Pipelined Mixed Precision Algorithms on FPGAs for Fast and Accurate PDE Solvers from Low Precision Components

Robert Strzodka
Stanford University, Max Planck Center
Stanford, CA 94305, USA
strzodka@stanford.edu

Dominik Göddeke
University of Dortmund, Applied Mathematics
44227 Dortmund, Germany
dominik.goeddeke@math.uni-dortmund.de

Abstract

FPGAs are becoming more and more attractive for high precision scientific computations. One of the main problems in efficient resource utilization is the quadratically growing resource usage of multipliers depending on the operand size. Many research efforts have been devoted to the optimization of individual arithmetic and linear algebra operations. In this paper we take a higher level approach and seek to reduce the intermediate computational precision on the algorithmic level by optimizing the accuracy towards the final result of an algorithm. In our case this is the accurate solution of partial differential equations (PDEs). Using the Poisson Problem as a typical PDE example we show that most intermediate operations can be computed with floats or even smaller formats and only very few operations (e.g. 1%) must be performed in double precision to obtain the same accuracy as a full double precision solver. Thus the FPGA can be configured with many parallel float rather than few resource hungry double operations. To achieve this, we adapt the general concept of mixed precision iterative refinement methods to FPGAs and develop a fully pipelined version of the Conjugate Gradient solver. We combine this solver with different iterative refinement schemes and precision combinations to obtain resource efficient mappings of the pipelined algorithm core onto the FPGA.

1. Introduction

There is a trend towards high precision floating point computations on FPGAs. The size of FPGAs grows quickly and allows to implement many parallel arithmetic units even for large number formats like the IEEE double standard. Additionally, dedicated hardwired 18x18 multipliers in some chips allow to reduce the consumption of logic resources for arithmetic operations on large number formats. Keith Underwood analyzes the resulting floating point performance trends of FPGAs against CPUs [25] and concludes that the peak performance of FPGAs already surpasses that of CPUs and even where actual application performance does not, it shall soon due to the much higher performance increases per year. However, not winning over the CPU, but higher performance in general is the goal, as scientific and commercial applications demand ever increasing processing power.

Scientific computations also pose high requirements on the accuracy of the solution which often leads to implementations using the double float format. When implementing arithmetic floating point operations on FPGAs one of the most important facts is the quadratically growing area of a multiplier in operand size, as opposed to a linear growth for an adder. Hardwired embedded multipliers alleviate the problem of logic consumption to some extent, but then they themselves are consumed quadratically with growing operand size. So despite the savings this does not solve but rather shifts the problem.

1.1. Floating Point Numbers on FPGAs

While micro-processors operate on fixed-width formats and thus usually have dedicated single or double floating point units, FPGAs can save resources by adapting the number format to the application. Moreover, multiple tradeoffs between latency and area can be exploited which has already been extensively studied for floating point formats [6, 15, 19]. There exist also parameterized IP cores which offer particularly efficient implementations for a given architecture. Fully parameterizable floating point libraries [1, 10] allow extensive explorations of different precisions and methods for automatic optimization of the operand sizes have been proposed [11, 12]. Another option for FPGAs is to use logarithmic number systems which avoid the quadratic complexity of the multiplier but complicate the adder [16, 21].

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)
0-7695-2661-6/06 $20.00 © 2006 IEEE

Beside the optimizations on the arithmetic level the configurability of logic and data paths allows additional gains in actual applications. Floating point FIR filters have been analyzed in detail [26], the Fast-Fourier-Transform has received particular attention [7, 17], and Lienhart et al. perform an N-body simulation [20] with custom floating point numbers. In our context vector and matrix operations are of particular interest. Efficient implementations for the kernels of these operations (BLAS) [23] and both dense [9, 27] and sparse [3] matrices have been studied. All these techniques offer valuable approaches to minimize the resource usage and maximize the throughput of scientific computations. These optimizations explore the structural level by exchanging the order and placement of operations and the implementational level by balancing the available resources against area and time constraints.

In this paper we explore the next higher level of algorithmic optimization. By including the semantic knowledge about the actual goal of the algorithm, we may cut down the resource usage much more radically, e.g. utilizing less than half of the original precision in most places, and still obtain the same accuracy. This is possible because we exploit the theoretical and empirical knowledge about how errors propagate and contribute to the final result. We apply this algorithmic optimization to the class of problems concerning the discrete solution of partial differential equations (PDEs). We describe the theoretical setting and how starting with the PDE we arrive at the problem of solving a linear equation system. For the solution we employ a Conjugate Gradient (CG) iterative solver, which we modify into a pipelined CG for a better hardware implementation, similar to CG variants on parallel computers. Then we apply different algorithmic precision optimizations to the pipelined CG, inspired by a technique known as mixed precision *iterative refinement*.

1.2. Mixed Precision Iterative Refinement

Iterative refinement techniques have already been introduced in 1966 by Wilkinson et al. [2]. For thorough information on these methods we refer to Demmel et al. [5]. The core idea of iterative refinement is to distinguish between different types of iterations in the iterative solver. Normally the steps are equal, i.e. in step k the solver would take an input vector $\mathbf{v}_k$, perform some vector and matrix operations and then output the result vector $\mathbf{v}_{k+1}$ which is used as input in the next step. Now we split the solution process into a computationally demanding low precision inner iteration loop and a computationally simple high precision outer correction loop. The computationally expensive inner loop can be performed with low precision components in parallel on the FPGA, while the few high precision operations of the outer loop could run on a small micro-processor on the FPGA board. The approach is very flexible as it allows to choose how many iterations to make in the inner loop depending on the performance ratio of the FPGA and the micro-processor and the bandwidth between them. Moreover, different iterative solvers can be employed in the two loops. For numerical examination of multigrid solvers in this context and an implementation on graphics processors see [14]. Here we study the pipelined Conjugate Gradient solver with a focus on an efficient hardware implementation, as multigrid methods generate a complex data-flow which is much harder to map to FPGAs.

FPGAs are ideal candidates for mixed precision methods as they continuously gain from the reduction of the number format in the inner loop. However, operating mainly on floats rather than doubles also benefits CPUs. Li et al. discuss the extension of the popular BLAS library to mixed precision algorithms [18]. Turner and Walker [24] accelerate the solution of an elliptic PDE. Geddes and Zheng [13] solve different problems with a mixed precision Newton iteration. This paper, in particular, demonstrates that the algorithmic precision optimization can be successfully applied to various problems.

As application performance is often limited by bandwidth rather than computational resources due to the so-called *memory wall* problem, we should point out that the reduction of the operand size in the inner loop also reduces the bandwidth requirements. In linear algebra computations which have a very low computational intensity (ratio of operations per memory access) this is of particular importance. Cutting the bandwidth in half for such operations often delivers almost the theoretical factor of two in speedup due to the smaller memory transfers alone.

1.3. Paper Overview

Our main contribution is the adaptation of the iterative refinement techniques to FPGAs, which allow to exploit the parallel processing power of FPGAs with a multitude of low precision components, and reduce the need of costly high precision arithmetic to a minimum. We start our approach of algorithmic optimization with a continuous PDE problem and go through the discretization, quantization and implementation steps all the way down to the synthesis of the solver core onto the FPGA. Section 2 explains how we arrive from the PDE formulation at the iterative refinement method. Drawing on ideas from high performance parallel computing we then develop a pipelined version of the CG algorithm as the original cannot fully exploit the capabilities of configurable data paths on FPGAs (Section 3.1). Based on our results with the mixed precision pipelined CG (Section 3.2), we are inspired to design a different iterative refinement technique for the pipelined CG solver (Section 3.3). Then we present a framework for the testing of the

solver variants and precision options and the implementation of the corresponding functionality in VHDL (Section 4). We compare the different solvers concerning accuracy, inner and outer iteration count and area usage on the FPGA (Section 5). We conclude with a summary of the results and suggestions for future work (Section 6).

2. Iterative Refinement for PDE Problems

Iterative refinement methods are a general tool applicable to the solution of many differential equations. For the clarity of presentation we choose one very common PDE, the Poisson Problem, for the derivation of the technique and the detailed result comparison. The Poisson Problem is a typical example of an elliptic PDE, also often encountered in time discretized parabolic and hyperbolic problems. For instance, projection schemes in Navier-Stokes simulations require the solution of the Poisson Problem to obtain the pressure. Moreover, the solution of the Poisson Problem is often the most time consuming step in such simulations and therefore accelerations thereof are of great interest.

2.1. Poisson Problem

For a given function $\mathbf{b} : \Omega \to \mathbb{R}$ on a domain $\Omega \subseteq \mathbb{R}^d$, say a rectangle $\Omega := [0, X] \times [0, Y] \subseteq \mathbb{R}^2$, we want to solve the Poisson equation

$$-\Delta \mathbf{u} = \mathbf{b}$$

where $\Delta := \sum_{i=1}^{d} \frac{\partial^2}{\partial_{x_i}^2}$ is the Laplace operator and d the dimension. For simplicity we will fix the dimension to 2D, i.e. $d = 2$, but all the reasoning also applies to higher dimensions. For our example we prescribe zero Dirichlet boundary conditions, i.e. $\mathbf{u}$ must be zero on the boundary of the domain, and define the function on the right hand side as $\mathbf{b} := -\Delta \hat{\mathbf{u}}$ with $\hat{\mathbf{u}}(x, y) = x(X - x)y(Y - y)$.

For the solution process the domain is discretized with an equidistant grid. We refer to [14] for the analysis of anisotropic grids in the context of iterative refinement methods. The continuous function is represented on the grid with functions of small support centered around the grid nodes. We use bilinear functions which form the function space of Conforming Bilinear Finite Elements (Q_1). The rectangle Ω is divided into 2^n, e.g 2^8 equal parts both horizontally and vertically, yielding $N = (2^n + 1)^2$ nodes and thus Finite Element (FE) functions. The collection of all coefficients corresponding to these functions forms a vector of size N. In form of these vectors the continuous equation from above can be written as

$$A\mathbf{u} = \mathbf{b}$$

where $\mathbf{u}, \mathbf{b}$ are FE vectors and A is a symmetric 9-band matrix, i.e. only the main diagonal and eight sub-diagonals contain entries other than 0. Although A has a simple structure it is still difficult to solve the system because of the size of the matrix. For instance, a discretization *level* of 8 already gives us $66,049(8)$ vector elements, in 3D more than 16 million (we will indicate the level in parentheses following the number of unknowns).

For problems of this size it is impractical to solve the linear equation system by direct methods such as Gaussian elimination. Instead, most often iterative methods are applied. Thereby, we choose a start vector $\mathbf{u}_0$ and then iterate

$$\mathbf{u}_{k+1} = F(\mathbf{u}_k)$$

for $k = 0, \ldots, k_{\max}$, where $k_{\max}$ is either a given positive integer or the first one which satisfies

$$\|\mathbf{b} - A\mathbf{u}_{k_{\max}}\| \quad < \quad \varepsilon \|\mathbf{b} - A\mathbf{u}_0\|, \tag{1}$$

i.e. the relative error drops below a threshold ε.

There exist very efficient methods for solving such systems, so-called multigrid solvers. These, however, require an explicit hierarchy in the discretizing geometry or must construct such an equivalent hierarchy from the matrix itself. In any case the resulting data-flow is much more complex than for methods which operate always in the same manner on the entire vector. Among the simpler methods the Conjugate Gradient (CG) algorithm, applicable to positive definite matrices, is very popular as it often delivers superlinear convergence. In the setting of Finite Elements the positive definiteness is usually satisfied and we may exploit it by using the Conjugate Gradient method. The algorithm reads:

$$\begin{aligned}
\mathbf{q}_k &= A\mathbf{p}_k, \\
\alpha_k &= \frac{\rho_k}{\mathbf{p}_k \cdot \mathbf{q}_k}, \\
\mathbf{u}_{k+1} &= \mathbf{u}_k + \alpha_k \mathbf{p}_k, \\
\mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{q}_k, \\
\rho_{k+1} &= \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \\
\beta_k &= \frac{\rho_{k+1}}{\rho_k}, \\
\mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k.
\end{aligned} \tag{2}$$

The inner product is denoted with the '$\cdot$' operator, i.e. $\mathbf{a} \cdot \mathbf{b} := \sum_{i=1}^{N} \mathbf{a}_i \mathbf{b}_i$. For a hardware implementation this poses a problem, as it requires global communication across all elements of the vectors and, as we know, the vectors can be very large and have to be stored outside of the FPGA. On its own the scalar product can be easily implemented in a streaming fashion. The problem of global communication comes into play, because, as we see above, we need the value of ρ_{k+1} to compute $\mathbf{p}_{k+1}$. But to compute ρ_{k+1} the entire $\mathbf{r}_{k+1}$ must have been processed, which implies that $\mathbf{r}_{k+1}$ and $\mathbf{p}_{k+1}$ must be computed sequentially. For optimal

performance we would like to pipeline the entire solver, but this global communication in the middle of the algorithm prohibits this. Section 3.1 discusses a pipelined CG algorithm which circumvents this problem. But before we turn to the detailed treatment of this particular solver for FPGAs, let us explain how the CG solver can be used in the general context of iterative refinement methods.

2.2. Iterative Refinement

The main idea of the iterative refinement algorithm is to split the loop of the iterative solver into two loops: an outer loop $l = 0, \ldots, l_{\max}$ which runs as long as the global error is above $\varepsilon_{\text{outer}}$, and an inner loop $k = 0, \ldots, k_{\max}$ which runs either for a constant $k_{\max}$ or as long as the inner error is above $\varepsilon_{\text{inner}}$. The inner loop does not have to solve the problem exactly and therefore may utilize a low precision number format. It should only gain a little bit of accuracy in comparison to the initial data it was given. If it manages to gain this local accuracy then the outer loop will try to translate this into a global accuracy gain and thus advance the overall solution. In a certain sense the outer loop uses the inner loop as a black box mechanism to gain a little bit of accuracy in each step. The two processes are strongly decoupled and can therefore easily use different number formats. Another view of this procedure is to interpret the inner loop as a preconditioner to the outer solver. Let

$$A^{\text{high}}\mathbf{u}^{\text{high}} = \mathbf{b}^{\text{high}}$$

be the linear equation system to be solved in high precision. Superscript $^{\text{high}}$ indicates high precision vectors used in the outer loop, as opposed to the low precision vectors used in the inner loop. We first note that we need the matrix in high precision, so we assemble it once in the high precision format and cast it to the lower precision format(s).

The outer loop is very simple. We start with the initialization where we compute the initial defect

$$\begin{aligned}
\mathbf{u}_0^{\text{high}} &= \text{initial guess}, \\
\mathbf{d}_0^{\text{high}} &= \mathbf{b}^{\text{high}} - A^{\text{high}}\mathbf{u}_0^{\text{high}}, \\
d_0^{\text{high}} &= ||\mathbf{d}_0^{\text{high}}||, \\
\rho_0^{\text{high}} &= d_0^{\text{high}}.
\end{aligned}$$

Then we start the outer loop with index $l = 0, \ldots, l_{\max}$ and already give the control over to the inner solver, e.g. the CG solver, and let it solve

$$A\mathbf{u} = \mathbf{b} := \frac{1}{\rho_l^{\text{high}}}\mathbf{d}_l^{\text{high}} \tag{3}$$

approximately in low precision. The outer solver does not care how this is done, it is only important that the inner solver gains a little bit of accuracy. After the iterations of the inner solver we obtain the approximate result $\mathbf{u}_{k_{\max}}$ and the outer loop continues with an update of the current vector

$$\mathbf{u}_{l+1}^{\text{high}} = \mathbf{u}_l^{\text{high}} + \rho_l^{\text{high}}\mathbf{u}_{k_{\max}}.$$

With the new $\mathbf{u}_{l+1}^{\text{high}}$ we can compute a new defect and norm

$$\begin{aligned}
\mathbf{d}_{l+1}^{\text{high}} &= \mathbf{b}^{\text{high}} - A^{\text{high}}\mathbf{u}_{l+1}^{\text{high}}, \\
\rho_{l+1}^{\text{high}} &= ||\mathbf{d}_{l+1}^{\text{high}}||.
\end{aligned}$$

Now we check for global convergence

$$\rho_{l+1}^{\text{high}} < \varepsilon_{\text{outer}} d_0^{\text{high}}$$

in which case we are done. Otherwise, we continue by starting the inner solver again with the new defect $\mathbf{d}_{l+1}^{\text{high}}$.

The amount of accuracy we gain in one iterative refinement step depends on the condition of the matrix A [5]. This is not surprising as the condition also tells us how difficult it is to solve the problem directly with a single solver. The main advantage in the presented procedure is the need for just one matrix vector product in the outer loop, whereas the inner solver definitely needs to perform several steps to gain some precision. Consequently this means that the workload on the low precision format is much higher than on the high precision format. In this situation it is advantageous to use a highly parallel architecture for the inner solver, which does not need to contain high precision arithmetic. With the FPGA in particular, we can control the inner precision on a bit level and save on resources with every bit of precision in the inner solution process which is not needed for the final accuracy.

3. Variants of the Conjugate Gradient Solver

We have already outlined that the standard CG scheme (Eq. 2) cannot be fully pipelined because of repeated dependencies on scalar values obtained from global communication between vector elements. In this section we derive a *pipelined CG* variant which circumvents this problem. In high performance computing this problem has also been addressed as the global communications incur high latencies for a parallel computer. Our scheme here is very similar to the one proposed by Meurant [22]. For an overview of different CG variants see Dongarra and Eijkhout [8], for a broader discussion of parallel iterative solvers Demmel et al. [4].

In Section 3.2 we combine the pipelined CG with the iterative refinement technique resulting in a *mixed precision pipelined CG*. After an analysis of this iteration scheme, we develop a different iterative refinement strategy specifically adapted to the CG algorithm to obtain the *residual guided pipelined CG*.

3.1. Pipelined CG

We want to perform all vector operations of the CG algorithm (Eq. 2) in parallel and reorder them into:

$$\begin{aligned} \mathbf{u}_{k+1} &= \mathbf{u}_k + \alpha_k \mathbf{p}_k, \\ \mathbf{r}_{k+1} &= \mathbf{r}_k - \alpha_k \mathbf{q}_k, \\ \mathbf{p}_{k+1} &= \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k, \\ \mathbf{q}_{k+1} &= A\mathbf{p}_{k+1}. \end{aligned} \tag{4}$$

At first the parallel computation seems impossible because

$$\begin{aligned} \rho_{k+1} &= \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \\ \beta_k &= \frac{\rho_{k+1}}{\rho_k} \end{aligned}$$

and we must finish computing $\mathbf{r}_{k+1}$ to start computing $\mathbf{p}_{k+1}$. To avoid this problem we compute a $\sigma_k = \rho_{k+1}$ without the explicit knowledge of $\mathbf{r}_{k+1}$, namely

$$\sigma_k = \alpha_k(\alpha_k \mathbf{q}_k \cdot \mathbf{q}_k - \mathbf{p}_k \cdot \mathbf{q}_k).$$

Due to the orthogonality relations among the directions $\mathbf{p}$ and residuals $\mathbf{r}$ in the CG algorithm it follows that σ_k is algebraically *exactly* ρ_{k+1}. The indirect nature of the computation, however, could lead to higher roundoff errors. In fact, it is crucial that ρ_{k+1} is computed directly from $\mathbf{r}_{k+1}$ after the vector operations. Extensive tests have shown that with ρ_{k+1} computed like this, the pipelined scheme performs as well as the plain CG algorithm with respect to convergence behavior and stability, see Section 5.

The computation of all scalar products follows immediately after the vector operations in Eq. 4:

$$\begin{aligned} \rho_{k+1} &= \mathbf{r}_{k+1} \cdot \mathbf{r}_{k+1}, \\ \alpha_{k+1} &= \frac{\rho_{k+1}}{\mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}}, \\ \sigma_{k+1} &= \alpha_{k+1}(\alpha_{k+1}\mathbf{q}_{k+1} \cdot \mathbf{q}_{k+1} - \mathbf{p}_{k+1} \cdot \mathbf{q}_{k+1}), \\ \beta_{k+1} &= \frac{\sigma_{k+1}}{\rho_{k+1}}. \end{aligned} \tag{5}$$

In comparison to the plain CG algorithm we need to compute one more scalar product $\mathbf{q}_{k+1} \cdot \mathbf{q}_{k+1}$. But we gain the great advantage that now even for very large vectors, which must be streamed from outside the chip, we can compute one step of the algorithm in a fully pipelined fashion, assuming that the matrix A is sparse and does not enforce global communication itself, which is true for most Finite Element discretizations.

3.2. Mixed Precision Pipelined CG

We have applied the iterative refinement scheme presented in Section 2.2 to our pipelined CG variant from the previous section and label this combination the *mixed precision pipelined Conjugate Gradient*. In fact, we can greatly reduce the precision in the inner solver and still obtain the same accuracy as a reference double precision solver. Section 5 shows the exact numbers. However, the iterative refinement scheme is a general procedure applicable to all sorts of solver combinations in the same way. Although this gives a lot of flexibility, the high independence of the solvers has also a disadvantage: certain information obtained by one solver is not shared with the other one, so the other one may implicitly be performing more iterations to recover information which is already present. Concerning the mixed precision pipelined CG this happens whenever the CG is restarted from the outer loop. The restarted CG has no knowledge about the previous iterations.

3.3. Residual Guided Pipelined CG

The CG solver maintains a set of additional vectors $\mathbf{p}, \mathbf{r}, \mathbf{q}$ which contain information about the solution process. In the case of the general iterative refinement this information cannot be reused from one outer iteration to the other, because each time a different problem (Eq. 3) is solved. Moreover, our tests (Section 5) show that CG reacts delicately to frequent restarts and performs much better when maintaining its set of additional vectors for some time. On the other hand, CG may not rely on its auxiliary vectors for a long time, especially when computing in low precision, as these accumulate rounding errors over time. Even for a double precision solver it is recommended to recompute the defect $\mathbf{r}$ directly as $\mathbf{b} - A\mathbf{u}$ from time to time to avoid the loss of orthogonality.

Our new residual guided pipelined CG allows more frequent reinitialization without the complete loss of the generated auxiliary information. We replace the general outer loop of the iterative refinement with a new initialization

$$\begin{aligned} \mathbf{u}_0^{\text{high}} &= \text{initial guess}, \\ \mathbf{r}_0^{\text{high}} &= \mathbf{b}^{\text{high}} - A^{\text{high}}\mathbf{u}_0^{\text{high}}, \\ s_0^{\text{high}} &= ||\mathbf{r}_0^{\text{high}}||, \\ \mathbf{u}_0 &= 0, \\ \mathbf{r}_0 &= \frac{1}{s_0^{\text{high}}}\mathbf{r}_0^{\text{high}}, \\ \alpha_0 &= 0, \\ \beta_0 &= 0. \end{aligned}$$

and a new high precision outer loop

$$\begin{aligned} \mathbf{u}_{l+1}^{\text{high}} &= \mathbf{u}_l^{\text{high}} + s_l^{\text{high}}(\mathbf{u}_k + \alpha_k \mathbf{p}_k), \\ \mathbf{r}_{l+1}^{\text{high}} &= \mathbf{b}^{\text{high}} - A^{\text{high}}\mathbf{u}_{l+1}^{\text{high}}, \\ s_{l+1}^{\text{high}} &= ||\mathbf{r}_{l+1}^{\text{high}}||. \end{aligned}$$

The low precision $\mathbf{u}_k, \alpha_k, \mathbf{p}_k$ are results from the inner loop, see Eqs. 4 and 5. The computation of the defect here is very similar to the general iterative refinement. However, instead of starting the inner low precision solver with a new problem we now continue with the same right hand side and reuse the information from the previous run of the inner solver by setting

$$\begin{aligned}
\mathbf{u}_0 &= 0, \\
\mathbf{r}_0 &= \frac{1}{s_{l+1}^{\text{high}}} \mathbf{r}_{l+1}^{\text{high}}, \\
\mathbf{p}_0 &= \mathbf{p}_{k_{\max}} - (\mathbf{r}_0 \cdot \mathbf{p}_{k_{\max}})\, \mathbf{r}_0, \\
\alpha_0 &= 0, \\
\beta_0 &= \frac{s_{l+1}^{\text{high}}}{s_l^{\text{high}} \rho_{k_{\max}}}.
\end{aligned}$$

With these values the pipelined solver enters in Eq. 4. The main effect is that we still have an idea about the orthogonal descend directions $\mathbf{p}$ from the previous steps and thus can continue decreasing the error faster. We do not simply copy the new direction as the computation of the new defect in the outer loop introduces a new defect $\mathbf{r}_0$ which must fulfill the orthogonality relations. Thus we correct the old $\mathbf{p}_{k_{\max}}$ to make it orthogonal to $\mathbf{r}_0$.

The last improvement we have applied to the scheme is the evaluation of the accumulation in the scalar products in a higher precision. This is an addition over a very high number of elements, therefore it makes sense to reserve a higher precision for this operation. In terms of resources this is not too expensive as the multiplications and the final format still use the low precision. This is a standard technique to avoid losing too much accuracy due to the very high number of addends. We use the double float format in these accumulations.

4. Test Framework

For the different solvers we examine four main parameters: accuracy, iteration count until convergence, area coverage and frequency on the FPGA. The first two parameters are evaluated on the CPU in a framework with the ability to simulate different number precisions. The two other parameters are evaluated within the Xilinx ISE.

4.1. CPU Tests

To emulate different floating point formats, we extend the `half` class (available as part of the OpenEXR framework by Industrial Light & Magic). Our version of this class supports all combinations of mantissa and exponent bits up to the IEEE floating point standard (s23e8). Additionally, we can turn off denormalized numbers (denorms) and rounding to nearest, as they are often not implemented and even not desired in floating point FPGA libraries. We tested five different precision formats for the inner loop:

- double, native CPU 64-bit s52e11, with full compiler optimizations, but without SSE,
- float, native CPU 32-bit s23e8, with full compiler optimizations, but without SSE,
- s23e8, without denorms and rounding,
- s20e8, without denorms and rounding,
- s17e8, without denorms and rounding.

The test with the inner precision set to double helps us to distinguish between the effects caused by the iterative refinement scheme and those caused by the reduced precision. Reducing the inner format below s17e8 yields problems that cannot be solved for large matrices (number of unknowns, matrix condition) using the mixed precision approach (cf. [14]). We also consider the choice of the inner stopping criteria. We tested both possibilities of running a fixed number of inner iterations and iterating the inner solver until one to four digits of accuracy are gained locally.

4.2. FPGA Tests

The evaluation of the required area on a FPGA is performed with the Xilinx ISE 7.1. We use the customizable floating point library designed by Belanovic and Leeser [1] to generate fully generic floating point solver kernels in VHDL. The CG algorithm consists of

- multiply and add operations between vectors,
- a banded matrix vector multiply,
- scalar products between vectors.

Note, that all three operations translate into multiply and add operations on the components. However, not all of these multiply and adds are the same, since we want to use higher precision in the accumulation stage of a scalar product. Building the entire CG core based on the generic library allows us to experiment with different data precisions in different stages.

Unfortunately, we have no hardware platform available for actual application performance testing. Such numbers would be very specific to the device, as for large vectors which do not fit the block RAM on chip, the data must be streamed from outside and then the number of user IO pins and the bandwidth of the connection to the larger RAM become crucial for the overall performance. But nevertheless we plan to run performance tests on actual hardware to show how the resource savings translate in actual application speedup. Thereby, we could also quantify another advantage of mixed precision methods, namely the reduction in bandwidth requirements due to the smaller number formats.

5. Results

We first confirm that in the context of this paper all iterative refinement schemes deliver exactly the same accuracy as the double precision reference solver, independent of the involved number formats and refinement strategies. Clearly, if the inner number format is too small in comparison to the condition of the matrix, which grows quadratically with the inverse grid element size, the solvers simply fail to converge. These cases are marked as *divergent* in all tables. But if the solver converges then we can always achieve the accuracy of the reference solution.

Concerning the accuracy of the pipelined CG scheme, we have run extensive tests in comparison to the plain CG scheme both in the direct setting and the iterative refinement approach and conclude that the pipelined CG solver shows almost identical convergence behavior to its classical counterpart. The maximal deviation in the iteration count is less than 3% and neither solver is a clear winner, both perform sometimes better than the other.

5.1. Performance Comparison

In the following tables we always show at the top the performance of the direct reference double precision solver for comparison. The iterative solvers with the different inner precisions described in Section 4 then follow in separate blocks. For the iterative solvers we have two performance numbers [sum of all inner iterations]:[outer iterations]. The comparison rows at the end of each block show the ratios [inner+outer iterations]/[direct double iterations] and [outer iterations]/[inner+outer iterations]. The former shows how many more iterations overall must be performed due to the iterative scheme and the reduced precision. The latter then puts into perspective which part of the overall work load must be performed in double precision.

5.2. Mixed Precision Pipelined CG

With the mixed precision pipelined CG solver (cf. Section 2.2) we perform inner iterations until the inner defect relative to the initial defect (when we started the loop) drops below a certain constant, see Eq. 1. In other words, we demand of the inner solver to reduce the inner defect by one to four digits, see Table 1. We know that the low precision format cannot gain a lot of accuracy in one loop, so it will struggle hard for improvement if the solution is already fairly accurate. Therefore, we want to abort the loop before we experience these diminishing returns. But we also do not want to abort too early, because the CG loses all its intermediate vectors with each outer loop iteration.

Comparing the solver running with double inner precision first, we see that each additional outer loop iteration has a negative effect on the overall number of iterations. The comparison row $\delta 4$/direct shows that the iterative scheme as such needs approximately 1.35 times more iterations than the direct solver.

Table 1. Performance of the mixed precision pipelined CG with the number of inner loop iterations depending on a reduction of the inner defect by δ digits. Section 5.1 explains the table structure.

N(n)	66,049(8)	263,169(9)	1,050,625(10)
double reference solver			
direct	342	676	1357
double			
$\delta = 1$	859 : 10	1648 : 10	3472 : 10
$\delta = 3$	544 : 4	1098 : 4	2233 : 4
$\delta = 4$	459 : 3	915 : 3	1817 : 3
$\delta 4$/direct	1.35 : 6e-3	1.36 : 3e-3	1.34 : 2e-3
float			
$\delta = 1$	882 : 10	1601 : 10	3425 : 11
$\delta = 3$	564 : 4	1155 : 4	2563 : 4
$\delta = 4$	542 : 4	1064 : 4	2191 : 4
$\delta 4$/direct	1.60 : 7e-3	1.58 : 4e-3	1.62 : 2e-3
s23e8			
$\delta = 1$	1003 : 10	2267 : 10	4070 : 11
$\delta = 3$	732 : 4	1673 : 4	3287 : 5
$\delta = 4$	780 : 3	1601 : 4	3685 : 4
$\delta 3$/direct	2.15 : 5e-3	2.48 : 2e-3	2.43 : 2e-3
s20e8			
$\delta = 1$	1103 : 10	2728 : 11	9889 : 18
$\delta = 3$	1170 : 6	3220 : 9	12578 : 17
$\delta = 4$	1293 : 6	4073 : 8	17732 : 18
$\delta 1$/direct	3.25 : 9e-3	4.05 : 4e-3	7.30 : 2e-3
s17e8			
$\delta = 1$	1138 : 10	2673 : 11	divergent
$\delta = 3$	1168 : 6	3220 : 9	divergent
$\delta = 4$	1290 : 6	4090 : 9	divergent
$\delta 1$/direct	3.36 : 9e-3	3.97 : 4e-3	divergent

For the lower precision formats this factor grows to 1.62 for float and 2.48 for s23e8. The native CPU float benefits, because it is often promoted to double precision for intermediate computations and register storage in the CPU. For s20e8 and s17e8 we have to execute up to 4 times more iterations up to level 9. For level 10 either the solver diverges or we obtain an excessive number of iterations. Given the high condition of the matrix, the inner precision is too low

to represent the inner accuracy gains correctly. Either the inner solver thinks it is gaining inner accuracy, but in reality it drifts in the wrong global direction, or it is incapable of making any progress at all. In any case executing more iterations does not help the goal of improving the global accuracy. For such low precision more frequent outer loop iterations must be performed.

Looking at the comparison row of the s23e8 format, we see that the costs of achieving the same accuracy as the double precision reference solver are approximately 2.48 times more iterations. However, most of these iterations happen on the low precision format and only 1% or even far less for the larger problems still require double precision. So while we do have to do more iterations, the FPGA can do this in parallel on many simple s23e8 arithmetic units, without any handling of denormalized numbers or rounding, whereas the reference double precision solver executes on an optimized FPU with potentially higher internal precision. Implementing the arithmetic units of the FPU on the FPGA would consume a very high number of resources (see Section 5.4).

5.3. Residual Guided Pipelined CG

Rather than correcting defects, here we use the high precision residuals as guidance for the low precision solver, see Section 3.3. Table 2 shows that with this scheme we may enter the outer loop more often without fearing the deteriorating effects of the CG restarts. The stability of the scheme is also emphasized by the fact that in general the low precision formats s20e8 and s17e8 can handle large matrices faster than in Table 1. The performance does not drop so dramatically against the double precision reference and also does not increase so severely with the growing problem size. However, the best performance is delivered by the s23e8 format with 10 inner iterations. Approximately 1.8 times more iterations are performed, but only 9% of them use the double precision format.

5.4. FPGA Design Space

We list the area consumption and maximum frequency of the pipelined CG core under different precisions (Table 3). These numbers were obtained with the XST Synthesizer opimitizing for speed, the global timing optimization set for maximal delay, no IOB register use and automatic MUL18x18 selection or pipelined LUT multipliers. For a clearer comparison we want to 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 both slices and user IO and therefore the large xc2v8000 is used for all designs.

We synthesize the core of the algorithm. It computes the vector operations and scalar products but does not perform the further combinations of the scalar values nor anything else in the outer loop. We think that a micro-processor on the FPGA board would be a good candidate to take over this task. The outer loop can in principle also be implemented on the FPGA, but since it is executed only infrequently and requires a high precision matrix vector product it is doubtful if these resources were wisely spend, after all the idea of mixed precision methods is to replace the expensive cores with many parallel simple ones. An interesting solution would be to use parameter controlled arithmetic units which could be used as either one high precision or many low precision multipliers.

Table 2. Performance of the residual guided CG with a constant number of inner loop iterations (i10-i50). Section 5.1 explains the table structure.

N(n)	66,049(8)	263,169(9)	1,050,625(10)
double reference solver			
direct	342	676	1357
double			
i10	343 : 35	677 : 68	1358 : 136
i25	343 : 14	677 : 28	1358 : 55
i50	343 : 7	677 : 14	1358 : 28
i50/direct	1.00 : 2e-3	1.00 : 2e-3	1.00 : 2e-3
float			
i10	536 : 54	1233 : 124	2495 : 250
i25	570 : 23	1178 : 48	2527 : 102
i50	633 : 13	1268 : 26	2522 : 51
i10/direct	1.72 : 0.09	2.01 : 0.09	2.02 : 0.09
s23e8			
i10	525 : 53	1154 : 116	2222 : 223
i25	594 : 24	1236 : 50	2480 : 100
i50	595 : 12	1241 : 25	2554 : 52
i10/direct	1.68 : 0.09	1.88 : 0.09	1.80 : 0.09
s20e8			
i10	815 : 82	1662 : 169	3849 : 386
i25	881 : 36	1936 : 78	3462 : 140
i50	1021 : 21	2475 : 50	4050 : 82
i10/direct	2.62 : 0.09	2.71 : 0.09	3.12 : 0.09
s17e8			
i10	1148 : 116	2609 : 263	4540 : 455
i25	1344 : 55	2589 : 105	divergent
i50	2343 : 48	2573 : 52	6671 : 137
i10/direct	3.70 : 0.09	4.25 : 0.09	3.68 : 0.09

Table 3. Estimated area consumption and maximum frequency of the pipelined CG core on the xc2v8000ff1517-5.

precision	slices	MUL18x18s	IOBs	frequency
PipeLUT multiplier				
s17e11	4517	0	498	71 MHz
s20e11	5102	0	549	67 MHz
s23e11	5812	0	600	66 MHz
s52e11	14545	0	1093	49 MHz
MUL18x18 multiplier				
s17e11	4449	12	498	134 MHz
s20e11	4902	48	549	87 MHz
s23e11	5412	57	600	85 MHz
s52e11	10271	135	1093	59 MHz

If the entire core is implemented in logic the area consumption is quadratic in the size of the number format as expected (Table 3). Utilization 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.

Let us pick an example to summarize the results. By going from a direct double to a double-s23e8 residual guided solver in case of a large Poisson Problem (Section 2) with $1,040,625(10)$ elements per vector, we can obtain exactly the same accuracy in the final result by executing 9% of the matrix vector products in double precision and all other 2222 in s23e8 precision (Table 2). We will perform 1.80 times more iterations than the double solver, but will also have 2.5 times more logic resources available, 1.44 times higher clock frequency and 0.55 times less IO requirements. So one possiblity would be to use the xc2v1500 (xc2v2000 for full IO) instead of the xc2v4000 (xc2v6000 for full IO) and suffer just a $1.80/1.44 = 1.25$ slowdown (plus the costs of the outer loop). Obviously, we can easily gain speedups by using the large devices and implementing the low precision core in parallel, but the actual speedup is more difficult to estimate as it depends on the final frequency of the parallel design. If we want to reduce the double precision computations in the outer loop to a minimum, we may, alternatively, use the mixed precision pipelined CG and perform just five double precision matrix vector products and 2.43 times more iterations on the low precision format (Table 1) to arrive at the double precision solution.

6. Conclusions and Future Work

We have examined how mixed precision iterative refinement methods offer a tradeoff between computation and the usually more restricted resources of off-chip data bandwidth, latency and resource utilization. Although the number precision is reduced dramatically in most computations no loss of accuracy in the final result is encountered, which is the guiding idea behind the algorithmic resource optimization. In detail, we have studied the Conjugate Gradient solver which is often employed in the solution process of PDEs. Based on ideas from high performance computing our pipelined CG variant is much better suited for an efficient FPGA implementation than the plain CG algorithm. Both the standard defect correction and our new residual guided iterative refinement work well with the pipelined CG and offer different performance characteristics. The generic core of the algorithm written in VHDL allows to estimate the resource savings in advance and our CPU test suite can report on the accuracy and performance of the chosen solver and precision combination.

The mixed precision scheme requires additional computational resources inside or outside of the FPGA for the efficient execution of the outer loop in high precision. But depending on the processing power the workload of this component may be easily varied between 1%-10% or even lower for large problems. In future, we will consider efficient implementations of the outer loop within the same device or a combination of different devices.

Although FPGAs are very well suited for such gradual number format changes, the mixed precision approach can be applied to several hardware and software platforms. In particular, for coarse grained reconfigurable architectures a decision must be made before fabrication which parts will be hardwired and which configurable. This decision typically implies preferred data formats for which the processing elements are optimized. Iterative refinement methods can efficiently exploit the preferred number format, e.g. single float, without losing the ability to obtain fast and accurate results in a higher precision format, such as double float. In future we want to further examine these possibilities.

Acknowledgments

We thank Pavle Belanovic and Miriam Leeser for the availability of their generic floating point library, Industrial Light & Magic for the `half` class and Xilinx for their ISE. This research has been partly funded by a Max Planck Center for Visual Computing and Communication fellowship.

References

[1] P. Belanovic and M. Leeser. A library of parameterized floating-point modules and their use. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-*

Programmable Logic and Applications, pages 657–666, London, UK, 2002. Springer-Verlag.

[2] H. Bowdler, R. Martin, G. Peters, and J. Wilkinson. Handbook series linear algebra: Solution of real and complex systems of linear equations. *Numerische Mathematik*, 8:217–234, 1966.

[3] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 75–85, New York, NY, USA, 2005. ACM Press.

[4] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In *Acta Numerica 1993*, pages 111–198. Cambridge University Press, Cambridge, UK, 1993.

[5] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjeek, and E. J. Riedy. Error bounds from extra precise iterative refinement. *ACM Transactions on Mathematical Software*, 2006.

[6] J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A flexible floating-point format for optimizing data-paths and operators in FPGA based DSPs. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 50–55, New York, NY, USA, 2002. ACM Press.

[7] T. Dillon. An efficient architecture for ultra long FFTs in FPGAs and ASICs. In *High Performance Embedded Computing Conference (HPEC04)*, Sept. 2004.

[8] J. Dongarra and V. Eijkhout. Finite-choice algorithm optimization in Conjugate Gradients. LAPACK Working Note 159, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, Jan. 2003.

[9] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 86–95, New York, NY, USA, 2005. ACM Press.

[10] F. Fang, T. Chen, and R. Rutenbar. 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*, pages 879–892, Sept. 2002.

[11] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 523–533, London, UK, 2002. Springer-Verlag.

[12] A. A. Gaffar, O. Mencer, W. Luk, and P. Y. K. Cheung. Unifying bit-width optimisation for fixed-point and floating-point designs. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04)*, pages 79–88, 2004.

[13] K. O. Geddes and W. W. Zheng. Exploiting fast hardware floating point in high precision computation. In *ISSAC '03: Proceedings of the 2003 international symposium on Symbolic and algebraic computation*, pages 111–118, New York, NY, USA, 2003. ACM Press.

[14] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of mixed precision solvers in FEM simulations on anisotropic grids. *International Journal of Parallel, Emergent and Distributed Systems*, 2006. submitted.

[15] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *18th International Parallel and Distributed Processing Symposium (IPDPS04),Workshop 3*, page 149b, 2004.

[16] M. Haselman, M. Beauchamp, A. Wood, S. Hauck, K. Underwood, and K. S. Hemmert. A comparison of floating point and logarithmic number systems on FPGAs. *IEEE Transactions on Computers*, 2005. submitted.

[17] P. Jackson, C. Chan, C. Rader, J. Scalera, and M. Vai. A systolic FFT architecture for real time FPGA systems. In *High Performance Embedded Computing Conference (HPEC04)*, Sept. 2004.

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

[19] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for FPGAs. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 185, Washington, DC, USA, 2003. IEEE Computer Society.

[20] G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific n-body simulations. In *FCCM '02: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 182, Washington, DC, USA, 2002. IEEE Computer Society.

[21] R. Matousek, M. Tichy, Z. Phol, J. Kadlec, C. Softley, and N. Coleman. Logarithmic number systems and floating-point arithmetics on FPGA. In *12th International Conference on Field Programmable Logic and Applications*, pages 627–636, London, UK, 2002. Springer-Verlag.

[22] G. Meurant. Multitasking the conjugate gradient method on the CRAY X-MP/48. *Parallel Computing*, 5:267–280, 1987.

[23] M. C. Smith, J. S. Vetter, , and S. R. Alam. Scientific computing beyond CPUs: FPGA implementations of common scientific kernels. In *2005 MAPLD International Conference*, Sept. 2005.

[24] K. Turner and H. F. Walker. Efficient high accuracy solutions with gmres(m). *SIAM J. Sci. Stat. Comput.*, 13(3):815–825, 1992.

[25] K. Underwood. FPGAs vs. CPUs: Trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM Press.

[26] A. Walters and P. Athanas. A scaleable FIR filter using 32-bit floating-point complex arithmetic on a configurable computing machine. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines 1998*, pages 333–334, Apr. 1998.

[27] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, New York, NY, USA, 2005. ACM Press.