GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement

Hartwig Anzt[1], Piotr Luszczek[2], Jack Dongarra[2,3,4], and Vincent Heuveline[1]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany,
[2] University of Tennessee, Knoxville, USA,
[3] Oak Ridge National Laboratory, Oak Ridge, USA
[4] University of Manchester, Manchester, UK
{hartwig.anzt, vincent.heuveline}@kit.edu
{luszczek, dongarra}@eecs.utk.edu

Abstract. In hardware-aware high performance computing, block- asynchronous iteration and mixed precision iterative refinement are two techniques that are applied to leverage the computing power of SIMD accelerators like GPUs. Although they use a very different approach for this purpose, they share the basic idea of compensating the convergence behaviour of an inferior numerical algorithm by a more efficient usage of the provided computing power. In this paper, we want to analyze the potential of combining both techniques. Therefore, we implement a mixed precision iterative refinement algorithm using a block-asynchronous iteration as an error correction solver, and compare its performance with a pure implementation of a block-asynchronous iteration and an iterative refinement method using double precision for the error correction solver. For matrices from the University of Florida Matrix collection, we report the convergence behaviour and provide the total solver runtime using different GPU architectures.

Keywords: mixed precision iterative refinement, block-asynchronous iteration, GPU, linear system, relaxation

1 Introduction

Classical relaxation methods such as Gauss-Seidel and Jacobi require data transfer between each iteration which constitutes a synchronization point. This implies a severe restriction for parallel implementations. Block-asynchronous iteration removes this synchronization barrier, updating components using the latest available values. It allows a large freedom in the update order and the number of updates per component, while every component update uses the latest available values for the other components. In the end, the obtained algorithm is neither deterministic nor does it imply convergence for all systems that can be solved by the classical Jacobi approach, in fact it requires the linear equation system to fulfill additional conditions. While, due to the poor convergence rate, they may seem to be very unattractive from the mathematical point of view, the block-asynchronous iteration is, in contrast to most other iterative methods, able to exploit the high computational power of modern hardware platforms, often accelerated by GPUs. Another well-known technique

used to leverage the potential of accelerators is mixed precision iterative refinement. The basic idea is to use a lower precision format for the error correction solver inside an iterative refinement method at full precision. Without impacting the accuracy of the final solution approximation, the acceleration of the solving process is possible since the computations in the less complex floating point format can be conducted faster on the respective device. While the time for computations in the usually implemented single and double precision formats differs by a factor of two for most devices, additional acceleration may be possible since using single precision reduces the pressure on the memory bandwidth, that is often crucial in scientific computing on hybrid hardware. An open question is how a combination of these two techniques impacts the convergence and properties and the performance. On the one hand the methods are similar: they both compensate their low complexity by leveraging the high computational power of GPUs. But on the other hand, they are contradictory since the iterative refinement artificially introduces synchronization points that we try to avoid. The most suitable applications are linear systems with condition numbers which require high iteration counts of the error correction solver. The mixed precision approach may suffer from these, since the error correction is impacted by them. The paper is organized as follows. First, we provide some mathematical background by outlining the algorithms for iterative refinement and the mixed precision variant, and block-asynchronous iteration. We then introduce the hardware platforms used for the experiments and give details about the linear equation systems we target. Additionally, we outline the GPU implementation we use for the tests. In the numerical experiment section, we compare the convergence behaviour of iterative refinement using a double-precision and a single-precision error correction solver. We report the total solver runtimes and compare it with a plain implementation of the block-asynchronous iteration for various GPUs. In the last section we conclude and provide ideas for further optimization.

2 Mathematical Background

Block-Asynchronous Iteration. The motivation for an asynchronous iteration is modern hardware, which provides a large number of cores that achieve excellent performance when running in parallel, but suffer when synchronizing or exchanging data. Therefore, algorithms that lack any synchronization would achieve outstanding performance on these devices, while most of the numerical algorithms are poorly parallel and require regular data exchange. For computing the next iteration in relaxation methods, one usually requires the latest values of all components. For some algorithms, e.g., Gauss-Seidel [16], even the already computed values of the current iteration step are used. This requires a strict order of the component updates, limiting the parallelization potential to a stage, where a component cannot be updated several times before all the other components are updated.

If this order is not adhered to, i.e., the individual components are updated independently and without consideration of the current state of the other components, the resulting algorithm is called a chaotic or asynchronous iteration method. In the past, the convergence behaviour and performance of these methods were analyzed

in several papers [13, 12, 4, 10]. Due to the superior convergence properties of synchronized iteration methods, they came out of the main focus of high performance computing, while research was put on investigating the convergence properties [20, 5]. Today, due to the complexity of heterogeneous hardware platforms and the large number of computing units in parallel devices like GPUs, these schemes may become interesting again for applications like multigrid methods, where highly parallel smoothers are required on the distinct grid levels [9]. While traditional relaxation methods like the sequential Gauss-Seidel obtain their efficiency from their fast convergence, an asynchronous iteration scheme may compensate for its inferior convergence behavior by superior scalability [3]. In [2] we proposed a block-asynchronous iteration, that, in addition to the global iterations, iterates on the subdomains determined by the iteration components that are handled by the same stream in the GPU implementation. The motivation for this is due to the design of graphics processing units and the CUDA programming language. As the subdomains are relatively small and the data needed largely fits into the multiprocessor's cache, these additional iterations on the subdomains come for almost free. During these local iterations, the x values used from outside the block are kept constant, equal to their values at the beginning of the global iteration. After the local iterations, the updated values are communicated. This approach is inspired by the well known hybrid relaxation schemes [9, 8]. The obtained algorithm, visualized in Figure 1, can be written as a componentwise update of the solution approximation to form $x_k^{(m+1)}$:

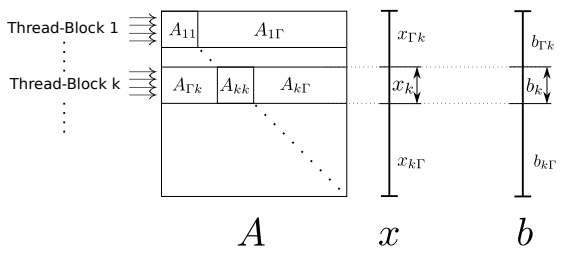
$$x_k^{(m)} + \frac{b_k}{a_{kk}} - \underbrace{\sum_{j=1}^{T_S} \frac{a_{kj} x_j^{(m-v(m+1,j))}}{a_{kk}}}_{\text{global part}} - \underbrace{\sum_{j=T_S}^{T_E} \frac{a_{kj} x_j^{(m)}}{a_{kk}}}_{\text{local part}} - \underbrace{\sum_{j=T_E}^{n} \frac{a_{kj} x_j^{(m-v(m+1,j))}}{a_{kk}}}_{\text{global part}}, \tag{1}$$

where T_S and T_E denote the starting and the ending indexes of the matrix/vector part in the thread block. Furthermore, for the local components, the antecedent values are always used, while for the global part, the values from the beginning of the iteration are used. The shift function $v(m+1, j)$ denotes the iteration shift for the component j – this can be positive or negative, depending on whether the respective other thread block has already conducted more or less iterations. Note that this gives a block Gauss-Seidel flavor to the updates. It should also be mentioned that the shift function may not be the same in different thread blocks. While the GPU hardware encourages this approach, the idea is similar to a two-staged asynchronous iteration [7].

Mixed Precision Iterative Refinement. While error correction methods have been known for more than 100 years, they finally became of interest with the rise of computer systems in the middle of the last century. The core idea is to use the residual of a computed solution as the right-hand side to solve a correction equation.

The motivation for the iterative refinement method can be obtained from Newton's method. Newton developed a method for finding successively better approximations to the zeros of a function $f(\cdot)$ by updating the solution approximation x_i through

$$x_{i+1} = x_i - (\nabla f(x_i))^{-1} f(x_i). \tag{2}$$



$$x_k += D_{kk}^{-1}\left(b_k - A_{\Gamma k}x_{\Gamma k} - A_{kk}x_k - A_{k\Gamma}x_{k\Gamma}\right)$$

Fig. 1: Visualizing the asynchronous iteration in block description used for the GPU implementation.

We may now apply Newton's method (2) to the function $f(x) = b - Ax$ with $\nabla f(x) = -A$. By defining the residual $r_i := b - Ax_i$, we obtain

$$\begin{aligned} x_{i+1} &= x_i - (\nabla f(x_i))^{-1} f(x_i) \\ &= x_i + A^{-1}(b - Ax_i) \\ &= x_i + A^{-1} r_i. \end{aligned}$$

Denoting the solution update with $c_i := A^{-1} r_i$, we can obtain:

1: initial guess as starting vector: x_0
2: compute initial residual: $r_0 = b - Ax_0$
3: **while** ($\| Ax_i - b \|_2 > \varepsilon \| r_0 \|$) **do**
4: $\quad r_i = b - Ax_i$
5: $\quad$ solve: $Ac_i = r_i$
6: $\quad$ update solution: $x_{i+1} = x_i + c_i$
7: **end while**

Algorithm 1: Error Correction Method

The underlying idea of mixed precision error correction methods is to use different precision formats within the algorithm of the error correction method, updating the solution approximation in high precision, but computing the error correction term in lower precision which has been suggested before [15, 14, 6, 11].

Hence, one regards the inner correction solver as a black box, computing a solution update in lower precision. The term high precision refers to the precision format that is necessary to display the accuracy of the final solution, and we can obtain the following algorithm where $.^{high}$ denotes the high precision value and $.^{low}$ denotes the value in low precision. The conversion between the formats will be left abstract throughout this paper. Because the conversion of the matrix A is especially expensive, it should be stored in both precision formats, high and low precision. This leads to the drawback of a higher memory need.

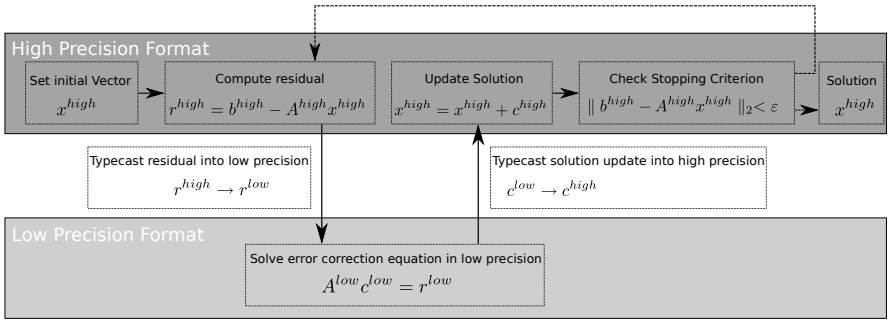


Fig. 2: Visualizing the mixed precision approach to an iterative refinement method.

Using the displayed algorithm, we obtain a mixed precision solver. If the final accuracy does not exceed the smallest number ε_{low} that can be represented in the lower precision, it may generate the same approximation quality as if the solver was performed in the high precision format. It should be mentioned, that the solution update of the error correction solver is usually not optimal for the outer system, since the representation of the problem in the lower precision format contains rounding errors, and it therefore solves a perturbed problem. When comparing the algorithm of an error correction solver to a plain solver, it is obvious, that the error correction method has more computations to execute. Each outer loop consists of the computation of the residual error term, a typecast, a vector update, the scaling process, the inner solver for the correction term, the reconversion of the data and the solution update. The computation of the residual error itself consists of a matrix-vector multiplication, a vector addition and a scalar product. The mixed precision refinement approach to a certain solver is superior to the plain solver in high precision, if the additional computations and typecasts are overcompensated by the cheaper inner correction solver using a lower precision format [1, 11].

3 Experiment Setup

Linear Equation Systems. In our experiments, we search for the approximate solutions of linear systems of equations, where the respective matrices are taken from the University of Florida Matrix Collection (UFMC; see `http://www.cise.ufl.edu/research/sparse/matrices/`).

Due to the convergence properties of the iterative methods we analyze, the experiment matrices have to be chosen properly, fulfilling the necessary and sufficient convergence condition [12].

The matrix properties and sparsity plots are in Table 1. and Figure 3.

The first matrix, CHEM97ZTZ, comes from statistics [5]. Matrices FV1 and FV3 are finite element discretizations of the Laplace equation on a 2D mesh. Therefore, they

[5] For more details see http://www.cise.ufl.edu/research/sparse/mat/Bates/README.txt

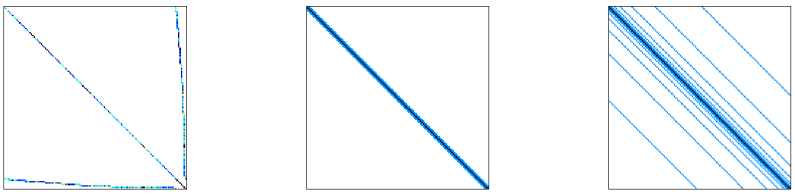
Matrix name	$\#n$	$\#nnz$	cond(A)	cond($D^{-1}A$)	$\rho(M)$
CHEM97ZTZ	2,541	7,361	1.3e+03	7.2e+03	0.7889
FV1	9,604	85,264	9.3e+04	12.76	0.8541
FV3	9,801	87,025	3.6e+07	4.4e+03	0.9993
TREFETHEN_2000	2,000	41,906	5.1e+04	6.1579	0.8601

Table 1: Dimension and characteristics of the SPD test matrices and the corresponding iteration matrices.

share a common sparsity structure, but differ in dimension and condition number due to the different finite element choice. The matrix TREFETHEN_2000 [21] is a 2000×2000 matrix where all entries are zero except for the ones at the positions (i, j) where $|i - j| = 2, 4, 8, 16 \ldots$. Furthermore, the main diagonal is filled with the primes $2, 3, 5, 7, 11 \ldots 17389$. Hence, this matrix has many off-diagonal entries distributed over the diagonals that are by a power of 2 distant to the main diagonal.

Implementation Issues. The GPU implementations of the block-asynchronous iteration is based on CUDA [18], while the respective libraries used are from CUDA 2.3 for the C1060 and the GTX280, and CUDA 4.0.17 [19] for the C2070 and GTX580 implementation. The kernels updating the respective components, launched through different streams, use thread blocks of size 512. The thread block size, the number of streams, along with other parameters, were determined through empirically based tuning. For the iterative refinement implementation we use a first outer iteration to analyze the residual improvement and then adapt the number of inner iterations such that the residual improvement equals the accuracy of floating point precision in every outer update. Hence, while the first error correction loop may provide different improvement for the distinct test cases, the further loops all decrease the residual by 6 to 8 digits.

In case of the mixed precision implementations, the error correction solver is implemented using single precision. Hence, due to the low precision representation of the linear equation system, additional rounding errors may be expected, slowing down the convergence of the iterative refinement. To analyze this issue, we compare



(a) CHEM97ZTZ (b) FV, FV3 (c) TREFETHEN_2000

Fig. 3: Sparsity plots of test matrices.

Name	Tesla C2070	Tesla C1060	GTX 580	GTX280a
Chip	T20	T10	GF110	GT200
Transistors	$3 \cdot 10^9$	$1.4 \cdot 10^9$	$3 \cdot 10^9$	$1,4 \cdot 10^9$
Core frequency	1.15 GHz	1.3 GHz	1.5 GHz	1.3 GHz
Thread Processors	448	240	512	240
GFLOPS (single)	1030	933	1580	933
GFLOPS (double)	515	78	790	78
Shared Memory/L1	64 KB	16 K	64 K	16 KB
L2 Cache	768 KB	-	768 KB	-
Memory	6 GB GDDR5	4 GB GDDR3	1.5 GB GDDR5	1 GB GDDR3
Memory Frequency	1.5 GHz	0.8 GHz	2 GHz	1.1 GHz
Memory Bandwidth	144 GB/s	102 GB/s	192.4 GB/s	141 GB/s
ECC Memory	yes	no	yes	no
Power Consumption	190 W	200 W	244 W	236 W
IEEE double/single	yes/yes	yes/partial	yes/yes	yes/partial

Table 2: Key system characteristics of the four GPUs used. Computation rate and memory bandwidth are theoretical peak values [17].

in a first experiment the convergence behaviour of the iterative refinement method using a double- and a single- precision error correction solver, respectively. Using different precision formats, the vectors and the linear system have to be converted from double to single precision. This typecast, handled by the GPU, triggers some overhead and may be crucial for problems where only very few iterations of the error correction solver are executed.

To analyze the impact of the overhead of iterative refinement and the use of different precision formats we provide the solver runtimes for the different linear equation systems for the plain block-asynchronous iteration in double precision, the iterative refinement in double precision and the mixed precision iterative refinement, whereas the latter ones use the block-asynchronous iteration as an error correction solver.

Hardware Platforms. We target four GPU architectures located at the Engineering Mathematics and Computing Lab (EMCL)[6] at the Karlsruhe Institute of Technology, Germany, to analyze the potential of mixed precision block-asynchronous iteration. They are taken from the Fermi and the Tesla line of Nvidia. The C2070 and the C1060 are the server versions of the line, the GTX580 and the GTX280 are the consumer version, respectively. While the chip and on-board memory specifications are given in table 2, the host system may have minor influence on the performance, since all computations are exclusively handled by the graphics. Note that the price for the larger (ECC protected) memory in the server versions is a lower memory bandwidth.

[6] Supported by NVIDIA as Cuda Research Center

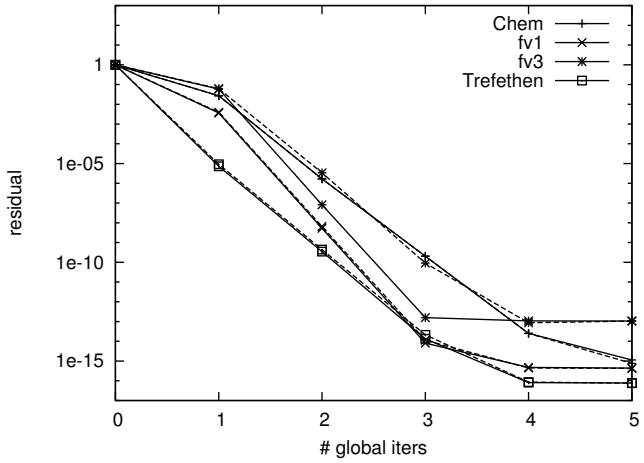


Fig. 4: Iterative refinement convergence, solid lines are double-precision error correction, dashed lines are single-precision error correction.

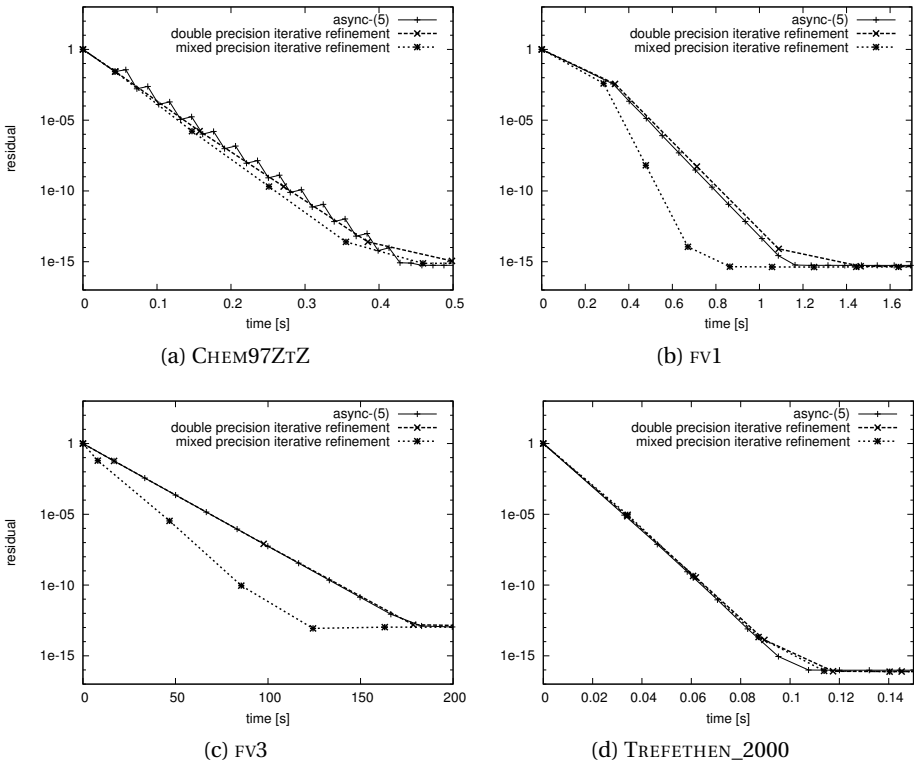
4 Numerical Experiments

In the first experiment, we analyze how using lower precision for the block-asynchronous iteration error correction solver impacts the iterative refinement convergence rate. Therefore, we report the relative residual depending on the iteration number for the different linear equation systems introduced in Section 3. Note that due to the implementation, the first outer loop is used to determine the residual improvement, while the further iterations improve the approximation iterate by 6 to 8 digits, depending on the rounding error.

The results show that for the test matrices CHEM97ZTZ, FV1, and TREFETHEN_2000, using single precision for the error correction solver has a nearly negligible impact on the convergence of the iterative refinement. Only for the FV3 test case, does the convergence rate suffer. This was expected since the high condition number triggers representation errors in the low precision format that make the approximation updates less beneficial.

But while the convergence behaviour is interesting from the theoretical point of view, the next experiment is dedicated to analyzing how handling the error correction equation in single precision impacts the performance. The motivation is that using single instead of double as working precision, triggers at least a speedup of two, and may potentially overcompensate for the overhead associated with the typecast between the formats.

While the convergence, with respect to iteration number, is independent of the hardware used, the performance depends on the architecture. We use the C2070 for this experiment, as this 'Fermi' generation is the state of the art from the Nvidia GPU manufacturer. In addition to the convergence performance of the iterative refinement, using a double or single precision error correction solver, we report the results for the plain block-asynchronous iteration in double precision.



(a) CHEM97ZTZ (b) FV1

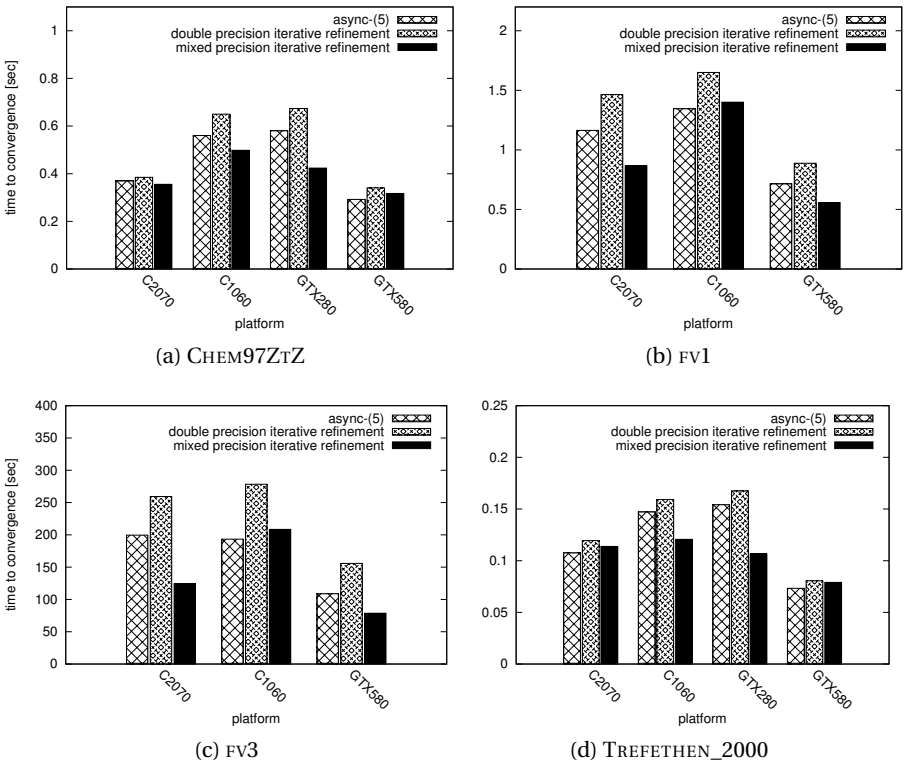
(c) FV3 (d) TREFETHEN_2000

Fig. 5: Iterative refinement performance, time-dependent relative residual.

We observe, that for all test cases, the overhead is negligible when embedding the block-asynchronous iteration in double precision into the iterative refinement framework.

For the small test cases CHEM97ZTZ and TREFETHEN_2000, switching to the mixed precision iterative refinement approach gives no improvement. For the larger matrices e.g. FV1, the improvement by using low precision for the error correction solver is relevant. The mixed precision implementation converges almost twice as fast. Even for the test case FV3, where we observed a slower convergence rate for the mixed precision approach in Figure 4, we benefit in terms of performance.

For the test cases FV1 and FV3, despite the performance difference between single and double precision of around 10 on the Tesla line, the mixed precision iterative refinement performs inferior to the plain double implementation of async-(5). The reason is, that for these systems, the memory bandwidth is the limiting factor and the overhead, due to the iterative refinement framework, can not be compensated for by the single precision performance. For the small matrices, things are different. Since the size of CHEM97ZTZ and TREFETHEN_2000 allows for the caching of the iteration vector as well as the right-hand side, the C1060 and the GTX280 are able to lever-



(a) CHEM97ZTZ (b) FV1

(c) FV3 (d) TREFETHEN_2000

Fig. 6: Total solver runtime.

age the single precision performance more efficiently. Still, the bandwidth remains the limiting factor, since the complete matrix cannot be loaded into cache, and the higher memory bandwidth of the consumer version explains the better performance for the mixed precision approach. Using double precision, the server version is superior, probably due to the more sophisticated memory structure. Unfortunately, the very limited main memory on the GTX280 does not allow for the handling of the large systems.

Note that the total solver runtime for TREFETHEN_2000 is on the GTX280 even smaller than on the server version of the Fermi line. An explanation may be that the overall runtime also includes the initialization process, which has to be taken into account for this system, and is longer for systems using CUDA in version 4.0 and equipped with more memory.

Targeting the Fermi generation, we observe that, especially for the large systems, we benefit from the mixed precision framework. Although we may only expect a factor of two concerning the floating point performance, the sophisticated memory hierarchy enables even higher speedups. This speedup stems from the fact that, not only are we able to keep the iteration vector and the right-hand side local due to the

larger L1 cache, but also because the L2 cache allows for the efficient data access of the iteration matrix.

Note that for the test case FV3, the iterative refinement in double precision fulfills the critical stopping criterion after 4 iterations, while we could observe in Figure 4 that it is already very close after 3 iterations. Hence, the double precision iterative refinement runtime would benefit from choosing a smaller number of inner iterations for the last global iteration.

5 Conclusions

We were able to show that embedding block-asynchronous iteration into a mixed precision iterative refinement method not only retains its convergence properties, but may even be beneficial to the performance. Depending on the GPU architecture, we were able to achieve a performance increase of up to a factor of two for linear equation systems taken from the University of Florida Matrix Collection. The potential lies within Hardware systems that have large differences in the double–single precision performance, and a sophisticated memory hierarchy enabling them to transfer this performance factor to speedups of the asynchronous iteration solver. While our analysis focused on the typically used single- and double precision formats, especially when targeting artificially created extended formats, the mixed precision iterative refinement approach is inevitable. Asde from this, further research should focus on determining a priori, whether embedding the block-asynchronous iteration into the mixed precision iterative refinement framework is beneficial for a certain problem. This may depend not only on the problem characteristics, i.e. the condition number, but also on the hardware platform used.

References

1. H. Anzt, V. Heuveline, and B. Rocker. An error correction solver for linear systems: Evaluation of mixed precision implementations. In *VECPAR'10*, pages 58–70, 2010.
2. H. Anzt, S. Tomov, J. Dongarra, and V. Heuveline. A Block-Asynchronous Relaxation Method for Graphics Processing Units. Technical report, Innovative Computing Laboratory, University of Tennessee, UT-CS-11-687, 2011.
3. H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline. Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems. Technical report, Innovative Computing Laboratory, University of Tennessee, UT-CS-11-689, 2011.
4. U. Aydin and M. Dubois. Generalized asynchronous iterations. pages 272–278, 1986.
5. U. Aydin and M. Dubois. Sufficient conditions for the convergence of asynchronous iterations. *Parallel Computing*, 10(1):83–92, 1989.
6. M. Baboulin, A. Buttari, J. J. Dongarra, J. Langou, J. Langou, P. Luszczek, J. Kurzak, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.
7. Z.-Z. Bai, V. Migallón, J. Penadés, and D. B. Szyld. Block and asynchronous two-stage methods for mildly nonlinear systems. *Numerische Mathematik*, 82:1–20, 1999.
8. A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, S. Martin, and U. Meier Yang. Scaling algebraic multigrid solvers: On the road to exascale. *Proceedings of Competence in High Performance Computing CiHPC 2010*.

9. A. H. Baker, R. D. Falgout, T. V. Kolev, and U. Meier Yang. Multigrid smoothers for ultra-parallel computing, 2011. LLNL-JRNL-435315.
10. D. P. Bertsekas and J. Eckstein. Distributed asynchronous relaxation methods for linear network flow problems. *Proceedings of IFAC '87*, 1986.
11. A. Buttari, J. J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. of High Performance Computing & Applications*, 21(4):457–486, 2007.
12. D. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and Its Applications*, 2(7):199–222, 1969.
13. A. Frommer and D. B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123:201–216, 2000.
14. D. Göddeke and R. Strzodka. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations (part 2: Double precision GPUs). Technical report, Fakultät für Mathematik, TU Dortmund, July 2008. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 370.
15. D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations. *Int. J. of Parallel, Emergent and Distributed Systems*, 22(4):221–256, 2007.
16. C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.
17. NVIDIA Corporation. *Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi*.
18. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 edition, August 2009.
19. NVIDIA Corporation. *CUDA TOOLKIT 4.0 READINESS FOR CUDA APPLICATIONS*, 4.0 edition, March 2011.
20. D. B. Szyld. The mystery of asynchronous iterations convergence when the spectral radius is one. Technical Report 98-102, Department of Mathematics, Temple University, Philadelphia, Pa., October 1998.
21. N. Trefethen. Hundred-dollar, hundred-digit challenge problems. *SIAM News*, 35(1), January 2, 2002. Problem no. 7.