbf{A} + \delta\mathbf{A}$ could be singular in the given floating point system and no solution can be obtained. If, however, $m\|\mathbf{A}^{-1}\|_2 = 2^{-p}$ for some $p > 1$, then the successive iterates $\mathbf{x}^{(s)}$ of the refinement process satisfy ($\mathbf{x}$ denoting the exact solution):

$$\|\mathbf{x} - \mathbf{x}^{(s+1)}\|_2 \leq \frac{2^{-p}}{1 - 2^{-p}} \|\mathbf{x} - \mathbf{x}^{(s)}\|_2,$$

and the refinement procedure converges to working accuracy.

They also suggest the use of a LU decomposition to solve the system quickly given multiple right hand sides. But the procedure in itself is very general and can in principle be used with any solver, especially iterative ones for large sparse matrices.

A lot of research efforts have been devoted to improve this method, and in the following, we will briefly explain selected approaches that are relevant for the context of this paper. For a more comprehensive overview and references, see Demmel *et al.* and Zielke and Drygalla [15,16].

Turner and Walker [17] analyse the applicability of the iterative solver GMRES(m) in the context of iterative refinement methods and conclude that they can achieve the same accuracy as a reference double precision solver with roughly the same arithmetic work. The original GMRES(m) solver already involves restarts and is therefore particularly well suited for the interleaving of low precision computations with few high precision corrections. The numerical examples in their publication are based on discretised linear elliptic PDEs and the scheme is implemented in FISHPACK and achieves noteworthy speedup factors.

Geddes and Zheng [18] propose a mixed precision linear Newton iteration. They demonstrate their method for a wide range of different problems such as least-squares fits of overdetermined systems and the solution of multivariate nonlinear polynomial equations.

For each specialised adaptation of their algorithm, they provide a detailed cost analysis and thorough numerical tests. This work, in particular, demonstrates that the algorithmic mixed precision optimisation can be successfully applied to various problems.

Langou *et al.* [19] evaluate iterative refinement techniques for dense matrices on a wide variety of modern CPUs such as the Opteron, Itanium, Cray, PowerPC and Cell processors. The speedup is achieved by executing the most expensive operation, namely the LU decomposition, in lower precision. The decomposition can be computed much faster in single than in double precision and dominates the solution time with $\mathcal{O}(N^3)$ arithmetic work. Once the LU decomposition is available, few iterations of the iterative refinement scheme (each requiring only $\mathcal{O}(N^2)$ work) suffice to solve the problem. Adapting results from Stewart [20] the authors provide a convergence condition with an upper bound for the number of required iterations, which depends on the low and high precision bit length and the matrix condition. More general error analysis of fixed and mixed precision iterative refinement techniques is provided by Stewart and Higham [20,21].

3.2. Mixed precision iterative refinement algorithm framework

In the following, we describe a mixed precision iterative refinement algorithm for the large sparse linear equation systems arising in FEM simulations. In particular, it is not feasible to apply an LU decomposition or similar direct methods in this context.

The core idea of the algorithm is to split the solution process into a computationally intensive but less precise inner iteration and a computationally simple but precise outer correction loop. The loops are coupled by a scaling heuristics to enlarge the exponent range locally and consequently the dynamic range of the scheme as a whole. To solve the defect equation, an arbitrary iterative solver running in low precision can be employed. In our implementation and numerical tests, we make use of conjugate gradient and multigrid solvers (cf. Section 5.1).

Another view on this scheme is to interpret the inner solver as a black-box preconditioner ensuring the reduction of residuals by a prescribed number of digits for an outer Richardson iteration. Other outer solvers are of course conceivable, for example, a multigrid or a BiCGStab iteration.

Let $\varepsilon_{\text{inner}}$ and $\varepsilon_{\text{outer}}$ denote the stopping criteria for the inner and outer solver and $\mathbf{Ax} = \mathbf{b}$ the linear equation system to be solved in high precision. Furthermore, α and d^0 are two scalars used for scaling and convergence control. Defect and iteration vectors are labelled $\mathbf{d}$ and $\mathbf{c}$, respectively. Subscript $_{\text{low}}$ and $_{\text{high}}$ indicate the precision of the data vectors. The conversion between the two precision formats is intentionally left abstract, the details depend on the target hardware: On some systems, the conversion might mean duplicating the values into a new array with different precision, while on other systems, the values can be casted on the fly when they are first accessed. In any case, no special operations are performed apart from a regular number format conversion. The template form of the algorithm can then be written as:

General template of the mixed precision iterative refinement solver. Input: System matrix $\mathbf{A}_{\text{high}}$, right hand side $\mathbf{b}_{\text{high}}$, convergence parameters $\varepsilon_{\text{inner}}$ and $\varepsilon_{\text{outer}}$. Output: Solution $\mathbf{x}_{\text{high}}$.

(i) Set initial values and calculate initial defect:
$\alpha_{\text{high}} = 1.0$, $\mathbf{x}_{\text{high}} =$initial guess,
$\mathbf{d}_{\text{high}} = \mathbf{b}_{\text{high}} - \mathbf{A}_{\text{high}}\mathbf{x}_{\text{high}}$,
$d^0_{\text{high}} = \|\mathbf{d}_{\text{high}}\|$.

(ii) Set initial values for inner solver and convert data to low precision:
$\mathbf{A}_{\text{low}} = \mathbf{A}_{\text{high}}$, $\mathbf{d}_{\text{low}} = \mathbf{d}_{\text{high}}$, $\mathbf{c}_{\text{low}} =$initial guess,
$d^0_{\text{low}} = \|\mathbf{d}_{\text{low}} - \mathbf{A}_{\text{low}}\mathbf{c}_{\text{low}}\|$.
(iii) Iterate inner solver until $\|\mathbf{d}_{\text{low}} - \mathbf{A}_{\text{low}}\mathbf{c}_{\text{low}}\| < \varepsilon_{\text{inner}} \cdot d^0_{\text{low}}$.
(iv) Update outer solution: $\mathbf{x}_{\text{high}} = \mathbf{x}_{\text{high}} + \alpha_{\text{high}}\mathbf{c}_{\text{low}}$.
(v) Calculate defect in high precision: $\mathbf{d}_{\text{high}} = \mathbf{b}_{\text{high}} - \mathbf{A}_{\text{high}}\mathbf{x}_{\text{high}}$.
(vi) Calculate norm of defect: $\alpha_{\text{high}} = \|\mathbf{d}_{\text{high}}\|$.
(vii) Check for convergence ($\alpha_{\text{high}} < \varepsilon_{\text{outer}} \cdot d^0_{\text{high}}$);
otherwise scale defect: $\mathbf{d}_{\text{high}} = (1/\alpha_{\text{high}})\mathbf{d}_{\text{high}}$ and continue with step (ii).

Note that in high precision, we only need to assemble the system once and perform a matrix–vector multiplication, a vector update and a norm calculation in each outer iteration.

Since the matrix is assembled in high precision, we gain two additional advantages: First, in the hardware-oriented scenario that we target with this work, we need the final solution in high precision on the CPU to incorporate it in the main application (cf. Sections 4 and 8.1). Second, although some systems can be represented in low precision, they cannot be assembled therein. For instance, the transformation to the reference element used in certain FEM implementations suffers strongly from cancellation effects in single precision during the computation of the Jacobi determinant for very anisotropic elements. Consequently, the approach is inapplicable if the defect equation cannot be represented in low precision at all.

As scaling heuristics, we tested both normalising the defect vector (as in the above algorithm) and shifting the exponent by multiplying with an appropriate power of two, and both work equally well.

As stopping criterion, we evaluated predefining a threshold for the (relative) reduction of the norm of the residuals by a fixed number of digits, and executing the inner solver for a fixed number of iterations. The latter has the disadvantage that eventually too many inner iterations, especially in the last step, or too frequent update steps are performed. Which of these drawbacks is more critical depends on the actual hardware, but in general an update step can be considered costly compared to a solver iteration. This is particularly true for co-processor architectures where the data bus between the different chips is comparatively slow. In our numerical experiments, we were not able to construct test cases (apart from pathological examples that are not representable in low precision) which we could not solve by iterating the inner solver until a moderate error reduction of two digits in each outer iteration was achieved. We discuss the influence of the stopping criterion further in Section 5.4.

We finally note that the iterative refinement approach can be cascaded to form a multilevel mixed precision solver. Instead of solving a given problem expensively in high precision, the solution process is split up recursively into a series of defect equations which are solved in successively decreasing precision. This way, most of the arithmetic work is performed on cheap low precision PEs. The feasibility of this cascade depends on the available hardware accelerated precisions, and the minimal precision necessary to gain relative accuracy reasonably fast on a defect problem of given condition.

4. Acceleration scenarios

There are several parallel architectures which are particularly suitable for the mixed precision iterative refinement algorithms presented in this paper. The idea is to use the parallel low precision hardware to quickly gain relative digits of accuracy and accumulate

these gains in high precision. The accumulation and the residual computation, that must be performed in high precision, may either be executed on a general purpose processor (typically the CPU of the host system), in a more expensive high precision configuration of the parallel device, or with an emulation on the parallel device. Despite the high resource usage, a high precision configuration is preferable to the slow emulation (cf. Section 2.2) and should be used if available. Therefore, we use this distinction in the following discussion of parallel co-processors, and classify them into two categories:

S: devices that can only compute in single floating point precision. For the double precision computation they have to utilise the CPU or emulate them.
SD: devices that can compute both in single and double precision. Thereby, the single performance is usually much higher than the double performance.

For one representative from each class, we give more details on the configuration for the mixed precision iterative refinement in Sections 4.2 and 4.3, and present concrete implementations of a mixed precision solver and results in Sections 6 and 7. The sections also examine the resulting advantages under different aspects. The evaluation of the GPU demonstrates which speedups can be achieved with this approach, the FPGA implementation concentrates on the possible resource savings assuming unchanged accuracy and performance requirements. In other words, we first examine the savings in time and then in space.

4.1. Parallel co-processors

This section offers a glimpse of the diversified field of parallelism on a chip. By discussing several important classes of parallel architectures along with some representatives we hope to convey the lurking possibilities of such hardware configurations in the context of mixed precision methods. Overviews of parallel computing systems, focusing especially on data-stream-based processing, are provided by Hartenstein [22,23]. Sankaralingam *et al.*, Guo *et al.* and Taylor *et al.* discuss the various types of parallelism and their advantages and disadvantages [24–26], and Suh *et al.* and Strzodka compare the performance of different parallel architectures on scientific kernels and PDE problems [27,28].

4.1.1. Field programmable gate array (FPGA). FPGAs (class SD) allow us to configure any hardware design and run it as though such a chip had been fabricated. This is very practical for the testing of future chips but is also widely exploited to map problem solvers into a hardware configuration, generating a processor designed specifically for the application in mind.

FPGAs have been integrated on a wide variety of boards. For use in a PC one obtains a PCI, PCI-X (PCI extended) or PCIe (PCI express) board with one or more FPGAs and usually additional local memory and a micro-controller. There exist also stand-alone FPGA boards that connect to a PC for example via USB, but due to the low bandwidth connection these are obviously less suitable for an interleaved CPU–FPGA computation. In the mixed precision iterative refinement algorithm one can then configure the FPGA to perform many parallel low precision computations of the inner solver and accumulate the corrections on the CPU or reconfigure the FPGA for high precision computations. However, the latter option is only feasible if special configuration techniques such as partial or compressed configurations are available, minimising the configuration overhead. In case of the CPU utilisation, the

overhead comes from the transportation of the data back and forth to the card, which can be large for the PCI bus, but should be much smaller on the newer PCIe bus. Systems which connect the FPGA directly to a wide system bus such as the Cray XD1 or the SGI Altix offer additional advantages concerning the available bandwidth to the parallel co-processor.

Since the FPGA allows the configuration of any number format, one can further cascade the mixed precision approach and use several different precision formats within the FPGA. In particular, it is easy to generate fused operations that apply only one rounding to the final result of a composed operation, e.g. a dot product. Large accumulators can also simply use a higher intermediate precision for better accuracy.

4.1.2. Processor array/multi-core processor. Both terms describe chips comprising many PEs that operate in parallel. Processor arrays usually refer to architectures with simple PEs that communicate with each other directly, have a common control unit that assigns data to them, or simply share local memory for data exchange. The term multi-core processors describes architectures with more complex PEs, that often themselves contain multiple elementary PEs. Beside the registers, both classes of processors typically feature small, very fast, local PE memories and larger inter chip memory accessible from all PEs. In contrast to caches, this three-level memory hierarchy is under full control of the programmer, which enables the design of highly efficient application specific data-flow, but also significantly increases the program complexity, especially as even more memory levels may be present and the communication over a bus with the host must also be taken into account. A framework specifically dedicated to a more portable and reusable exploitation of these memory hierarchies has been presented by Fatahalian *et al.* [29].

The Clearspeed CSX architectures [30] are examples of processor arrays. A board with several CSX chips can be integrated into a workstation via the PCI-X bus. The PEs in the CSX architecture (class SD) support single and double precision arithmetic at basically the same speed. However, in favour of a reduction of internal and external bandwidth requirements it is still more advantageous to utilise the numerous parallel PEs for single float computations and apply the iterative refinement algorithms described in this paper. Smaller number formats also allow us to fit more data words into the very restricted PE memories, which is highly beneficial for many algorithms which have to store local state in addition to the processed data streams. So there are multiple reasons for using single float arithmetic even when the peak double precision performs is equal, but the transition from a pure double precision FPU to a dual-mode FPUs costs relatively few transistors, so that in future Clearspeed might decide to offer architectures that perform twice the number of single float than double float operations.

A representative of the multi-core processors is the Cell BE processor (Sony, Toshiba, IBM) [31]. It is available as a blade server or a PCIe accelerator board for the PC [32]. The Cell BE (class SD) also offers single and double float arithmetic, but the peak single float performance is more than ten times higher than the peak double performance. In common scientific computing applications this discrepancy results in a slowdown by a factor of 5–20 for double precision [33]. Moreover, similar to the CSX chips, a smaller number format increases the word throughput and local storage of the PEs in the Cell BE.

The general purpose CPU is quickly developing towards a multi-core processor. After the double-core versions both Intel and AMD already announced quad-core processors for the near future. Although current CPUs still offer little parallelism in comparison to the other

architectures discussed here, with the growing number of cores and a possible widening of the SSE unit the situation might change in the future.

4.1.3. Application specific hardware. Some application specific hardware products expose a lot of parallel PEs that are well suited for the solution of PDEs. Two examples from the world of graphics applications are the AGEIA PhysX processor [34] and modern GPUs. Little is known about the internal structure of the PhysX processor. But one knows that it is a highly-parallel architecture and the company claims that the computing model is particularly well suited for FEM simulations of physical phenomena. Allegedly the computation units use single float (class S?), which in view of the mixed precision methods would actually give better hardware efficiency. However, it is not clear how the data-flow for the mixed precision approach could be organised on the PhysX processor and whether a high precision emulation or corrections on the CPU were more efficient. In the latter case the low bandwidth of the connecting PCI bus could be a problem for a fast CPU–PhysX mixed precision configuration.

Modern graphics cards contain mainly one GPU and video memory. The seldom exceptions are cards that implement the different parts of the graphics pipeline in different chips rather than one GPU, and cards with two GPUs on one card. Graphics cards are connected to the PC via the AGP or the faster PCIe bus. Most low cost GPUs are integrated directly into the chipset. GPUs offer single float computations throughout the graphics pipeline (class S). Although the numerical access to the parallel PEs is obscured by the graphics specific API, the wide availability and cost efficiency of the hardware makes them very attractive candidates for mixed precision methods. The GPU consists of independent groups of PEs. Within each group there is implicit (instruction level parallelism) and explicit data exchange (four vector components) through registers. Above the registers GPUs have no user controllable local PE memory where local state could be stored, instead they feature dedicated cache arrangements optimised for 2D coherent memory access patterns. This means that intra group communication happens through high level memory that resides on the graphics card outside the GPU. For the GPU this implies many memory transfers to a much higher memory level than for the other architectures, including a lower bandwidth for these transfers. On large data sets the GPU can still compete with the other parallel chips, because the external bandwidth is pushed to the limit with dedicated graphics memory chips and the entire system is optimised for latency minimisation. But the lack of intermediate local memories leads more often to bandwidth bound algorithms on the GPU and considerable effort must be devoted to trading computation for bandwidth with special, often application specific algorithmic reformulations [35].

4.2. Graphics processor unit (GPU)

Since 2003, graphics processors (GPUs) offer programmable floating point arithmetic. Together with the high memory bandwidth and tremendous computational performance GPUs have emerged as powerful numerical co-processors. A new field of research commonly called *general purpose computation on graphics hardware (GPGPU)* has evolved. We refer the reader to the survey paper by Owens *et al.* [36], and to the GPGPU community web site [37] for online material with introductions, paper archive, course notes, tutorial codes and further information. For a GPGPU introduction without any graphics terminology see Strzodka *et al.* [38].

Although GPUs are just one representative among the many powerful parallel computing devices discussed in the previous section, they attract additional attention because of their ubiquitous availability and excellent price-performance ratio. Graphics processors built into PC chipsets usually offer only very limited computational performance, but buying and installing a more powerful graphics card off the shelf is an easy and comparably inexpensive procedure. Therefore, many users would consider upgrading their GPU if they could gain significant speedups for their applications.

GPUs natively support a quasi-IEEE s23e8 floating point format which lacks denormalisation and always rounds to zero. For a detailed analysis of the differences between the standard IEEE single precision and the format on the GPU, refer to Hillesland and Lastra [39] and Daumas *et al.* [40]. These differences are however minor in the context of this paper.

Da Graca and Defour analyse the applicability of double–single emulation techniques and conclude that because of faithful rounding and guard bits, the emulated precision arithmetic is well suited for GPUs [41]. Extension of this technique to complex and quad-precision numbers has been presented by Hitz and Payne [42], and a residue-based number representation with variable precision is discussed by Thall [43].

The emulated high precision multiplication (cf. Section 2.2) requires less instructions if a fused multiply–add operation is present. This is true for GPUs, but the composed MAD (multiply–add) instruction is not necessarily fused with respect to floating point rounding, i.e. the computed result must be the rounded exact result of the composed operation. ATI GPUs (X1K series) offer a fused multiply–add with respect to rounding, so that here the shorter emulation of the multiplication can be used, while NVIDIA GPUs (GeForce 6 and 7 series) apply intermediate rounding/truncation and thus require the longer emulation for the multiplication (according to the vendors' respective developer support).

In contrast to emulation, Dale *et al.* [44] suggest equipping GPUs with dual-mode PEs, realised through small-scale reconfigurability. As discussed in the introduction (cf. Section 1.1), this improves double precision performance over emulation but lowers the transistor efficiency for single float arithmetic, so the decision depends on how often double precision arithmetic is indispensable for the GPU. In view of our mixed precision results (cf. Section 6) we rather discourage the overall integration of double precision PEs into GPUs, as they are not needed in high numbers to achieve accurate results in PDE-based physical simulations, i.e. the application area in which computer games have probably the highest demand for better accuracy. But adding some dual-mode PEs, e.g. only the moderately large double precision adders, might be useful to enable a fast execution of the high precision corrections on the GPU itself, rather than always delegating these tasks to the CPU.

The double–single emulation on the GPU is not always sufficient, as the double format has a clearly larger mantissa and exponent range than the emulated format: $52/46 = 1.13$, $2^{11}/2^8 = 8$, and three term emulations become truly expensive. Therefore, we perform the high precision corrections on the CPU rather than emulating them on the GPU (cf. Section 6). Moreover, for complex applications the corrections on the GPU do not eliminate the necessity of data transfers with the host. For complex applications it is infeasible and also unnecessary to reimplement thousands of lines of performance uncritical code on the GPU. Instead, the GPU accelerates the more regular, most demanding parts of the algorithms, and leaves the subsequent processing of the results to the well tested, higher abstraction code on the CPU.

4.3 Field programmable gate array (FPGA)

Hardwired processors operate on fixed-width formats and thus usually have dedicated single or double precision FPUs. They also have fixed data paths and synchronised scheduling. FPGAs have fully configurable memory and logic elements, data paths and clock generators, and thus a lot of freedom in adapting the hardware to the application.

- The input and output data formats depend on the application and not on the hardware, e.g. some imaging devices deliver data with 12-bit precision which is stored in 16-bit shorts for CPU processing.
- FPGAs are natural mixed precision processors, there is no need to utilise the same number format throughout the algorithm, or even the same operation, e.g. for a more accurate accumulation of s23e8 data we may configure a s23e8 + s36e11 → s36e11 accumulator.
- We can generate more efficient custom fused operators, e.g. redundant number representations allow us to evaluate a sum of multiple addends $\sum_i a_i$ much faster with the successive computation of b, c, such that $b + c = a_i + a_{i+1} + a_{i+2}$, rather than the repeated reduction $b = a_i + a_{i+1}$.
- There are many options to trade latency, throughput and area in operations, e.g. an $n \times n$-bit adder can have $\mathcal{O}(1)$ area and $\mathcal{O}(1/n)$ throughput or $\mathcal{O}(n)$ area and $\mathcal{O}(1)$ throughput.
- While FPGAs have dedicated logic and memory elements, some also have configuration elements which can be used as logic or memory. Thus additional tradeoffs between computation- and lookup-based operators are possible, e.g. a 4 input 1 output Boolean operator is realised through a 2^4-bit configuration table, which can alternatively be used to store 16-bit.
- The freedom in the design of custom operators makes alternative number representations sometimes feasible on the FPGA, e.g. in the logarithmic number representation we store $\log(a)$ instead of a and obtain the big advantage that multiplications translate into additions $\log(a \cdot b) = \log(a) + \log(b)$.
- All PEs operate concurrently in a FPGA, so the maximum efficiency is obtained when all PEs contribute to the solution and never stall. Thus to avoid areas with idle operators which are not required all the time, one can reuse parts of these operators in a modular design for other computations, e.g. a float adder can reuse the adder of the mantissas for integer operations.
- When reusage of a configuration is difficult because of very different operators, reconfiguration overhead can be minimised with special reconfiguration techniques, e.g. with partial reconfigurability we can replace only a part of the circuit and thus require a much smaller configuration bitstream.

The application of these techniques from scratch requires intrinsic knowledge of a hardware description language (HDL). However, HDLs now offer many levels of abstraction and although floating point numbers are not part of the language, parameterised libraries are available [45,46]. Automatic *a priori* analysis can be performed to determine the necessary precision in composed operations [47]. The tradeoffs between latency, throughput and area for floating point formats have been studied in detail [48–50]. Analysis of logarithmic number systems has also revealed under which conditions this representation is advantageous [51,52]. For more background and additional information we refer to general

surveys of reconfigurable hardware presented by Bondalapati and Prasanna, and Compton and Hauck [53,54].

The above optimisations reduce the area consumption locally on the level of individual and composed operations. The mixed precision iterative refinement applied to the FPGA allows us to save resources globally by reducing the precision of most intermediate operations. Although the FPGA can be configured with double precision FPUs, this consumes so many resources, that we rather use it similar to the GPU, as a fast parallel, low precision processor and perform the double precision corrections on the CPU or a micro-controller on the FPGA board. In case of the FPGA we examine the dual aspect of the reduced precision and analyse the savings in area rather than time (cf. Section 7).

5. Test procedure and algorithmic properties

Before we present results of the proposed schemes on actual hardware configurations in the next two sections, we want to discuss several properties of the approaches in more detail and describe our test procedure in general.

5.1. General test procedure

For a given test function, we solve several variants of the Poisson equation $-\Delta\mathbf{u} = \mathbf{f}$ prescribing zero Dirichlet boundary conditions. Such variants of the Poisson problem are common building blocks in CFD or CSM and numerical simulation in general. For instance, projection type schemes in Navier–Stokes simulations [55] lead to Pressure–Poisson problems, often the most time-consuming step at the core of the computation (especially in non-stationary simulations). Another example is that of grid deformation techniques [56] that move grid points towards regions in the simulation domain with high errors, and which calculate the new positions with one Poisson problem in each deformation step.

Conforming bilinear finite elements (Q_1) are used for the spatial discretisation of the exemplary domain $\Omega = [0, X] \times [0, Y] \subseteq \mathbb{R}^2$ for different levels of refinement of an underlying generalised tensorproduct mesh, leading to problem sizes of $N = 3^2$ to $N = 1025^2$ unknowns (levels 2–10, respectively). In this section and in the discussion of actual acceleration scenarios in the following sections, all results are based on a uniform refinement of the unit square, and we do not employ a stencil but assemble the matrix fully. In Section 8, we will extend the results on anisotropic discretisations and more realistic domains, but the matrix structure and hence the cost for a matrix–vector multiplication remains the same. The discretisation scheme yields a band matrix with nine bands which we store as separate vectors with appropriate zero padding.

We define a test function $\mathbf{u}_0 : \Omega \rightarrow \mathbb{R}$ analytically as $\mathbf{u}_0(x, y) = x(X - x)y(Y - y)$, $(x, y) \in \Omega$. With zero Dirichlet boundary conditions, neither additional data approximation errors nor cubature errors in the right hand side are introduced which would obfuscate the accuracy issues we want to examine. The function and its analytically known derivatives are used to predefine a right hand side $\mathbf{f} = -\Delta\mathbf{u}_0$, yielding a Poisson problem with analytically known solution. During our experiments we successfully evaluated many additional test functions, but throughout this paper, we present all results based on the above test function to provide better comparability of our results.

We employ the following solvers for the numerical tests:

CG as an example of a basic iterative Krylov space solver. We use the standard formulation of the solver, without preconditioning or damping.
Multigrid with Jacobi smoother (**MG_JAC**) as an example of a more advanced solver, capable of dealing with small grid distortions while being unbeatably fast for Cartesian meshes. We employ an F-cycle and vary the damping factor and the number of pre- and post-smoothing steps as necessary.

We compute two quality measures in each test case, the L_2 error and the number of iterations until solution. The L_2 error (integrated over Ω) of the computed solution against the analytically known reference result should behave like $\mathcal{O}(h^2)$ (mesh width $h = 1/\sqrt{N}$), yielding an error reduction factor of four in each refinement step. The CG solver should double the amount of iterations per refinement level while the multigrid solvers should converge independent of the level.

5.2. Influence of the input data precision

In the first test series, we analyse the influence of the input data precision on the overall error. The right hand side for the Poisson problem is therefore precomputed in double precision as the *discrete* Laplacian $\mathbf{f} = -\Delta_h \mathbf{u}_0$ to avoid discretisation errors and then truncated to double, single or half [57] float precision, while the solver itself is executed in double precision. The computed results are then compared (again in double precision) with the known reference solution. This test is performed on a uniform subdivision of the unit square. Table 1 subsumes the impact of reducing the input data precision, exemplarily for the problem sizes $N = 257^2$ and $N = 513^2$.

We observe that representing all vectors in double precision is essential. Although the matrix assembly and the computation are always performed in double precision, the reduction (clamping) of the right hand side to single precision already costs approximately three digits of accuracy in the result. The furthergoing reduction to half precision costs an additional three digits. In view of the targeted co-processor scenario, these results support our claim that it does not make sense to improve the internal computational precision without introducing a higher precision format for input and output.

Table 1. Errors caused by reduced input data precision for a double precision solver (cf. Section 5.2).

Level	*Half precision*	*Single precision*	*Double precision*
8	2.333E-6	7.718E-10	2.750E-13
9	1.008E-6	7.726E-10	1.051E-12

5.3. Accuracy in single and double precision

In this test series, we compare how the *computational precision* in the solution process influences the *accuracy of the result*. In particular, we want to emphasise the highly non-monotonic relation between them.

We perform the matrix assembly and the error calculations in double precision. The arising systems are solved in single and double precision, respectively. For the solution in single precision, we cast the matrix and vectors to single precision. Table 2 lists the number of multigrid cycles, the error compared to the reference solution and the reduction factor in the

Table 2. Influence of solver precision on solution accuracy with increasing problem size (level of refinement), cf. Section 5.3.

	Single precision			*Double precision*		
Level	*Cycles*	*Error*	*Reduction*	*Cycles*	*Error*	*Reduction*
2	1	2.391E-3		1	2.391E-3	
3	2	5.950E-4	4.02	2	5.950E-4	4.02
4	2	1.493E-4	3.98	2	1.493E-4	3.99
5	2	3.750E-5	3.98	2	3.728E-5	4.00
6	2	1.021E-5	3.67	2	9.304E-6	4.01
7	2	6.691E-6	1.53	2	2.323E-6	4.01
8	2	2.012E-5	0.33	2	5.801E-7	4.00
9	2	7.904E-5	0.25	2	1.449E-7	4.00
10	2	3.593E-4	0.22	2	3.626E-8	4.00

error for refinement levels 2–10 (3^2–1025^2 unknowns). The stopping criterion for all tests is identically set to $\varepsilon_{\text{outer}} = 10^{-3}$. This threshold suffices to achieve the expected error reduction by a factor of four and thus avoids the potential accumulation of round-off errors by performing too many inner solver iterations.

We observe that the double precision solver exhibits the expected error reduction by a factor of four per refinement. The single precision solver however does not. From levels 5–7, the error reduction degrades slowly to a mere factor of 1.5, and for levels 8 and 9, the error even increases again, thus wasting the additional work and memory requirements for these levels of refinement. Note that this behaviour cannot be detected based on evaluating only the defects during the solution process. In both cases, the reduction rate of the residuals is identical. This is especially critical in practical scenarios where a reference solution is naturally not available. The consistent reduction of residuals on finer levels misguides one to believe that also the overall accuracy is improved, but in fact the opposite is the case.

We examined other causes of the accuracy degradation. Neither performing more nor performing less iterations of the inner single precision solver improves the accuracy, indicating that the issue is really the lack of precision and not a side effect of accumulated round-off errors.

In our experiments, we encountered many examples that, although performing worse from the point of view of accuracy, behaved much better from a practical point of view. For higher levels of refinement (i.e. worse matrix condition), the single precision solvers simply stopped converging properly, which can be detected in practice by monitoring the convergence rate of the solvers.

In summary, single precision solvers for certain examples fail to gain accuracy upon further refinement. In some cases, they diverge or just slowly degrade their rate of convergence, until no further error reduction can be achieved. In other cases, instead of degrading accuracy, they increase the error again. This is especially critical as there is no straightforward means to detect this last case during the solution process.

5.4. Influence of the stopping criteria in iterative refinement

We evaluated two stopping criteria for the mixed precision iterative refinement algorithm, executing the inner solver for a fixed number of iterations or until the norms of the residuals have been reduced by a predefined number of digits. Our first result is quite obvious: As long as not too much work is demanded of the inner solver with respect to precision (e.g. gaining

Table 3. Influence of the stopping criterion (cf. Section 5.4). Mixed precision iterative refinement solver based on multigrid executing 1, 2 or 3 cycles ($i1$,$i2$,$i3$), or gaining 1, 2 or 3 digits ($\delta 1$, $\delta 2$, $\delta 3$) in each outer iteration, respectively. The column labelled "double" lists reference values obtained in native double precision. Notation: Colon separates outer and sum of all inner iterations.

Level	*Double*	*i1*	*i2*	*i3*	*δ1*	*δ2*	*δ3*
8	8	8:8	4:8	4:12	8:8	5:9	4:11
9	8	8:8	4:8	4:12	8:8	5:9	4:11
10	8	8:8	5:10	5:15	8:8	5:9	4:11

less than four digits in one outer iteration), the choice of the termination criterion does not disturb convergence or the accuracy of the final result. The latter is effectively determined by the number of *outer* iterations, which we always control by monitoring global defects in double precision.

Moreover, we experienced that the choice of the inner solver has the greatest impact on the performance of the iterative refinement scheme. In the following paragraphs, we present numerical results for both the unpreconditioned CG solver as well as the multigrid solver with Jacobi smoother. Like all results in this section, the measurements are based on a uniform refinement of the unit square and the solver is iterated until the global residuals have been reduced by ten digits.

Table 3 lists the number of iterations for the iterative refinement scheme using multigrid (with $2 + 2$ Jacobi smoothing steps) as inner solver.

We observe that this type of solver only leads to a mild increase in overall iteration count, if any. A more detailed analysis of the convergence history reveals that the increase can be attributed to our choice of convergence control. When demanding too much error reduction per outer update step, it might happen that the system is not solved until the predefined ten digit reduction, but further. We conclude that the multigrid solver is very well suited to be used within a mixed precision iterative refinement solution procedure.

The CG solver, however, shows an entirely different convergence behaviour, as shown in table 4. Note that the overall iteration count increases dramatically. We first assumed that this is caused by the lack of precision, namely due to roundoff errors and cancellation, subsequent steps in the algorithm fail to compute orthogonal search directions. We therefore implemented a variation of our proposed scheme, a fixed precision iterative refinement algorithm executing the inner CG solver in double precision. To avoid distractions by performing superfluous iterations due to the less frequent convergence control of the outer solver, we also computed the global defect after each inner iteration and thus could terminate the solution process as soon as it fell below $\varepsilon_{\text{outer}}$.

As we see from the numbers in table 5, the lack of computational precision is only one reason for the dramatic increase in iterations. We clearly see that the CG solver reacts

Table 4. Influence of the stopping criterion (cf. Section 5.4). Mixed precision iterative refinement solver based on CG executing 100, 250 or 500 iterations ($i100$, $i250$, $i500$), or gaining 1, 2 or 3 digits ($\delta 1$, $\delta 2$, $\delta 3$) in each outer iteration, respectively. The column labelled "double" lists reference values obtained in native double precision. Notation: colon separates outer and sum of all inner iterations.

Level	*Double*	*i100*	*i250*	*i500*	*δ1*	*δ2*	*δ3*
8	342	13:1300	4:1000	4:2000	10:1047	5:861	4:944
9	676	68:6800	9:2250	5:2500	10:2002	6:2256	5:2380
10	1357	264:26400	40:10000	12:6000	10:4504	6:4501	6:5242

Table 5. Influence of the stopping criterion (cf. Section 5.4). Fixed precision iterative refinement solver based on CG executing 100, 250 or 500 iterations ($i100$, $i250$, $i500$), or gaining 1, 2 or 3 digits ($\delta 1$, $\delta 2$, $\delta 3$) in each outer iteration, respectively. The inner solver uses double precision as well, and an immediate global convergence check after each inner iteration is applied. The column labelled "double" lists reference values obtained in native double precision. Notation: Colon separates outer and sum of all inner iterations.

Level	*Double*	*i100*	*i250*	*i500*	*δ1*	*δ2*	*δ3*
8	342	10:970	2:368	1:342	10:859	5:673	4:544
9	676	66:6588	8:1802	2:763	10:1648	5:1363	4:1098
10	1357	261:26013	38:9470	8:3629	10:3472	5:2379	4:2233

very delicately to frequent restarts. The reason for it is that the CG algorithm builds up a search space with its utility vectors, and this information is lost when the solver is restarted.

This problem can be partly alleviated by reusing the utility vectors. Obviously this is only possible if the inner solver operates always with the same right hand side, i.e. we cannot use the standard iterative refinement formulation. But the CG algorithm already contains an accumulation of the solution vector, and therefore we can integrate the iterative refinement method directly into the core of the algorithm. The high precision correction is then performed on the residual directly and not on the solution vector, for details see the *residual guided CG* variant [58]. In the limit, when the inner precision is exactly the same as the outer precision we simply obtain the original CG algorithm. The reuse of the utility vectors is particularly helpful when the inner precision is very low (below single float) and many outer solver iterations are necessary to ensure convergence. Therefore, we have studied this variant in the context of reconfigurable hardware, where the inner precision is not limited by fixed hardware functionality but can be chosen arbitrarily [58].

5.5. Implementation on the CPU

Targeting the outstanding parallel compute power, we first used parallel devices for performance gains with mixed precision methods [59,58]. Recent work of Langou *et al.* [19], showing speedups for a *direct* LU solver in the mixed precision iterative refinement setting on the CPU, encouraged us to also implement a CPU version of the mixed precision *iterative* solvers. However, due to the sparsity of the matrix we cannot benefit from readily available optimised BLAS and LAPACK implementations such as GotoBLAS, ATLAS or vendor-tuned packages. Benchmarking reveals that approximately 85% of the time is spend in the sparse matrix–vector multiplication, for which no BLAS or LAPACK routines are available: For the banded matrix structure and storage as individual vectors, we would require a "saxpy_v" operation ($y_i = y_i + a_i x_i$ instead of $y_i = y_i + \alpha x_i$) to implement the multiplication (cf. Section 5.1). Without such a SSE-optimised matrix–vector multiplication, speedups of the refinement scheme over a full double precision format are not achievable. In contrast to the direct LU solver, we would not benefit from the different complexity order of the solution steps in case of iterative solvers, but based on the iteration numbers of the schemes (cf. Section 5.2) we believe that a fully optimised SSE variant would at least execute the mixed precision multigrid solver faster than a full double precision implementation.

6. Solver speedups with GPUs as co-processors

We have previously published first results of the GPU–CPU mixed precision iterative refinement algorithm [59]. Here, we use a slightly modified defect correction procedure as described in Section 3.2 and improve performance by implementing the optimisations discussed in our initial publication. The performance numbers are directly comparable as we continue to use the same test problem, namely the Poisson problem on the unit square with uniform refinement, cf. Section 5.1. Furthermore, we add the emulated double–single floating point format and a multigrid solver with Jacobi smoothing to our GPU framework.

6.1. GPU–CPU configuration

For the solution of a linear equation system with an iterative refinement technique, the GPU executes the low precision (single float) inner solver (cf. Section 3.2) as a co-processor to the general purpose CPU which accumulates the corrections and computes new defects in high precision (double float). For optimal performance of the inner solver the involved matrices and vectors must be stored locally and thus duplicated into the video memory of the graphics card. During this transfer, the conversion from double to single float is performed. The computed single precision correction is transferred back to main memory and later implicitly converted to double precision by the CPU in the process of accumulating it to the current double precision solution vector.

Analogously, the computation is performed on the GPU alone when using the emulated double–single floating point format without any transfers of intermediate results. Again, the data is stored locally and the double precision values are split into their high- and low-order components during the transfer of the initial matrix and data. After the solution on the GPU, the two components of the result vector are read back and combined on the CPU in double precision to form the final high precision result.

The multigrid solver employs F-cycles and we configure it to perform $2 + 2$ and $4 + 4$ pre- and post-smoothing steps, respectively. In our tests, we compare execution time (including all required transfers and set-up times of the co-processor) and accuracy of the solution against a carefully optimised version of the corresponding solver, taken from the FEAST package (cf. Section 8.1).

We execute all performance tests on a high-end workstation PC with the following hardware details:

- CPU: AMD Athlon64 X2 4400 + , 2.2 GHz, 2MB cache
- GPU: NVIDIA GeForce 7800 GTX, 24 fragment pipelines, 430 MHz, connected via PCIe x16 to an NFORCE4 chipset
- CPU memory: 2 GB DDR400, PC3200
- GPU memory: 256 MB DDR3, 600 MHz, memory interface 256-bit

Note that all tests are performed as single-threaded, serial jobs and only benefit from the dual-core architecture in that other threads on the machine do not interfere with the computation.

We have also performed all tests on a comparable system with two Intel Xeon EM64T 3.6 GHz processors and a NVIDIA Quadro FX 4500 graphics card, and achieved similar results. We conclude that no high-end graphics card is required (the Quadro boards cost about four times as much as the mainstream GeForce models) for good performance, though this

might be an issue in a cluster environment where reliability is of much higher importance than in a single workstation.

In the following paragraphs, we compare absolute performance of a double–single solver and the iterative refinement scheme as speedup against the CPU which computes in double precision. We also analyse relative performance and report on the actual iterations required until solution. All solvers are iterated until the norms of the residuals have been reduced by ten digits, and the iterative refinement schemes perform the high precision update step every two digits.

6.2. Emulation vs. CPU: accuracy

We first confirm that the CG solver delivers exactly the same accuracy as the CPU reference solver. Each level of refinement translates to the expected error reduction by a factor of four. The multigrid solver, however, cannot deliver the required precision and stalls for large problem sizes. Recall that during the multigrid cycles, the coarse grid correction term is computed, prolongated and then accumulated to the defect on finer levels. A detailed analysis of our implementation and the test problem reveals that at some point during the multigrid cycles (when the norm of the defect has dropped below 1e–12 (absolute values) already), very small corrections are added to a comparatively large (with respect to absolute values) intermediate approximation of the solution. The emulated floating point format is not able to capture these large differences properly, and the solution process stalls. We refer to the appendix for tables A1–A3 with the exact numbers.

6.3. Emulation vs. CPU: performance

Figure 1 shows the absolute performance of the three solvers implemented in double precision on the CPU and the emulated double–single format on the GPU. The values are normalised to show the time until solution per unknown.

In view of the dramatic increase in iterations that the CG solver performs in the iterative refinement context (cf. Sections 5.4 and 6.5), we first observe that the number of iterations is identical to the double precision reference solvers for both multigrid and CG. Due to the

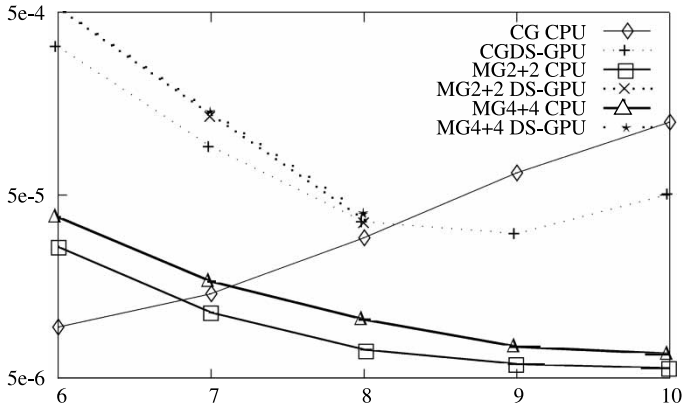

Figure 1. Normalised time until solution for the emulated double–single format (DS) with three different solvers (CG and MG with different number of smoothing steps) on the CPU and the GPU, for levels 6–10 (cf. Section 6.3).

overhead associated with executing code on the GPU and transferring initial data and the result vector, we see a significant slowdown compared to the CPU for small problem sizes. However, for the two largest problem sizes (levels 9 and 10, respectively), we note a speedup of more than a factor of two. Given the fact that for a multiplication, the operation count is increased by a factor of 32 (cf. Section 2.2), this speedup shows that the raw compute performance of the GPU is remarkable, since the doubled memory requirements are easily hidden because of its high arithmetic intensity. We again refer to the appendix for tables A1–A3 with the exact numbers.

6.4. Iterative refinement vs. CPU: accuracy

We confirm that all variants of the solvers deliver exactly the same accuracy as the CPU reference solver. Since the multigrid solvers are only demanded to reduce local defects by two digits in each outer iteration, they converge smoothly and do not stall as we experienced in the emulated double–single implementation. Refer to the appendix for tables A4–A6 with the exact numbers.

6.5. Iterative refinement vs. CPU: performance

Timings for the iterative refinement solvers are shown in figure 2. The values are normalised and show time until solution per unknown. As before, the exact numbers are tabulated in the appendix (tables A4–A6).

Compared to the reference implementation on the CPU and especially to the double–single emulated precision results from the previous subsection, we observe a promising speedup of both the CG and the multigrid implementation. As expected, the CPU outperforms the GPU for small problem sizes. Although multigrid executing F-cycles is not particularly suited for the GPU since the majority of work is performed on small levels which makes it hard to saturate the parallel PEs, we observe a notable factor of more than four in speedup, with only a small negligible increase in cycle count.

On the other hand, the CG solver performs on average 3.3 times more iterations than the reference implementation (cf. Section 5.4) and still executes more than four times faster than the CPU.

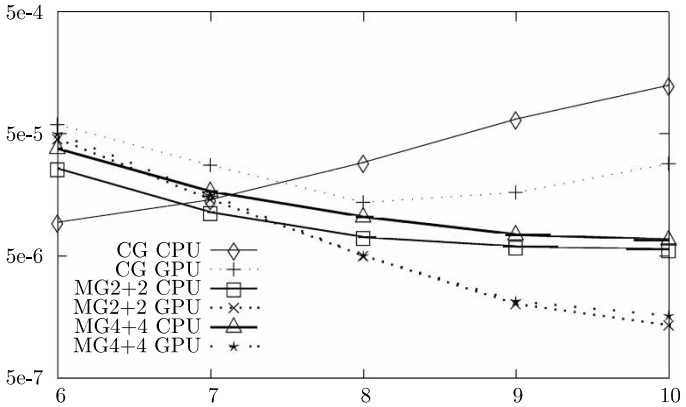

Figure 2. Normalised time until solution for the mixed precision iterative refinement scheme with three different solvers on the CPU and the GPU, for levels 6–10 (cf. Section 6.5).

Looking at the multigrid timings in more detail, we observe that the speedup factor increases much faster when more smoothing steps are performed. In summary, we conclude that in order to adapt the multigrid to the GPU in a better way, the solver might have to be reformulated to increase smoothing and interpret a medium level as the final coarse grid instead of restricting all the way to level 1 where execution is highly inefficient as not all pipelines are saturated. The trade-off might be slightly worse convergence rates per cycle, but this should not pose too much of a problem since we are interested in the overall time to solution.

Another aspect is the utilisation of the devices. Currently, the CPU is effectively idle when the GPU computes (except minor work orchestrating the GPU) and also the other way round. This is an artefact of our testing procedure where we deliberately solve only one linear system at a time. In the domain-decomposition context of FEAST (cf. Section 8.1) we have implemented a streaming version of solving linear systems associated with the refinement of several macro elements. Thus, we concurrently compute on the GPU for one system, update on the CPU for another, and use DMA transfers for a third one. In this way, we achieve a much better utilisation of the resources than solving all systems sequentially [60].

7. Resource savings on FPGAs

On the FPGA the benefits from using low-precision formats translate directly into resource savings. We summarise our results for a CG core synthesised with different precisions. The *pipelined* CG version that we use is based on ideas from parallel computing, where vector–vector and reduction operations are grouped together to reduce the deteriorating effects of global communication. The authors have evaluated this and another CG version under various parameters on the FPGA [58]. Here we extend the discussion of the resource consumption.

7.1. Iterative refinement with the FPGA

The FPGA is configured with a low precision pipelined CG core. In contrast to hardwired architectures, we can vary the precision bitwise, e.g. reducing the inner precision below the s23e8 single float format. Then the expensive inner solver can be performed with low precision components in parallel on the FPGA, while the few high precision operations of the outer solver could run on a small micro-processor on the FPGA board or the CPU of the host. As the iterative refinement technique is very flexible in the choice of the number of iterations in the inner solver, it may for example be derived from the performance ratio of the FPGA and the micro-processor/CPU and the bandwidth between them. The outer solver can in principle also be implemented on the FPGA, but since it is executed only infrequently and requires a high precision matrix-vector multiplication it is doubtful if these resources were wisely spent; after all the idea of mixed precision methods is to replace the expensive cores with many parallel simple ones. One possible approach would be to configure dual-mode arithmetic units which could be used as either one high precision or many low precision adders/multipliers.

We list the area consumption and maximum frequency of the pipelined CG core under different precisions (table 6). The core includes all vector–vector and reduction operations but no further combinations of the scalar values nor anything from the outer solver. The resource numbers are obtained with the Xilinx 7.1 ISE optimising for speed.

Table 6. Estimated area consumption and maximum frequency of the pipelined CG core on the Xilinx xc2v8000ff1517-5 FPGA, cf. Section 7.

Precision	*Slices*	*MUL18 × 18s*	*IOBs*	*Frequency (MHz)*
PipeLUT multiplier				
s17e11	4517	0	498	71
s20e11	5102	0	549	67
s23e11	5812	0	600	66
s28e11	6976	0	685	64
s36e11	9391	0	821	53
s44e11	11458	0	957	51
s52e11	14545	0	1093	49
MUL18 × 18 multiplier				
s17e11	4449	12	498	134
s20e11	4902	48	549	87
s23e11	5412	57	600	85
s28e11	6257	60	685	82
s36e11	7611	60	821	65
s44e11	8946	135	957	61
s52e11	10271	135	1093	59

For a clearer comparison we exclude the much smaller savings resulting from smaller exponents and therefore use 11-bit exponents in all the formats. For the double precision core we need a lot of resources in terms of slices and input/output blocks (IOB) and therefore the large Xilinx xc2v8000 (46,592 slices, 1108 IOBs) is used for all designs — otherwise we would have to use different chips for different precisions, which would make the numbers less comparable. In addition, the Xilinx xc2v8000 contains 168 hardwired 18×18-bit multipliers (MUL18 × 18), that are inserted in increasing numbers into new FPGAs to save configurable logic elements (cf. Section 4.3). We generate results with automatic selection of the MUL18 × 18 multipliers and with purely logic based pipelined LUT multipliers.

If the entire core is implemented in logic the area consumption is quadratic in the size of the number format as expected (table 6). Utilisation of the hardwired multipliers reduces the growth to linear, but then the multipliers themselves are consumed rapidly. Moreover, we see a significant drop in the frequency and very high IO requirements for the high precision core.

7.2. Analysis of results

Figures 3a–d visualise the data from table 6 and additional associated information. If the multipliers are configured with logic resources then figure 3a clearly shows the resulting quadratic area consumption. The quadratic coefficient of the growth of the number of slices is smaller than for the number of four input LUTs because the number of flip flops grows only linearly. In figure 3c, we see the effect of the automatic utilisation of the block MUL18 × 18 multipliers on the area consumption. The growth of the number of slices is reduced to linear at the cost of many MUL18 × 18 multipliers. The graph of the number of MUL18 × 18s is less smooth because of granularity effects, e.g. even where a 12-bit multiplier would suffice a MUL18 × 18 is consumed. This explains the comparably low number of MUL18 × 18s for the s36e11 format, where $36 = 2 \cdot 18$ splits very well under the given granularity and allows optimal utilisation of the hardwired resources.

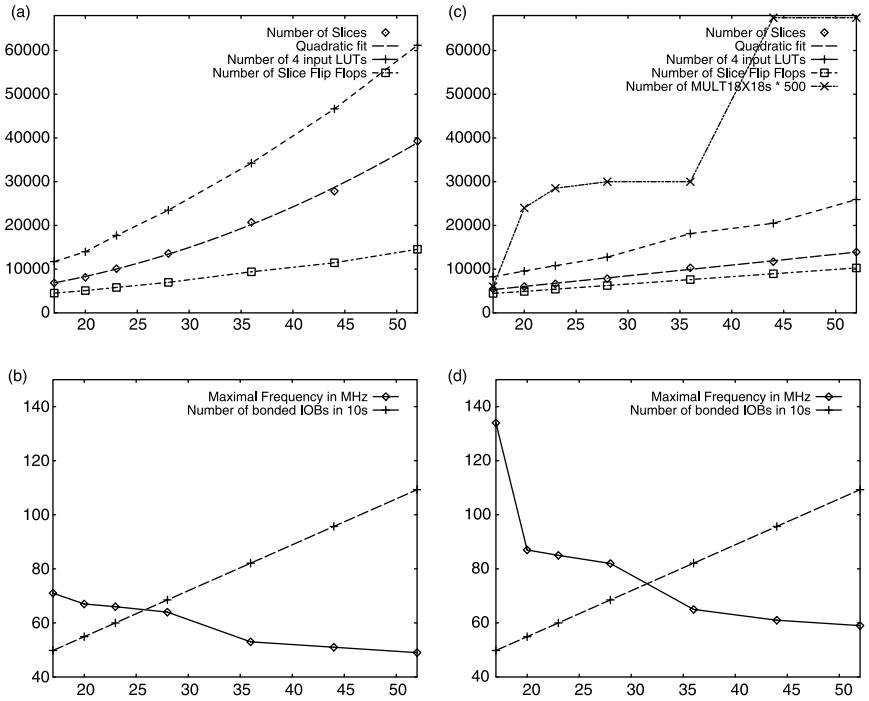

Figure 3. Resource consumption and frequency of the pipelined CG core on the Xilinx xc2v8000ff1517-5 FPGA, on the left with PipeLUT, on the right with MUL18 × 18 multipliers (cf. table 6).

Figure 3b and d show the IO requirements and estimated frequencies. In the logic-based design the frequency drops fairly smoothly, while the MUL18 × 18 design reveals a more volatile behaviour, that can again be attributed to the granularity of the hardwired multipliers. In any case the smaller cores are clearly in advantage. The number of required user IO obviously grows linearly, but the overall costs for production of a board connecting the FPGA with such wide buses to on-board memory supersede the linear growth. Therefore, the reduction from the 1093 IO blocks necessary for a double precision core to the 600 of a s23e11 core are a greater relief than the ratio of the number suggests.

Similar reasoning also applies to the required area, as prices of high-end devices do not scale linearly with their capacity. So the overall cost savings scale superlinearly on top of the quadratic area savings from the design. One possibility that opens up with the smaller low precision cores is to implement them on small FPGAs and employ several of them in parallel on the board rather than using an equivalently large single FPGA.

As FPGAs are available in various sizes, it turns out that the above resource savings quickly translate into concrete economic advantages. A s23e8 single precision pipelined CG core fits into a xc2v1500 (7680 slices), whereas a double precision core requires a xc2v4000 (23,040 slices). Given the additional frequency gains and the exact slowdown [58] in the convergence process of the mixed precision CG solver, the much smaller chip would be approximately 1.25 slower in obtaining the final result (plus the costs of the outer solver). If this is acceptable then the economic impact would be huge as the

xc2v1500 costs only one third of the xc2v4000. Moreover, in an integrated solution other components could also be downsized as the smaller chip consumes less energy and has fewer pins. If we are willing to sacrifice some of the resource savings for speedup, we can simply use a larger device than the xc2v1500 and implement several of the low precision cores in parallel. Effectively, the mixed precision iterative refinement method on the FPGA allows a continuous design space exploration between smallest area consumption and highest parallelism/speedup.

8. Mixed precision schemes on anisotropic grids

8.1. Goals and test design

As shown in Section 6, the proposed adaption of iterative refinement techniques to fast, parallel low-precision co-processors yields promising speedups compared to a CPU implementation without sacrificing any accuracy in the final result. In this section, we want to expand this work and analyse the numerical behaviour of the iterative refinement scheme with very ill-conditioned matrices. In theory the iterative refinement scheme converges as long as the matrices are "not too ill-conditioned" (cf. Section 3), and we want to quantify this in the practical FEM context.

Most publications discussing mixed precision iterative refinement techniques utilise carefully constructed specialised test cases to examine the impact of insufficient precision. For instance, defining problems based on the Hilbert matrix is a common technique to directly influence the condition number of the system matrix. In the FEM context and the implied matrix structure our work aims at, such test cases are not feasible. We propose to introduce *anisotropies* instead.

Since the "simple" Jacobi smoother is not robust enough to cope with anisotropies, we decided to use a powerful multigrid solver with sophisticated alternating linewise smoother (**MG_ADITRIGS**, [61]), capable of dealing with almost arbitrarily deformed meshes. This solver is very robust, and we have to apply neither damping nor more than two pre- and post-smoothing steps. Again, we employ an F-cycle. We did not implement the advanced ADI-TRIGS multigrid solver on fast co-processor hardware yet, because this is a non-trivial task due to the complex data flow and data dependencies. Instead, all numerical tests in this section (i.e. all three solvers) are performed with a simulation tool we implemented on top of FEAST, a powerful (parallel) hardware-oriented PDE solver toolkit, actively under development at Dortmund University. The results from this section, however, allow us to analyse the applicability of the proposed schemes for real-world situations and we will work towards adapting more powerful solver schemes to the co-processor hardware.

The core approach of FEAST is to exploit local structures with fast but simple numerical components while hiding anisotropies and globally-unstructured parts of the discretisation through a hierarchical domain-decomposition approach called SCARC [62]. The ultimate goal of our work is to include the co-processors as evaluated in this paper into FEAST, taking advantage of its domain decomposition approach to offload the compute-intensive smoothing parts of the solution process. For more details about FEAST, we refer to Turek *et al.* [63]. Figure 4 depicts an example of various fine-grain mesh details embedded in a global domain decomposition.

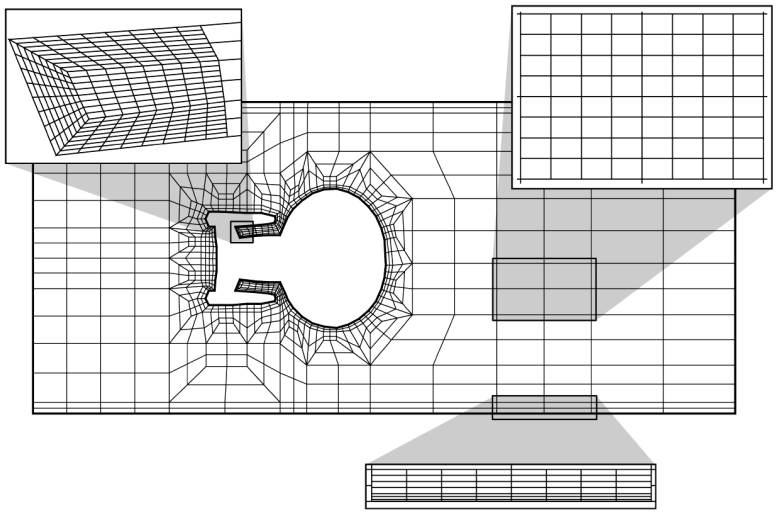

Figure 4. Illustration of various degrees of anisotropy in a complex CFD simulation using FEAST.

Apart from using different solvers, the test procedure remains identical to the one described in Section 5.1.

8.2. Anisotropies

One concern with mixed precision methods is the achieved accuracy in case of matrices with strongly varying coefficients. Rather than analysing artificial cases with randomly varying coefficients we utilise test matrices arising from anisotropic mesh refinements as they typically appear in FEM applications (cf. Figure 4):

8.2.1. Uniform anisotropies. Using rectangular instead of square elements allows us to specify the aspect ratio of each element in the refined discretisation. We refine the mesh uniformly (cf. figure 5) which leads to the same degree of anisotropy in each entry of the matrix.

8.2.2. Anisotropic refinement. In this strategy, we start with the unit square and in each refinement step, we subdivide the bottommost and rightmost layer of cells anisotropically (cf. figure 5). This refinement scheme is often used to accurately resolve boundary layers in real simulations. For each refined cell in these layers, the new midpoint x_c is calculated by recursively applying the formula $x_c = x_l + \nu((x_r - x_l)/2)$ with a given *anisotropy factor* ν

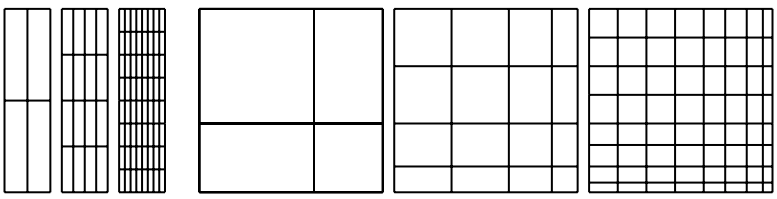

Figure 5. Example for different mesh anisotropies on levels 1–3. Left: Uniform subdivision of an anisotropic coarse mesh. Right: anisotropic subdivision of the rightmost and bottommost element layer in each refinement step.

($\nu = 1.0$ yields uniform refinement) and x_l and x_r denoting the coordinates of the left and right edge of a cell before subdividing (analogously for the y-component). All other cells are refined uniformly. This leads to matrices with locally condensed anisotropies instead of uniform ones. We refer to Kilian for a comprehensive overview of anisotropies in the FEM context [62].

We note that the above cases generate matrices which also occur when discretising anisotropic operators. For example, operators of the type $-\mathrm{div}(\mathbf{G}\nabla\mathbf{u})$ for a diagonal matrix $\mathbf{G}$ lead to the same matrix entries as introducing a corresponding degree of anisotropy directly on the coarse mesh level. Consequently, our test results also apply to these problems of operator anisotropy.

For each test case, we list the maximum aspect ratio $\mathrm{AR}_{\max}$ and the length of the smallest edge $h_{\min}$ in the maximum refinement level as indicators of the condition of the arising systems.

8.3. Reference values

In this section, we present the reference values of the three solver configurations executing in native double precision on several test cases, incorporating both uniform anisotropies as well as anisotropic refinement.

Table 7 lists the number of iterations or multigrid F-cycles (#Iter), the rate of convergence (ρ) and the L_2 error against the analytically known reference solution for the following test cases:

A. No anisotropies: $\Omega = [0,1]^2$, uniform refinement, $\mathrm{AR}_{\max} = 1.0$, $h_{\min} = 1.93{\cdot}10^{-1}$.
B. Weak uniform anisotropy: $\Omega = [0, 0.25] \times [0,1]$, uniform refinement, $\mathrm{AR}_{\max} = 4.0$ $h_{\min} = 4.88{\cdot}10^{-4}$.
C. Medium uniform anisotropy: $\Omega = [0, 0.0625] \times [0,1]$, uniform refinement, $\mathrm{AR}_{\max} = 16.0$, $h_{\min} = 1.22{\cdot}10^{-4}$. While this system can be represented well in single precision (by converting it from a double precision format), a direct assembly of the matrix in single precision fails on higher levels due to Jacobian determinants evaluating to zero.
D. Very high uniform anisotropy: $\Omega = [0, 10^{-11}] \times [0,1]$, uniform refinement, $\mathrm{AR}_{\max} = 1.0{\cdot}10^{11}$, $h_{\min} = 1.95{\cdot}10^{-14}$. This test case is complicated further by the fact that the solution is zero almost everywhere and the system can barely be assembled even in double precision.

   We tabulate only levels 8 and 9 ($N = 257^2$ and $N = 513^2$ unknowns) to improve clarity of the presentation.

As shown in table 7, all solvers perform as expected. The CG solver doubles the number of iterations in each refinement, and the two multigrid solvers converge independently of the level. The results for the multigrid with Jacobi smoother require some further explanation. Even for weak anisotropies (test case B), the Jacobi smoother necessitates the impractical amount of $64 + 64$ pre- and post-smoothing steps despite strong damping. In test case B, level 9 for instance, the number of necessary cycles approximately doubles if the number of smoothing steps is cut in half: $4 + 4$ results in 121 cycles ($\rho = 0.83$), $8 + 8$ in 61 cycles ($\rho = 0.68$), $16 + 16$ in 31 cycles ($\rho = 0.47$) and $32 + 32$ in 16 cycles ($\rho = 0.23$).

Table 8 summarises the convergence behaviour in the case of anisotropically-refined coarse meshes, again for levels 8 and 9. Five test cases are evaluated:

E. Weak anisotropic refinement: $\nu = 0.75$, $\mathrm{AR}_{\max} = 16.6$, $h_{\min} = 1.47{\cdot}10^{-4}$.
F. Medium anisotropic refinement: $\nu = 0.5$, $\mathrm{AR}_{\max} = 7.68{\cdot}10^{2}$, $h_{\min} = 3.81{\cdot}10^{-6}$. We expect the multigrid with Jacobi smoother to exhibit massive problems.
G. Hard anisotropic refinement: $\nu = 0.25$, $\mathrm{AR}_{\max} = 4.59{\cdot}10^{5}$, $h_{\min} = 7.45{\cdot}10^{-9}$. This test case yields a problem which cannot be assembled in single precision.
H. Hard anisotropic refinement: $\nu = 0.0625$, $\mathrm{AR}_{\max} = 1.33{\cdot}10^{11}$, $h_{\min} = 2.84{\cdot}10^{-14}$. This test case also yields a problem which cannot be assembled in single precision.
I. Extremely hard anisotropic refinement: $\nu = 0.03125$, $\mathrm{AR}_{\max} = 2.16{\cdot}10^{12}$, $h_{\min} = 3.55{\cdot}10^{-15}$. This test case yields a problem which cannot be assembled in double precision for levels greater than 8. This test is therefore extremely challenging for our mixed precision solver.

Table 7. Convergence history for three solvers: Uniform anisotropies (cf. Section 8.3).

		CG		*MG_JAC*		*MG_ADITRIGS*		
Test	*Level*	*#Iter*	ρ	*#Iter*	ρ	*#Iter*	ρ	*Error*
A	8	342	0.9349	8	0.0604	6	0.0135	5.7816E-7
A	9	676	0.9665	8	0.0606	6	0.0132	1.4454E-7
B	8	859	0.9735	8	0.0468	8	0.0413	1.7652E-8
B	9	1731	0.9868	8	0.0529	7	0.0351	4.4131E-9
C	8	1568	0.9854	31	0.4715	8	0.0541	5.4048E-10
C	9	3198	0.9928	31	0.4688	8	0.0519	1.3512E-10
D	8	1570	0.9854	n/a	n/a	6	0.0182	1.7387E-34
D	9	2810	0.9918	n/a	n/a	6	0.0154	4.3450E-35

Table 8. Convergence history for three solvers: Anisotropic refinement (cf. Section 8.3).

		CG		*MG_JAC*		*MG_ADITRIGS*		
Test	*Level*	*#Iter*	ρ	*#Iter*	ρ	*#Iter*	ρ	*Error*
E	8	804	0.9718	5	0.0046	6	0.0199	7.5197E-7
E	9	1647	0.9861	5	0.0053	6	0.0210	1.8799E-7
F	8	829	0.9726	6	0.0150	7	0.0361	1.1224E-6
F	9	1683	0.9864	10	0.0939	7	0.0372	2.8059E-7
G	8	901	0.9746	29	0.4457	8	0.0428	1.6354E-6
G	9	1828	0.9875	63	0.6930	8	0.0433	4.0886E-7
H	8	1079	0.9787	n/a	n/a	7	0.0372	2.1218E-6
H	9	2224	0.9897	n/a	n/a	8	0.0463	5.3045E-7
I	8	1129	0.9797	n/a	n/a	9	0.0705	8.8600E-6
I	9	n/a	n/a	n/a	n/a	n/a	n/a	n/a

We observe that the ADI-TRIGS smoother can handle all degrees of anisotropy without a substantial increase in the number of required iterations. The CG solver performs surprisingly well. Multigrid with Jacobi smoothing again fails to work properly even for moderate anisotropies, test cases E–G required $32 + 32$, $128 + 128$ and $128 + 128$ pre- and post-smoothing steps, respectively.

8.4. Performance of the mixed precision schemes with anisotropies

In this test series, we evaluate the mixed precision solver. The main target of our experiments is accuracy, so we are especially interested in the hard cases C, D and H, I of the previous test series. In table 9, we summarise the performance and the errors compared to the reference solution of the mixed precision algorithm for all previous test cases (cf. tables 7 and 8), using ADI-TRIGS–multigrid as inner solver. We note that the achieved accuracy is independent of the choice of the inner solver, only the convergence behaviour differs, moreover we need to apply the robust ADI-TRIGS solver to treat the anisotropies properly. In this test series, the inner solution process is terminated once two digits are gained, inspired by the fact that the multigrid solver with ADI-TRIGS smoother generally gains more than one digit per cycle. As a scaling heuristics, we normalise the defect with its l_2-norm in each outer iteration.

Most importantly, we note that the mixed precision solver is able to deliver exactly the same results regarding accuracy as a direct solution in double precision, even for problems that cannot be assembled in single precision. The convergence behaviour of the mixed precision iteration is remarkably good compared to the direct iteration using multigrid with ADI-TRIGS smoothing. In most test cases, the iteration counts are identical, occasionally the mixed precision approach performs one inner iteration more or less than the direct solver. We conclude that the only additional numerical work required by our approach is the calculation of few defects in the outer loop. Even for very hard problems, the mixed precision algorithm does not get stalled by the lack of precision in the inner solver.

Table 9. Convergence history and accuracy comparison for the mixed precision solver using multigrid with ADI-TRIGS as smoother on levels 8 and 9 (cf. Section 8.4). Columns labelled "$\text{Error}_{\text{ref}}$" reproduce the values from tables 7 and 8 for reference. Notation: Colon separates number of outer from sum of all inner iterations.

	Level 8			*Level 9*		
Test	*#Iter*	$Error_{mix}$	$Error_{ref}$	*#Iter*	$Error_{mix}$	$Error_{ref}$
A	4:6	5.7816E-7	5.7816E-7	4:6	1.4454E-7	1.4454E-7
B	5:8	1.7652E-8	1.7652E-8	5:8	4.4131E-9	4.4131E-9
C	5:8	5.4048E-10	5.4048E-10	5:8	1.3512E-10	1.3512E-10
D	4:5	1.7387E-34	1.7387E-34	4:5	4.3450E-35	4.3450E-35
E	4:6	7.5197E-7	7.5197E-7	4:6	1.8799E-7	1.8799E-7
F	4:7	1.1224E-6	1.1224E-6	4:7	2.8059E-7	2.8059E-7
G	4:7	1.6354E-6	1.6354E-6	4:7	4.0886E-7	4.0886E-7
H	5:8	2.1218E-6	2.1218E-6	5:8	5.3045E-7	5.3045E-7
I	5:10	2.2149E-6	8.8600E-6	n/a	n/a	n/a

9. Summary and future work

We have presented fast algorithms for the solution of large linear equation systems as they typically arise in finite element discretisations. Our analysis is hardware-motivated, we focus on techniques that can exploit the enormous compute power of single precision parallel devices to achieve highly accurate results.

Based on a common PDE example, we show that low precision in the computation alone can lead to highly inaccurate solutions, although the convergence behaviour of the solver does not hint at this at all. Knowing about the error propagation during the solution process, however, enables us to concentrate the high precision to the few places where it is necessary, instead of using high precision throughout the algorithm.

The combined double–single precision format can be used to emulate higher precision on hardware which only supports native low precision, but the emulation is very expensive in the number of operations. The mixed precision iterative refinement technique makes it feasible to offload up to 99% of the arithmetic work to a fast low-precision co-processor without sacrificing the accuracy of the final result. We have discussed several hardware combinations that are particularly suitable for such techniques.

We have demonstrated the applicability of these approaches on two particular hardware combinations: Using the GPU as a fast, parallel co-processor for the general purpose CPU, we have shown that the emulation technique is significantly slower for our PDE problem than the iterative refinement technique. On FPGAs, we have highlighted the resource savings that can be obtained by changing the core of the solver to a lower precision and executing the high precision corrections on a different resource, e.g. CPU, micro-controller.

Our numerical tests are based on a conjugate gradient and a multigrid solver with Jacobi smoothing. Since the Jacobi smoother is not suitable to solve systems arising from the discretisation of realistic, complex domains, we have evaluated the stability of the iterative refinement techniques for ill-conditioned matrices using a simulation tool that provides stronger smoother. Instead of using matrices with randomly varying coefficients, we introduce strong anisotropies into the discretisation to mimic ill-conditioned systems while keeping the practical applicability in mind. Our results confirm that the techniques are very robust up to impractical degrees of anisotropy.

In the future, we will work towards adding more robust solvers to our GPU toolkit to make more realistic simulation domains feasible. We will improve our currently prototypical realisation of incorporating the parallel devices as co-processors into the finite element package FEAST. Finally, we want to proceed with the development of the iterative refinement procedure towards other problems, e.g. for generalised convergent iterative schemes [64].

Acknowledgements

We would like to thank the FEAST developers, Christian Becker, Sven Buijssen, Matthias Grajewski, Susanne Kilian and Hilmar Wobker for their patience and invaluable support. The ATI and NVIDIA developer support helped us to sort out issues with the rounding modes in the emulated double–single floating point formats. This research has been partly supported by a Max Planck Center for Visual Computing and Communication fellowship and by the German Science Foundation (DFG), project TU102/22-1.

References

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

[2] Ho, C.H., Leong, P., Luk, W., Wilton, S. and Lopez-Buedo, S., 2006, Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*.

[3] Dekker, T.J., 1971, A floating-point technique for extending the available precision, *Numerische Mathematik*, **18**, 224–242.

[4] Knuth, D.E., 1997, *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms* (Boston, MA: Addison-Wesley Longman Publishing Co., Inc.).

[5] Møller, O., 1965, Quasi double-precision in floating point addition, *BIT*, **5**(1), 37–50.

[6] Hida, Y., Li, X.S. and Bailey, D.H., 2001, Algorithms for quad-double precision floating point arithmetic. In: N. Burgess and L. Ciminiera (Eds.) *Proceedings of the 15th Symposium on Computer Arithmetic*, pp. 155–162.

[7] 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.
[8] Priest, D.M., 1991, Algorithms for arbitrary precision floating point arithmetic. *10th IEEE Symposium on Computer Arithmetic*, pp. 132–143.
[9] Shewchuk, J.R., 1997, Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Discrete & Computational Geometry*, October **18**(3), 305–363.
[10] Bailey, D.H., Hida, Y., Jeyabalan, K., Li, X.S. and Thompson, B., 2006, High-precision software directory, http://crd.lbl.gov/∼dhbailey/mpdist/
[11] Free Software Foundation, Inc., *GNU Multiple Precision Arithmetic Library*, 4.2.1 edition, 2006. http://www.swox.com/gmp.
[12] Wilkinson, J.H., 1963, *Rounding Errors in Algebraic Processes* (New York, NY: Dover Publications, Incorporated).
[13] Martin, R.S., Peters, G. and Wilkinson, J.H., 1966, Handbook series linear algebra: iterative refinement of the solution of a positive definite system of equations, *Numerische Mathematik*, **8**, 203–216.
[14] Bowdler, H.J., Martin, R.S., Peters, G. and Wilkinson, J.H., 1966, Handbook series linear algebra: solution of real and complex systems of linear equations, *Numerische Mathematik*, **8**, 217–234.
[15] Demmel, J., Hida, Y., Kahan, W., Li, X.S., Mukherjeek, S. and Riedy, E.J., 2006, Error bounds from extra precise iterative refinement, *ACM Transactions on Mathematical Software*, June **32**(2), 325–351.
[16] Zielke, G. and Drygalla, V., 2003, Genaue Lösung linearer Gleichungssysteme, *GAMM-Mitteilungen*, **2**(1), 7–107.
[17] Turner, K. and Walker, H.F., 1992, Efficient high accuracy solutions with GMRES(*m*), *SIAM Journal on Scientific and Statistical Computing archive*, **13**(3), 815–825.
[18] Geddes, K.O. and Zheng, W.W., 2003, Exploiting fast hardware floating point in high precision computation. *ISSAC'03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation* (New York, NY: ACM Press), pp. 111–118.
[19] Langou, J., Langou, J., Luszczek, P., Kurzak, J., Buttari, A. and Dongarra, J.J., 2006, Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). *Proceedings of the ACM/IEEE SuperComputing 2006 (SC'06)*, to appear.
[20] Stewart, G.W., 1973, *Introduction to Matrix Computations* (San Diego: Academic Press).
[21] Higham, N.J., 2002, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. (Philadelphia, PA: Society for Industrial and Applied Mathematics).
[22] Hartenstein, R., 2001, A decade of reconfigurable computing: a visionary retrospective, *Design, Automation and Test in Europe—DATE*, March 2001.
[23] Hartenstein, R., 2003, Data-stream-based computing: models and architectural resources. *International Conference on Microelectronics, Devices and Materials (MIDEM 2003)*, Ocotober.
[24] 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. *ISCA 2003*, pp. 422–433.
[25] Guo, Z., Najjar, W., Vahid, F. and Vissers, K., 2004, A quantitative analysis of the speedup factors of FPGAs over processors. *ACM/IEEE International Symposium on Field-Programmable Gate Arrays*.
[26] Taylor, M.B., Kim, J.S., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J., 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.
[27] Suh, J., Kim, E., 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: D. DeGroot (Ed.) *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA-03), Volume 31 of Computer Architecture News* June (New York, NY: ACM Press), pp. 410–421.
[28] Strzodka, R., 2004, *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University of Duisburg-Essen.
[29] Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J. and Hanrahan, P., 2006, Sequoia: programming the memory hierarchy. *Proceedings of the ACM/IEEE SuperComputing 2006 (SC'06)*, to appear.
[30] Clearspeed, CSX600. www.clearspeed.com/downloads/CSX600Processor.pdf, 2006.
[31] IBM Sony, Toshiba. Cell BE. http://www.ibm.com/developerworks/power/cell.
[32] Mercury. Cell BE. http://www.mc.com/cell/.
[33] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P. and Yelick, K., 2006, The potential of the cell processor for scientific computing. *CF '06: Proceedings of the 3rd Conference on Computing Frontiers* (New York, NY: ACM Press), pp. 9–20.
[34] AGEIA. PhysX. http://www.ageia.com/products/physx.html.
[35] Göddeke, D. and Strzodka, R., 2006, Scientific computing on graphics hardware, tutorial at the 6th International Conference on Computational Science (ICCS 2006).
[36] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M.J., Krüger, J., Lefohn, A.E. and Purcell, T., 2005, A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, pp. 21–51.
[37] GPGPU. General-purpose computation using graphics hardware, http://www.gpgpu.org.
[38] 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.

[39] Hillesland, K. and Lastra, A., 2004, GPU floating-point paranoia. *Proceedings of GP2*.
[40] Daumas, M., Da Graça, G. and Defour, D., 2006, Caractéristiques arithmétiques des processeurs graphiques. *Symposium en Architecture de Machines*.
[41] Da Graça, G. and Defour, D., 2006, Implementation of float–float operators on graphics hardware. *7th Conference on Real Numbers and Computers, RNC7*, pp. 23–32.
[42] Hitz, M.A. and Payne, B.R., 2006, Implementation of residue number systems on GPUs. *ACM SIGGRAPH Conference Abstracts and Applications*.
[43] Thall, A., 2006, Extended-precision floating-point numbers for GPU computation. *ACM SIGGRAPH Conference Abstracts and Applications*.
[44] Dale, K., Sheaffer, J.W., Kumar, V.V., Luebke, D.P., Humphreys, G. and Skadron, K., 2006, Applications of small-scale reconfigurability to graphics processors. *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC2006)* (Berlin: Springer).
[45] Belanovic, P. and Leeser, M., 2002, A library of parameterized floating-point modules and their use. *FPL'02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications* (London: Springer-Verlag), pp. 657–666.
[46] Fang, F., Chen, T. and Rutenbar, R., 2002, Lightweight floating-point arithmetic: case study of inverse discrete cosine transform, *EURASIP Journal on Signal Processing, Special Issue on Applied Implementation of DSP and Communication Systems*, 879–892.
[47] Gaffar, A.A., Mencer, O., Luk, W. and Cheung, P.Y.K., 2004, Unifying bit-width optimisation for fixed-point and floating-point designs. *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04)*, pp. 79–88.
[48] Liang, J., Tessier, R. and Mencer, O., 2003, Floating point unit generation and evaluation for FPGAs. *FCCM'03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (Washington, DC: IEEE Computer Society), p. 185.
[49] Dido, J., Geraudie, N., Loiseau, L., Payeur, O., Savaria, Y. and Poirier, D., 2002, A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. *FPGA'02: Proceedings of the 2002 ACM/SIGDA 10th International Symposium on Field Programmable Gate Arrays* (New York, NY: ACM Press), pp. 50–55.
[50] Govindu, G., Zhuo, L., Choi, S. and Prasanna, V., 2004, Analysis of high-performance floating-point arithmetic on FPGAs. *18th International Parallel and Distributed Processing Symposium (IPDPS04), Workshop 3*, p. 149b.
[51] Matousek, R., Tichy, M., Phol, Z., Kadlec, J., Softley, C. and Coleman, N., 2002, Logarithmic number systems and floating-point arithmetics on FPGA. *12th International Conference on Field Programmable Logic and Applications* (London: Springer-Verlag), pp. 627–636.
[52] Haselman, M., Beauchamp, M., Wood, A., Hauck, S., Underwood, K. and Hemmert, K.S., 2005, A comparison of floating point and logarithmic number systems on FPGAs. *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 181–190.
[53] Bondalapati, K. and Prasanna, V.K., 2002, Reconfigurable computing systems, *Proceedings of the IEEE*.
[54] Compton, K. and Hauck, S., 2002, Reconfigurable computing: a survey of systems and software, *ACM Computing Surveys*, **34**(2), 171–210.
[55] Turek, S., 1999, *Efficient Solvers for Incompressible Flow Problems: An Algorithmic and Computational Approach* (Berlin: Springer).
[56] Grajewski, M., Köster, M., Kilian, S. and Turek, S., 2005, Numerical analysis and practical aspects of a robust and efficient grid deformation method in the finite element context, Ergebnisberichte des Instituts für Angewandte Mathematik, Nr. 294, FB Mathematik, Universität Dortmund.
[57] Industrial Light & Magic, OpenEXR, implementation of the half data type.
[58] Strzodka, R. and Göddeke, D., 2006, Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006)*.
[59] Göddeke, D., Strzodka, R. and Turek, S., 2005, Accelerating double precision FEM simulations with GPUs. In: F. Hülsemann, M. Kowarschik and U. Rüde (Eds.) *18th Symposium Simulations Technique* volume Frontiers in Simulation, ASIM 2005 (Erlangen: SCS Publishing House e.V.), pp. 139–144.
[60] Göddeke, D., Becker, Ch. and Turek, S., 2006, Integrating GPUs as fast co-processors into the parallel FE package FEAST. In: M. Becker and H. Szczerbicka (Eds.) *Proceedings of the 19th Symposium on Simulation Technique*, pp. 277–282.
[61] Altieri, M., Becker, Ch. and Turek, S., 1999, On the realistic performance of linear algebra components in iterative solvers. In: H.-J. Bungartz, F. Durst and Chr. Zenger (Eds.) *High Performance Scientific and Engineering Computing: Proceedings of the International FORTWIHR Conference on HPSEC, volume 8 of Lecture Notes in Computational Science and Engineering* (Berlin: Springer), pp. 3–12.
[62] Kilian, S., 2001, *Ein verallgemeinertes Gebietszerlegungs-/Mehrgitterkonzept auf Parallelrechnern*. PhD thesis, Universität Dortmund.
[63] Becker, Ch., Kilian, S. and Turek, S., 2002, Hardware-oriented numerics and concepts for PDE software. *FUTURE 1095* (Amsterdam: International Conference on Computational Science ICCS2002 Elsevier), pp. 1–23.
[64] Strzodka, R. and Göddeke, D., 2006, Mixed precision methods for convergent iterative schemes. *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*.

Appendix A: Detailed GPU results

This appendix contains detailed tables with timing and performance results on the GPU (cf. Section 6).

Table A1. Timings and speedup factors for different problem sizes of a CG solver: CPU in native double precision, GPU in emulated double–single floating point format (cf. Section 6.3).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	42	0.02	2.607000747E-5	42	1.33	2.607000747E-5	0.02
6	85	0.04	6.613757931E-6	85	1.37	6.613757928E-6	0.03
7	171	0.24	1.666003669E-6	171	1.53	1.666003655E-6	0.16
8	342	1.93	4.181054493E-7	342	2.36	4.181054014E-7	0.82
9	676	17.34	1.047283078E-7	676	8.09	1.047281043E-7	2.14
10	1357	131.63	2.620418257E-8	1357	53.28	2.620376988E-8	2.47

Table A2. Timings and speedup factors for different problem sizes of a multigrid solver performing $2 + 2$ Jacobi smoothing steps: CPU in native double precision, GPU in emulated double–single floating point format (cf. Section 6.3).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	8	0.07	2.607000747E-5	8	2.16	2.607000747E-5	0.03
6	8	0.11	6.613757930E-6	8	2.21	6.613757960E-6	0.05
7	8	0.19	1.666003671E-6	8	2.28	1.666003788E-6	0.08
8	8	0.47	4.181054499E-7	8	2.41	4.181059363E-7	0.2
9	8	1.56	1.047283072E-7	max	max	6.734639345E-7	n/a
10	8	5.93	2.620418261E-8	max	max	3.573254856E-7	n/a

Table A3. Timings and speedup factors for different problem sizes of a multigrid solver performing $4 + 4$ Jacobi smoothing steps: CPU in native double precision, GPU in emulated double–single floating point format (cf. Section 6.3).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	7	0.1	2.607000747E-5	7	2.17	2.607000747E-5	0.05
6	7	0.16	6.613757934E-6	7	2.24	6.613757966E-6	0.07
7	7	0.28	1.666003670E-6	7	2.37	1.666003775E-6	0.12
8	7	0.69	4.181054493E-7	7	2.6	4.181059399E-7	0.27
9	6	1.95	1.047283071E-7	max	max	6.739087494E-7	n/a
10	6	7.07	2.620418265E-8	max	max	3.587387658E-7	n/a

Table A4. Timings and speedup factors for different problem sizes of a CG solver: CPU in native double precision, GPU executing the mixed precision iterative refinement strategy with support from the CPU (cf. Section 6.5).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	42	0.02	2.607000747E-5	5:99	0.17	2.607000756E-5	0.12
6	85	0.04	6.613757931E-6	5:190	0.25	6.613757931E-6	0.16
7	171	0.24	1.666003669E-6	5:412	0.46	1.666003870E-6	0.52
8	342	1.93	4.181054493E-7	5:861	0.91	4.181055132E-7	2.12
9	676	17.34	1.047283078E-7	6:2256	4.36	1.047281051E-7	3.98
10	1357	131.63	2.620418257E-8	6:4500	29.9	2.620347100E-8	4.40

Table A5. Timings and speedup factors for different problem sizes of a multigrid solver with $2 + 2$ Jacobi smoothing steps: CPU in native double precision, GPU executing the mixed precision iterative refinement strategy with support from the CPU (cf. Section 6.5).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	8	0.07	2.607000747E-5	5:10	0.16	2.607000747E-5	0.44
6	8	0.11	6.613757930E-6	5:9	0.19	6.613757928E-6	0.58
7	8	0.19	1.666003671E-6	5:9	0.24	1.666003655E-6	0.79
8	8	0.47	4.181054499E-7	5:9	0.33	4.181054014E-7	1.42
9	8	1.56	1.047283072E-7	5:9	0.53	1.047281043E-7	2.94
10	8	5.93	2.620418261E-8	5:9	1.42	2.620376988E-8	4.18

Table A6. Timings and speedup factors for different problem sizes of a multigrid solver with $4 + 4$ Jacobi smoothing steps: CPU in native double precision, GPU executing the mixed precision iterative refinement strategy with support from the CPU (cf. Section 6.5).

	CPU			*CPU*			
Level	*#Iter*	*Time*	*Error*	*#Iter*	*Time*	*Error*	*Speedup*
5	7	0.1	2.607000747E-5	4:7	0.17	2.607000747E-5	0.59
6	7	0.16	6.613757934E-6	4:7	0.21	6.613757928E-6	0.76
7	7	0.28	1.666003670E-6	4:7	0.26	1.666003655E-6	1.08
8	7	0.69	4.181054493E-7	4:7	0.33	4.181054014E-7	2.09
9	6	1.95	1.047283071E-7	4:7	0.56	1.047281043E-7	3.48
10	6	7.07	2.620418265E-8	5:8	1.69	2.620376988E-8	4.18